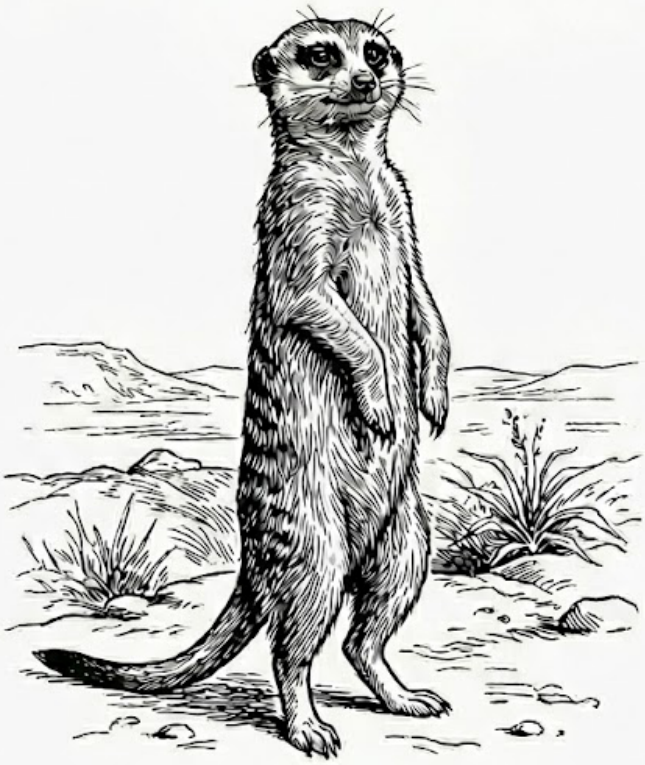
OxiDB

HER ŞEYİN VERİTABANI

Belge, SQL ve Zaman Serisi Motorları
S3 Nesne Depolama ve OxiMem Anahtar-Değer Katmanı



Barış AKIN

İçindekiler

Önsöz **1**

Bu kitap ne anlatıyor . . . 1

Bu kitap kime göre . . . 1

Kitabın adı üzerine . . . 1

Bu kitabın yöntemi . . . 2

Yazardan bir not: OxiDB ve bu kitap . . . 2

Kitabın düzeni . . . 2

1 Veri, Veritabanı ve Veritabanı Yönetim Sistemleri **4**

1.1 Veri nedir, enformasyon nedir . . . 4

1.2 Kalıcılık: neden bellek yetmez . . . 5

1.2.1 Depolama hiyerarşisi: hızın ve unutkanlığın katmanları . . . 5

1.2.2 Diskin acımasız kuralları: blok, sayfa ve sıralı erişim . . . 6

1.3 Neden sadece dosya kullanmıyoruz . . . 6

1.4 Veritabanı ile veritabanı yönetim sistemi . . . 7

1.4.1 Soyutlama katmanları: mantıksal, fiziksel ve aradaki perde . . . 8

1.5 Gömülü kütüphane mi, sunucu mu . . . 8

1.6 Bu kitabın kuracağı zihinsel harita . . . 8

2 Veri Modelleri: Hiyerarşikten Belgeye **9**

2.1 Veri modeli neyi belirler . . . 9

2.2 Hiyerarşik model: dünya bir ağaçtır . . . 10

2.3 Ağ modeli: bağlantılar her yöne . . . 10

2.4 İlişkisel model: büyük ayrışma . . . 11

2.5 İlişkisel modelin gölgesi: parçalanma ve uyumsuzluk . . . 11

2.6 Anahtar-değer modeli: en yalın biçim . . . 12

2.7 Belge modeli: yapılandırılmış bütünler . . . 12

2.8 Diğer modeller ve manzaranın bütünü . . . 13

3 İlişkisel Modelden Belge Modeline: Neden ve Ne Zaman **15**

3.1 Kral değil hizmetkâr: modeli erişim örüntüleri belirler . . . 15
3.2 Normalleştirme: bilgiyi nereye koyacağının disiplini . . . 16
3.3 Belge modeline geçişin dört itici gücü . . . 16
3.4 Belge modelinin parladığı durumlar . . . 17
3.5 Belge modelinin zorlandığı durumlar . . . 17
3.6 Çoğaltma ödünleşimine yeniden bakış . . . 18
3.7 "Ya hep ya hiç" yanılgısı ve melez gerçeklik . . . 18
3.8 Karar vermeden önce sorulacak sorular . . . 19

4 Belge Modeli Derinlemesine: JSON, Şema Esnekliği, Gömme ve Referans **20**

4.1 Bir belge neyden yapılır . . . 20
4.2 Kendini tanımlayan veri . . . 21
4.3 JSON nedir ve nereden geldi . . . 21
4.4 JSON neden kazandı . . . 22
4.5 Belgeler ve koleksiyonlar . . . 23
4.6 "Şemasızlık" yanılgısı: şema-okumada ve şema-yazmada . . . 23
4.7 Belge modelinin kalbindeki karar: gömmek mi, atıfta bulunmak mı . . . 24
4.8 Atomikliğin sınırı belgedir . . . 25

5 Depolama Motorları: Sayfa Tabanlı, Append-Only, LSM ve B-Ağaçları **26**

5.1 Depolama motoru ne yapar . . . 26
5.2 Her şeyi belirleyen kısıt: disk belleğe benzemez . . . 27
5.3 Temel gerilim: okumaya mı, yazmaya mı eniyilemek . . . 27
5.4 Sayfa tabanlı B-ağaçları: yerinde güncelleme . . . 27
5.4.1 Düğümün içi: fanout, dolum oranı, dallanma derinliği . . . 28
5.4.2 Bölme ve birleştirme: dengeyi korumanın bedeli . . . 28
5.5 Append-only ve log-yapılı yaklaşımlar: asla üzerine yazma . . . 29
5.6 LSM ağaçları: log-yapılının olgun hâli . . . 30
5.6.1 İki birleştirme stratejisi: leveled ve size-tiered . . . 30
5.6.2 Okuma yolunu kurtaran iki yardımcı: bloom filtresi ve fence pointer . . . 30
5.7 Üç büyütme: kaçınılmaz üçgen . . . 31
5.8 Anahtar ile değeri ayırmak: yazma büyütmesini kırmak . . . 32
5.9 Belleğin rolü ve veriyi belleğe yansıtmak . . . 32
5.10 Bu bölümün bıraktığı yer ve bir sonrakine köprü . . . 32

6 Dayanıklılık: Write-Ahead Log, fsync ve Çökmeden Kurtarma 34

6.1 "Yazdım" demek neden yetmez . . . 34
6.2 fsync: veriyi gerçekten diske zorlamak . . . 35
6.2.1 Boşaltma çağrılarının ailesi: fsync, fdatasync ve gerçek boşaltma . . . 35
6.3 İkinci tehlike: yarım kalmış yazma . . . 35
6.4 Çözüm: önce niyetini yaz . . . 36
6.4.1 Günlüğe ne yazılır: yineleme, geri-alma ve sıra numarası . . . 36
6.5 Kurtarma: enkazın içinden geri dönmek . . . 37
6.6 Yarım yazmayı yakalamak: sağlama toplamları . . . 38
6.7 Günlük sonsuza dek büyüyemez: denetim noktası . . . 38
6.8 Dayanıklılığın tayfı: katı, gevşek ve grup commit . . . 39
6.9 WAL nereye oturur . . . 39

7 İndeksleme: B-Ağacı İndeksleri, Bileşik İndeksler ve Ters İndeks 41

7.1 Tarama sorunu . . . 41
7.2 İndeks nedir: kitabın arkasındaki dizin . . . 42
7.3 İndeksin bedeli: bedava hız yoktur . . . 42
7.4 Sıralı indeksler: hem eşitlik hem aralık . . . 42
7.5 Seçicilik: her indeks aynı ölçüde işe yaramaz . . . 43
7.6 Bileşik indeksler: birden çok alan birlikte . . . 43
7.7 Kapsayan indeksler: belgeye hiç dokunmamak . . . 44
7.8 Kısmi ve seyrek indeksler: yalnızca işe yarayanı indeksle . . . 44
7.9 Ters indeks: metni aranabilir kılmak . . . 45
7.10 İndeksler de dayanıklı olmalı . . . 46
7.11 Bu bölümün bıraktığı yer . . . 46

8 Sorgu İşleme: Ayrıştırma, Planlama ve İndeks Seçimi 48

8.1 Soru ile yol arasındaki fark . . . 48
8.2 Birinci aşama: ayrıştırma . . . 49
8.3 İkinci aşama: planlama . . . 50
8.4 Birden çok koşulu birleştirmek ve en seçiciyi seçmek . . . 50
8.5 Kardinalite, seçicilik ve histogramlar . . . 50
8.6 Maliyet modeli: planları sayıya dökmek . . . 51
8.7 İndeks yoksa: tarama ve süzgeç . . . 51
8.8 Sıralama, atlama ve sınırlama . . . 52
8.9 Yalnızca gerekeni döndürmek . . . 52
8.10 İki kümeyi eşleştirmek: birleştirme algoritmaları . . . 52

8.11 Yineleyici modeli: planı çalıştıran iskelet . . . 53
8.12 Bildirimsel olmanın asıl kazancı . . . 53
8.13 Bu bölümün bıraktığı yer . . . 54

9 Toplama: Pipeline Modeli, Gruplama ve Pencere Fonksiyonları 55
9.1 Süzmenin ötesi: veriyi özetlemek . . . 55
9.2 Pipeline modeli: montaj hattı . . . 56
9.3 Pipeline'ın aşamaları . . . 56
9.4 Sıranın önemi . . . 56
9.5 Gruplamanın derinliği . . . 57
9.6 Gruplamanın iki gerçekleştirme yolu . . . 57
9.7 Birleştirme aşaması, perde arkasında bir hash birleştirmesidir . . . 58
9.8 Pencere fonksiyonları: satırı yok etmeden hesaplamak . . . 58
9.9 Toplamanın performans doğası . . . 59
9.10 Bu bölümün bıraktığı yer . . . 60

10 İşlemler: ACID, Yalıtım, Kilitleme, MVCC ve OCC 61
10.1 İşlem nedir: bölünmez bir bütün . . . 61
10.2 Dört güvence: ACID . . . 62
10.3 Yalıtımın çözdüğü sorun: eşzamanlılık anormallikleri . . . 62
10.4 Yalıtım düzeyleri: bir tayf . . . 63
10.5 Yalıtımı sağlamanın üç felsefesi . . . 63
10.5.1 Kilitleme: kötümser yaklaşım . . . 63
10.5.2 MVCC: çok sürümlü eşzamanlılık . . . 64
10.5.3 OCC: iyimser yaklaşım . . . 66
10.6 Belge dünyasında işlemler: tek belge kolay, çoğu zor . . . 66
10.7 Bedava güvence yoktur . . . 67
10.8 Bu bölümün bıraktığı yer . . . 67

11 Eşzamanlılık ve Tutarlılık 68
11.1 Neden birden çok kopya . . . 68
11.2 Yeni sorun: kopyalar anlaşmazlığa düşebilir . . . 69
11.3 Tutarlılık tayfı: katıdan gevşeğe . . . 69
11.4 Güçlü tutarlılığı kesinleştirmek: doğrusallaştırılabilirlik . . . 69
11.5 Neden mutlak güçlü tutarlılık her zaman mümkün değil: FLP . . . 71
11.6 Bölünme anı ve kaçınılmaz seçim . . . 71
11.7 Güçlü tutarlılık nasıl sağlanır: otorite ve çoğunluk . . . 71
11.8 Ayarlanabilir tutarlılık . . . 72

11.9 Olayları sıralamak: mantıksal saatler . . . 72
11.10 Tutarlılık ihtiyacı doğruluktan doğar . . . 73
11.11 Bu bölümün bıraktığı yer . . . 73

12 Ölçeklendirme: Replikasyon, Konsensüs ve Sharding 74
12.1 Neden tek makine yetmez . . . 74
12.2 Replikasyon: aynı verinin kopyaları . . . 75
12.3 Lider çökünce: failover ve iki büyük tehlike . . . 75
12.4 Üç topoloji: tek-lider, çok-lider, lidersiz . . . 75
12.5 Yetersayı: W + R > N kuralı . . . 76
12.6 Konsensüs: çoğunlukla anlaşmak . . . 77
12.7 Sharding: veriyi bölmek . . . 78
12.8 Yönlendirme ve parçalar-arası sorgu . . . 79
12.9 İyi bir parça anahtarı ve yeniden dengelemenin yükü . . . 79
12.10 İkisini birleştirmek . . . 80
12.11 Yine aynı ders: dağıtım bedava değildir . . . 80
12.12 Bu bölümün bıraktığı yer . . . 80

13 Bellek, Önbellek ve Disk Ödünleşimi 81
13.1 Temel asimetri ve oyunun özü . . . 81
13.2 Çalışma kümesi ve uçurum etkisi . . . 82
13.3 Önbellek: belleği akıllıca kullanmak . . . 82
13.4 Tahliye: yer açmak için ne atılır . . . 83
13.5 Önbelleğin kendi maliyeti ve sınırlama zorunluluğu . . . 83
13.6 İki felsefe: belleğe öncelikli ve diske öncelikli . . . 84
13.7 Tampon havuzu: veriyi sayfa sayfa yönetmek . . . 84
13.8 İşletim sistemine güvenmek: belleğe yansıtma . . . 85
13.9 Sıkıştırma: yer, işlemci ve sıfır-kopya üçgeni . . . 85
13.10 Bu bölümün bıraktığı yer . . . 86

14 Güvenlik: Kimlik Doğrulama, Yetkilendirme, Şifreleme ve Denetim 87
14.1 Tehdit ve katmanlı savunma ilkesi . . . 87
14.2 Kimlik doğrulama: sen kimsin . . . 88
14.3 Yetkilendirme: ne yapabilirsin . . . 89
14.4 Şifreleme: çalınsa bile okunamaz . . . 90
14.5 Denetim: kim, ne zaman, ne yaptı . . . 91
14.6 Güvenlik bir bütündür ve ödünleşimlerle doludur . . . 91
14.7 Kısım II'nin sonu ve Kısım III'e geçiş . . . 92

15 OxiDB'ye Genel Bakış ve Mimari **93**

15.1 OxiDB nedir . . . 94
15.2 Katmanların kuş bakışı görünümü . . . 94
15.3 Çekirdeğin ötesi: ek yüzeyler . . . 95
15.4 Çalışma alanının yapısı ve istemciler . . . 95
15.5 Bir isteğin yaşam döngüsü . . . 96
15.6 Üçüncü kısmın yol haritası . . . 97

16 OxiDB'nin Depolama Katmanı: In-RAM ve Disk-First, mmap, .bdat / .btree **98**

16.1 Çekirdek: kimlikten baytlara . . . 98
16.2 İki kip, iki felsefe . . . 99
16.2.1 Disk-öncelikli kip: varsayılan . . . 99
16.2.2 Belleğe öncelikli kip: opsiyonel . . . 100
16.3 Dosyaların dili: .btree, .bdat ve .bopts . . . 100
16.4 Belleğe yansıtmanın somut etkisi . . . 100
16.5 Bir kaydın anatomisi: status, uzunluk, yük . . . 101
16.6 Append-only olmanın sonuçları . . . 101
16.7 Sıkıştırma: yer, işlemci ve sıfır-kopya . . . 102
16.8 Açılış ve kurtarmaya kısa bir bakış . . . 102
16.9 Bu bölümün bıraktığı yer . . . 102

17 OxiDB'de WAL, Dayanıklılık ve Kurtarma; Katı ve Gevşek Senkronizasyon **103**

17.1 OxiDB'nin yazma-öncesi günlüğü . . . 103
17.2 Katı dayanıklılık: varsayılan tercih . . . 104
17.3 Grup commit: bedeli toplu yazmada eritmek . . . 105
17.4 Üç aşamalı dayanıklılık: günlük, veri, denetim noktası . . . 105
17.5 Gevşek senkronizasyon: hızı seçmek . . . 106
17.6 Kurtarma: çökmeden geri dönmek . . . 106
17.7 Denetim noktası ve günlüğün dizginlenmesi . . . 107
17.8 Günlüğün döndürülmesi ve mühürlü segmentler . . . 107
17.9 WAL her iki kibin de önünde durur . . . 108
17.10 Bu bölümün bıraktığı yer . . . 108

18 OxiDB'de İndeksler: Alan, Bileşik ve mmap Tabanlı Disk İndeksleri **109**

18.1 Alan indeksleri: sıralı bir eşleme . . . 109
18.2 Türler arası sıralama: IndexValue inceliği . . . 110
18.3 Bileşik indeksler ve önek kuralı . . . 111
18.4 İndeksten yanıt: belgeye dokunmadan saymak . . . 111

18.5 İndeks destekli sıralama ve erken sonlanma . . . 111
18.6 Disk-öncelikli indeksler: indeksi de bellekten çıkarmak . . . 112
18.7 Diğer indeks türleri . . . 112
18.8 İndekslerin bedeli ve dayanıklılığı . . . 112
18.9 Bu bölümün bıraktığı yer . . . 113

19 OxiDB'nin Sorgu Motoru: Operatörler, İndeks Destekli Yollar ve Bayt Düzeyinde Süzme 114
19.1 Sorgunun ayrıştırılması ve operatör dağarcığı . . . 114
19.2 İndeksli koşullar ile süzgeç koşulları . . . 115
19.3 Yürütme: indeks, süzgeç ve tarama . . . 115
19.4 Seçicilik temelli koşul sıralaması . . . 116
19.5 İki indekssiz hızlı yol: sayım ve indeks destekli sıralama . . . 117
19.6 Bayt düzeyinde süzme: çözmeden elemek . . . 117
19.7 Tekil işlemlerde erken sonlanma . . . 118
19.8 MongoDB uyumlu güncelleme anlamları . . . 118
19.9 Sunucu yoluyla ilişki . . . 119
19.10 Bildirimsel özgürlüğün yansıması . . . 119
19.11 Bu bölümün bıraktığı yer . . . 119

20 OxiDB'nin Toplama Pipeline'ı: Gruplama, $facet ve Pencere Fonksiyonları 121
20.1 Pipeline ve aşamaları . . . 121
20.2 Baştaki süzmeyi öne çekmek . . . 122
20.3 Akış halinde gruplama . . . 122
20.4 Çok-yönlü analiz: $facet . . . 122
20.5 Pencere fonksiyonları . . . 123
20.6 Zaman-serisi aşamaları ve boşluksuz zincir . . . 124
20.7 Dürüst bir envanter: henüz olmayan aşamalar . . . 124
20.8 Toplamanın performans gerçeği: disk-öncelikli durum . . . 125
20.9 Bu bölümün bıraktığı yer . . . 125

21 OxiDB'de İşlemler: İyimser Eşzamanlılık ve Üç Fazlı Commit 126
21.1 OxiDB neden iyimser yolu seçti . . . 126
21.2 İyimser akışın üç fazı . . . 127
21.3 Doğrulama ile uygulamanın bölünmezliği . . . 127
21.4 Dayanıklılıkla bağ . . . 128
21.5 Okuma anlık görüntüleri: yazarı bekletmeden tutarlı okuma . . . 128
21.6 Kilitlenmeye karşı tasarımla bağışıklık . . . 129
21.7 Tek belge ile çok belge . . . 130

21.8 Küme durumunda işlemler . . . 130
21.9 İyimserliğin bedeli . . . 130
21.10 Kötümser bir kaçış: satır kilidiyle okuma . . . 130
21.11 Bu bölümün bıraktığı yer . . . 130

22 OxiDB'de Sıkıştırma: Ölü Alan ve Otomatik Tetikleme 132
22.1 Ölü alan nereden gelir . . . 132
22.2 Sıkıştırma ne yapar . . . 133
22.3 Zor kısım: yaşayan sistemde güvenle yapmak . . . 133
22.4 İndeksler neden sağ kalır . . . 134
22.5 Açık çağrıdan otomatik tetiklemeye . . . 134
22.6 Kütlesel silmenin gizli tuzağı . . . 135
22.7 Sıkıştırmanın bedeli ve dengesi . . . 135
22.8 Bu bölümün bıraktığı yer . . . 135

23 OxiDB'nin Ek Yüzeyleri: Tam Metin Arama, Blob Depolama, Şifreleme ve PITR 137
23.1 Tam metin arama: ters indeksin somutu . . . 137
23.2 Tam metin aramanın iç mekaniği . . . 138
23.3 Blob depolama: büyük ikili nesnelere yer açmak . . . 139
23.4 Dururken şifreleme: depolama sınırında koruma . . . 139
23.5 Zaman-noktasına kurtarma: çökmenin ötesinde . . . 140
23.6 Ortak iplik: tek motor, isteğe bağlı yetenekler . . . 140
23.7 Bu bölümün bıraktığı yer . . . 141

24 OxiDB'nin Sunucu Katmanı: OxiWire Protokolü, Kimlik Doğrulama, RBAC ve Denetim 142
24.1 Çerçeveleme sorunu ve OxiWire protokolü . . . 142
24.2 Bağlantı modeli ve el sıkışma . . . 143
24.3 Kimlik doğrulama: SCRAM ile parolayı hattan geçirmeden . . . 143
24.4 Yetkilendirme: rol tabanlı geçit . . . 144
24.5 Çok veritabanı: tek sunucuda yalıtılmış dünyalar . . . 144
24.6 Aktarım şifrelemesi ve denetim . . . 145
24.7 Gözlemlenebilirlik: açıklama, yavaş sorgu ve ölçümler . . . 145
24.8 İsteğin sunucudaki yolu . . . 146
24.9 Bu bölümün bıraktığı yer . . . 146

25 OxiDB'de Ölçeklendirme: Raft Kümesi ve OxiPool ile Sharding **147**

25.1 Raft kümesi: replikasyon ve konsensüs . . . 147
25.2 Kümenin doğrulanması . . . 148
25.3 OxiPool: sharding ile veriyi bölmek . . . 148
25.4 Dürüst sınırlar . . . 150
25.5 İkisini birleştirmek . . . 150
25.6 Bu bölümün bıraktığı yer . . . 150

26 Uyumluluk Katmanları ve İstemciler **151**

26.1 Çekirdeğe iki kapı . . . 151
26.2 Diğer dillere köprü: FFI . . . 152
26.3 İstemci kütüphaneleri . . . 152
26.4 Uyumluluk katmanları: başka protokolleri konuşmak . . . 152
26.5 Tek çekirdek, çok yüz . . . 153
26.6 Bu bölümün bıraktığı yer . . . 154

27 Bellek Optimizasyonu ve Karşılaştırmalı Değerlendirme **155**

27.1 Belleği yerleşik yığından çıkarma yolculuğu . . . 155
27.2 Bellek ölçümünün dürüst yüzü . . . 156
27.3 Karşılaştırmanın felsefesi . . . 156
27.4 OxiDB nerede kazanır . . . 157
27.5 OxiDB nerede berabere kalır . . . 157
27.6 OxiDB nerede kaybeder ve neden . . . 157
27.7 Karşılaştırmanın asıl dersi: dayanıklılık merceği . . . 158
27.8 İlkelerin bir bütün oluşturması . . . 158
27.9 Bu bölümün ve ana metnin sonu . . . 158

28 OxiDB'nin İkinci Motoru: SQL **160**

28.1 Neden ikinci bir motor? . . . 160
28.2 Mimari: katalog, satır deposu, günlük, denetim noktası . . . 161
28.3 SQL yüzeyi: şemadan sorguya . . . 161
28.4 Birleştirmeler ve gruplama . . . 163
28.5 Analitik yüzey . . . 164
28.6 İndeksler ve sorgu hızlandırma . . . 166
28.7 İşlemler . . . 167
28.8 Saklı yordamlar: iki dil . . . 168
28.9 İstemciler . . . 169
28.10 EF Core: sağlayıcı, göç, iskele . . . 170
28.11 Performans . . . 170
28.12 Sınırlar ve kapanış . . . 171

29 Zaman Serisi Motoru: oxidb-tsdb **172**

29.1 Zaman serisi verisinin doğası . . . 172
29.2 Seri kimliği: ölçüm × etiket kümesi × alan . . . 173
29.3 Gorilla: bir noktayı iki bite indirmek . . . 173
29.3.1 Zaman damgaları: delta-of-delta . . . 173
29.3.2 Değerler: XOR . . . 174
29.3.3 Somut bit hesabı . . . 174
29.4 Bloklar, aktif tampon ve saklama . . . 175
29.5 Kalıcılık: MANIFEST tek commit noktasıdır . . . 175
29.6 Tipli alanlar . . . 175
29.7 Motoru açmak . . . 176
29.8 Yazmak . . . 176
29.8.1 InfluxDB satır protokolü . . . 177
29.9 Sorgulamak . . . 178
29.9.1 Toplama türleri . . . 179
29.10 Saklama ve kontrol noktası . . . 180
29.11 Sürekli toplama: rollup . . . 180
29.12 Gerçek kullanım: tick'ten muma . . . 181
29.13 Sınırlar ve motorun yeri . . . 182

30 OxiMem: Redis'in Telini Konuşan Bellek-İçi Anahtar-Değer Katmanı **183**

30.1 Neden Redis'in telini konuşmak? . . . 183
30.2 Açmak: bir port . . . 184
30.3 Komut ailesi . . . 184
30.4 TTL ve süre dolumu . . . 186
30.5 MULTI, EXEC ve WATCH: aynı fikir, küçük ölçekte . . . 187
30.6 EXECABORT: hataları erken yakalamak . . . 189
30.7 Sunucu tarafı betikler ve atomik borç düşme . . . 189
30.8 Gerçek bir desen: hız sınırlama . . . 190
30.9 Tek bir keyspace — bilinçli bir tercih . . . 191
30.10 Kalıcılık: iki seçenek, ikisi de isteğe bağlı . . . 191
30.11 Performans: Redis'e ne kadar yakın? . . . 192
30.12 Eksikler, sınırlar ve ne zaman hangisi . . . 192

31 S3 Uyumlu Nesne Depolama: Büyük Baytların Yeri **194**

31.1 Bir veritabanı neden nesne depolar? . . . 194
31.2 Kova ve nesne modeli . . . 195
31.3 Disk üzerindeki düzen: `.data` ve `.meta` . . . 195
31.4 Sunucuyu S3 için açmak . . . 196
31.5 Kimlik doğrulama: AWS Signature V4 . . . 196
31.6 Hangi S3 operasyonları uygulanmış? . . . 196
31.7 Hazır ekosistem: uyumluluğun asıl getirisi . . . 197
31.8 boto3 ile: istemcinin kurulumu ve tam bir tur . . . 198
31.9 Diğer diller: .NET, JavaScript, Go . . . 201
31.10 Yerel yüzey: wire komutları . . . 202
31.11 Asıl desen: belgede üst veri, nesne deposunda baytlar . . . 202
31.12 MinIO ile karşılaştırma . . . 204
31.13 Karar rehberi: ne zaman blob, ne zaman belge? . . . 204
31.14 Özet . . . 205

32 Motorları Birlikte Kullanmak: Tek Sunucu, Beş Yüzey, Tek Uygulama **206**

32.1 Motor seçimi: veri şekli, doğru motoru söyler . . . 206
32.2 Aynı sunucu, ayrı isim uzayları . . . 207
32.3 Sunucuyu ayağa kaldırmak . . . 208
32.4 Uygulama: OxiTrade borsası . . . 208
32.4.1 Adım 1 — Kullanıcılar: belge motoru . . . 209
32.4.2 Adım 2 — Emirler ve defter: SQL motoru . . . 209
32.4.3 Adım 3 — Fiyat akışı: TSDB motoru . . . 210
32.4.4 Adım 4 — Mumlar: sürekli toplama (rollup) . . . 211
32.4.5 Adım 5 — Belgeler ve görseller: nesne depolama . . . 212
32.4.6 Adım 6 — Sıcak durum: OxiMem . . . 213
32.4.7 Adım 7 — Raporlama: SQL'in evinde . . . 214
32.5 Sınırlar: motorlar arası atomiklik yoktur . . . 215
32.6 İşletim: tek sunucu, ayrı ömürler . . . 216
32.7 Bu bölümün bıraktığı yer . . . 217

Ek A — Sözlük **218**

Ek B — Kaynaklar **223**

32.8 Genel: veri-yoğun sistemlerin temelleri . . . 223
32.9 Veri modelleri ve ilişkisel temel . . . 223
32.10 JSON ve belge biçimleri . . . 223
32.11 Depolama motorları ve indeksleme . . . 223
32.12 İşlemler, eşzamanlılık ve tutarlılık . . . 224
32.13 Ölçeklendirme ve konsensüs . . . 224
32.14 OxiDB'nin kendi kaynakları . . . 224
32.15 Son bir söz . . . 224
32.16 Kaynakça . . . 224

Önsöz

Bir veritabanı, ilk bakışta sıradan bir araçtır: veriyi koyarsınız, sonra geri alırsınız. Oysa bu basit vaadin altında, bilgisayar biliminin en zarif ve en zorlu problemlerinden bazıları yatar. Elektrik kesildiğinde verinin kaybolmaması nasıl garanti edilir? Binlerce kullanıcı aynı anda aynı kaydı değiştirmeye kalktığında düzen nasıl korunur? Milyonlarca belge arasından aradığınız tek kaydı, diski baştan sona taramadan saniyenin binde birinde nasıl bulursunuz? Veri tek bir makineye sığmaz hale geldiğinde, onu nasıl bölersiniz ve böldüğünüzde tutarlılığı nasıl sürdürürsünüz? Bu kitap, işte bu soruların peşinden gider.

Bu kitap ne anlatıyor

Kitap iki yolculuğu birleştirir. Birincisi **kavramsal** yolculuktur: belge veritabanlarının ne olduğunu, neden var olduklarını ve içeride nasıl çalıştıklarını temelden kurar. Veriyle başlar; veriyi kalıcı kılmanın neden zor olduğunu, veri modellerinin tarih boyunca nasıl evrildiğini, belge modelinin bu evrimde nereye oturduğunu anlatır. Ardından bir belge veritabanının iç mimarisini katman katman açar: verinin diske nasıl yazıldığı, çökmeden nasıl kurtarıldığı, indekslerin aramayı nasıl hızlandırdığı, sorguların nasıl işlendiği, işlemlerin tutarlılığı nasıl koruduğu ve sistemin tek makinenin ötesine nasıl ölçeklendiği.

İkinci yolculuk **somut** olandır: bu kavramların gerçek bir sistemde nasıl hayata geçtiğini, OxiDB adlı bir belge veritabanı motoru üzerinden adım adım gösterir. Soyut bir ilkeyi öğrendikten sonra, onun gerçek bir mühendislik kararına nasıl dönüştüğünü, hangi ödünleşimlerin yapıldığını ve neden öyle yapıldığını görürsünüz. Böylece kitap yalnızca “ne” değil, “neden böyle” sorusunu da yanıtlar.

Bu kitap kime göre

Bu kitap, veritabanlarını kullanan ama içlerinde ne olup bittiğini merak eden yazılım geliştiricileri; sistemlerin nasıl kurulduğunu derinlemesine anlamak isteyen mühendisler; ve bir veri sisteminin altındaki ilkeleri kavramak isteyen öğrenciler için yazıldı. Belirli bir programlama diline ya da belirli bir ürünün kullanımına hâkim olmanız gerekmiyor. Bilgisayarların nasıl çalıştığına dair temel bir aşinalık — bir dosyanın diskte durduğunu, belleğin geçici olduğunu, bir programın komutları sırayla yürüttüğünü bilmek — yeterlidir.

Kitabın adı üzerine

Fizikçiler bir asırdır *her şeyin teorisi* peşinde koşar: doğanın birbirinden kopuk görünen kuvvetlerini tek bir çerçevede toplayan bir formül. Bu kitabın adı, o arayışa göz kırpar — ama bir iddia olarak değil, bir soru olarak. Veri dünyasında da uzun süre benzer bir bölünmüşlük yaşandı: belgeler bir sistemde, tablolar başkasında, ölçüm akışları bir üçüncüsünde, dosyalar bir nesne deposunda, sıcak geçici durum ise bir önbellek sunucusunda durdu. Her biri kendi protokolünü, kendi işletim yükünü, kendi yedekleme planını getirdi.

Her şeyin veritabanı diye bir şey var mıdır? Dürüst yanıt: yoktur — ve bu kitap size bunun neden böyle olduğunu, her modelin hangi ödünleşimin bedelini ödediğini anlatacak. Ama bir sistemin, bu farklı veri şekillerinin her birine kendi doğasına uygun bir motorla karşılık verip hepsini tek bir çatı altında, tek bir bağlantı ve tek bir işletim disiplini içinde sunması mümkündür. OxiDB'nin yaptığı budur: belge motoru, ilişkisel SQL motoru ve zaman serisi motoru; yanına S3 uyumlu nesne depolama ve bellek-içi anahtar-değer katmanı. Tek formül değil, bir arada yaşamayı bilen birkaç motor. Kitabın adı, o arayışın hem hedefini hem de sınırını anmak içindir.

Bu kitabın yöntemi

Kitabın ilk iki kısmında **örnek kod yoktur**. Bu bilinçli bir tercihtir: amaç, sizi belirli bir sözdizimine bağlamadan kavramların kendisini anlatmaktır. Bir fikrin özünü düz metinle, benzetmelerle ve adım adım akıl yürütmeyle anlatmak, çoğu zaman bir kod parçasından daha kalıcı bir kavrayış bırakır. Kod ezberlenir ve unutulur; bir mekanizmanın *neden* öyle çalıştığını bir kez gerçekten anladığınızda ise o bilgi sizinle kalır.

Üçüncü kısımda ise tutum değişir. Orada artık soyut bir ilkeyi değil, çalışan bir sistemi anlatıyoruz; ve bir motorun yüzeyini — hangi sorguyu kabul ettiğini, hangi yanıtı döndürdüğünü, bir işlemin nasıl açılıp kapandığını — sözcüklerle tarif etmek, onu göstermekten hem daha uzun hem daha bulanıktır. Bu yüzden üçüncü kısımda **bol örnek** bulacaksınız: tel üzerindeki JSON istekleri, SQL ifadeleri, Python, C#, JavaScript ve kabuk komutları. Bu örnekler metnin yerine geçmez; kavram önce düzyazıyla kurulur, örnek onu somutlaştırır. Hepsi de sistemin o an çalışan sürümüne karşı denenmiştir.

Kitap kademeli ilerler. Her kavram, kendinden öncekilerin üzerine kurulur: dayanıklılığı anlamadan işlemleri, indekslemeyi anlamadan sorgu işlemeyi tam kavrayamazsınız. Bu yüzden bölümleri sırayla okumanız önerilir. Yine de her bölüm, kendi başına da anlamlı olacak şekilde, gerektiğinde önceki kavramları kısaca hatırlatarak yazılmıştır.

Yazardan bir not: OxiDB ve bu kitap

Bu kitabın üçüncü kısmında adım adım incelenen sistem, OxiDB, benim — Barış AKIN'ın — sıfırdan tasarlayıp yazmaya başladığım bir veritabanı motorudur. OxiDB'yi yazmaya başladığım günden bu yana, bir veritabanının içinde verdiğim her kararın — verinin diske nasıl yerleştiğinden bir çökmeden nasıl geri dönüleceğine, bir indeksin nasıl yapılandırılacağından bir işlemin tutarlılığının nasıl korunacağına kadar — kitabın ikinci kısmında anlatılan o soyut ilkelerin somut bir karşılığı olduğunu gördüm. Bu kitap, işte o deneyimden doğdu.

Önemli bir noktanın altını çizmek isterim: bu kitap, OxiDB'nin **gerçek kod tabanına** (codebase) dayanılarak yazılmıştır. Üçüncü kısımda OxiDB hakkında okuyacağınız her mekanizma — depolama biçimleri, yazma-öncesi günlüğün kayıt düzeni, indeks yapıları, iyimser eşzamanlılık denetimi, sıkıştırma, küme replikasyonu — bir tasarım niyetinden ya da tanıtım metninden değil, sistemin o an çalışan kaynağından alınmış ve ona karşı doğrulanmıştır. Bir ödünleşim anlatıldığında o ödünleşim gerçekten verilmiştir; bir sayı verildiğinde o sayı gerçekten ölçülmüştür. Amacım, kavram ile çalışan kod arasındaki mesafeyi tümüyle kapatmaktı — okuduğunuz ilke ile onu hayata geçiren satırlar arasında hiçbir kopukluk kalmasın diye.

Kitabın düzeni

Kitap üç kısma ayrılır. **Birinci Kısım** temelleri kurar: veri ve veritabanı kavramı, veri modellerinin tarihi ve belge modelinin doğası. **İkinci Kısım**, herhangi bir belge veritabanının içeride nasıl çalıştığını genel ilkeler düzeyinde anlatır — depolama, dayanıklılık, indeksleme, sorgu, işlemler ve ölçeklendirme. **Üçüncü Kısım**, tüm bu ilkelerin OxiDB'de nasıl somutlaştığını adım adım gösterir.

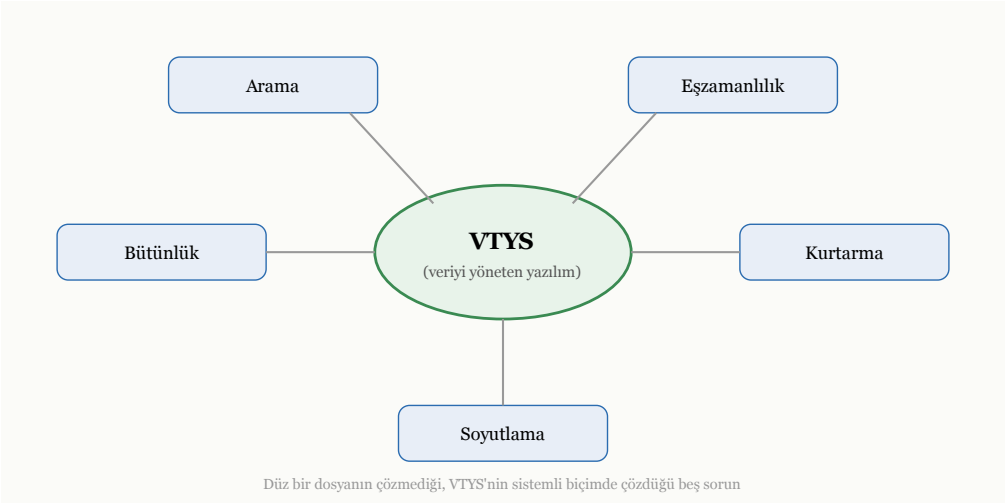
Üçüncü kısmın son bölümleri, kitabın adındaki iddiayı sınar. Belge motorunu enine boyuna inceledikten sonra, aynı sunucunun içinde yaşayan diğer motorlara geçeriz: satırları, tabloları ve join'leri olan **ilişkisel SQL motoru**; saniyede yüz binlerce ölçümü sıkıştırarak yutan **zaman serisi motoru**; büyük ikili nesneler için **S3 uyumlu nesne depolama**; ve sıcak, geçici durum için Redis'in telini konuşan **bellek-içi anahtar-değer katmanı**. Kapanış bölümü hepsini tek bir çalışan uygulamada buluşturur ve asıl soruyu yanıtlar: hangi veri hangi motora gider, ve neden?

Şimdi başlayalım. İlk durağımız, sandığınızdan daha derin bir soru: aslında bir veritabanı tam olarak nedir ve onu basit bir dosyadan ayıran nedir?

Bölüm 1

Veri, Veritabanı ve Veritabanı Yönetim Sistemleri

Her veritabanı, son derece mütevazı bir ihtiyaçtan doğar: bir şeyi hatırlamak. Bir bilgiyi bugün kaydedip yarın, hatta yıllar sonra geri almak isteriz. Bu ihtiyaç o kadar temeldir ki çoğu zaman üzerinde düşünmeyiz; oysa onu güvenilir biçimde karşılamak, bilgisayar mühendisliğinin en derin problemlerinden birini açar. Bu bölümde, bir veritabanını bir avuç dosyadan ayıran şeyin ne olduğunu temelden kuracağız. Henüz "belge" veritabanından söz etmeyeceğiz; önce her veritabanının çözmek zorunda olduğu evrensel sorunları anlamamız gerekiyor.



Şekil 1.1: Bir VTYS'nin düz dosyadan farkı: sistemli biçimde çözdüğü beş temel sorun.

1.1 Veri nedir, enformasyon nedir

İşe en baştan, "veri" sözcüğünden başlayalım. Veri, ham gerçeklerin kaydedilebilir biçimidir: bir sayı, bir ad, bir tarih, bir ölçüm. Tek başına veri sessizdir; anlamı, ona bir bağlam verdiğimizde belirir. "37" bir veridir; onun bir hastanın vücut sıcaklığı olduğunu söylediğimizde enformasyona, yani anlam taşıyan veriye dönüşür. Veritabanı, ham veriyi bağlamıyla birlikte, yani hangi sayının neyi ifade ettiğini de koruyacak şekilde saklamakla görevlidir.

Bu ayrım önemlidir, çünkü bir veritabanının değeri yalnızca sakladığı baytlardan değil, o baytları **yapılandırma** biçiminden gelir. Aynı "37" sayısı, bir tablodaki "sıcaklık" sütununda durduğunda ya da bir belgedeki `sicaklik` alanında durduğunda anlamını kazanır. Verinin nasıl yapılandırıldığı — buna veri modeli diyeceğiz ve bir sonraki bölümün konusu olacak — bir veritabanının kimliğini belirleyen en önemli karardır.

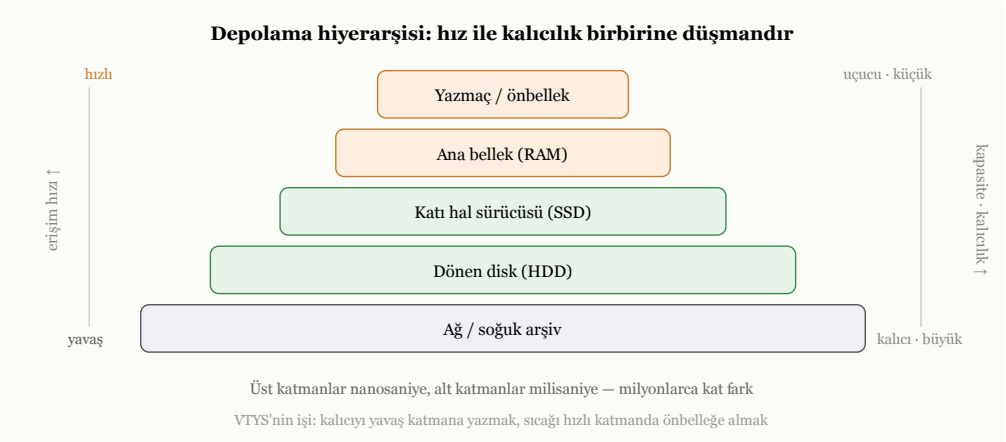
1.2 Kalıcılık: neden bellek yetmez

Bir program çalışırken verisini bellekte (RAM'de) tutar. Bellek hızlıdır ve kullanışlıdır, ama bir kusuru vardır: **uçucudur**. Elektrik kesildiğinde, program kapandığında ya da makine yeniden başladığında bellekteki her şey silinir. Oysa hatırlamak istediğimiz veri, programın ömründen uzun yaşamak zorundadır. Bir bankanın hesap bakiyesi, uygulama kapandı diye sıfırlanamaz.

Çözüm açıktır: veriyi **kalıcı depolamaya**, yani diske yazmak. Disk, elektrik gittiğinde içeriğini korur. Ama bu çözüm, beraberinde yeni bir dünya dolusu problem getirir; çünkü disk, belleğe hiç benzemez. Bellek nanosaniyeler mertebesinde erişilirken, geleneksel bir diskte bir konuma erişmek milisaniyeler alabilir — milyonlarca kat yavaş. Modern katı hal sürücüleri (SSD'ler) bu uçurumu daraltsa da, disk hâlâ belleğe kıyasla kat kat yavaştır ve çok farklı kurallarla çalışır. Disk, veriyi büyük bloklar halinde okuyup yazar; küçük, dağınık erişimleri sevmez; ve en önemlisi, bir yazma işleminin gerçekten fiziksel olarak tamamlanıp tamamlanmadığı, sandığınızdan çok daha incelikli bir sorudur. İkinci kısımda bu inceliğin üzerine uzun uzun eğileceğiz; şimdilik şunu aklınızda tutun: veriyi kalıcı kılmak, onu yalnızca diske göndermek değil, oraya **gerçekten ulaştığından emin olmaktır**.

1.2.1 Depolama hiyerarşisi: hızın ve unutkanlığın katmanları

Bellek ile disk arasındaki bu uçurum, aslında izole iki nokta değil; bir **depolama hiyerarşisinin** (storage hierarchy) iki ucudur. Bir bilgisayarda veri, bir piramidin katmanlarında yaşar ve her katman, hız ile kalıcılık, maliyet ile kapasite arasında farklı bir denge tutar. En tepede, işlemcinin içindeki yazmaçlar (register) ve önbellekler (cache) durur; bunlara erişim nanosaniyenin de altındadır, ama kapasiteleri kilobaytlar mertebesindedir ve uçucudurlar. Bir basamak aşağıda ana bellek (RAM) vardır: yine nanosaniyeler mertebesinde, gigabaytlar kapasiteyle, ama yine uçucu. Onun altında, ilk kalıcı katman olarak katı hal sürücüsü gelir; erişimi onlarca mikrosaniye sürer. En altta dönen disk (manyetik HDD) ve ondan da uzakta ağ üzerindeki ya da soğuk arşivdeki depolama bulunur; bunların erişimi milisaniyeler, hatta saniyeler alabilir.



Şekil 1.2: Depolama hiyerarşisi: yukarı çıktıkça hız artar, aşağı indikçe kalıcılık ve kapasite.

Bu hiyerarşinin değişmez kuralı şudur: bir katman ne kadar hızlıysa, o kadar pahalı, o kadar küçük ve neredeyse her zaman o kadar uçucudur. Hız ile kalıcılık doğada birbirine düşmandır. İşte bir VTYS'nin işinin

büyük kısmı, bu düşmanlığı yönetmektir: veriyi kalıcı olması için yavaş katmana yazmak, ama erişimi hızlı olsun diye sık kullanılanları hızlı katmanda **önbelleğe almak**. Bir veritabanının iç mimarisinde tekrar tekrar göreceğimiz örüntü budur; sıcak veriyi bellekte tutmak, soğuk veriyi diske bırakmak, ve ikisi arasında doğru sınırı çizmek.

Erişim **gecikmesinin** (latency) bu katmanlar arasında milyonlarca kat değişmesinin somut bir sonucu vardır: bir veritabanının performansını belirleyen şey, ham hesaplama gücünden çok, **veriye nereden ve kaç kez erişildiğidir**. Bir sorgunun bir milyon kez belleğe mi yoksa bin kez diske mi gittiği, çoğu zaman saniyelerle mikrosaniyeler arasındaki farktır. Bu yüzden veritabanı tasarımının neredeyse tüm sanatı, yavaş katmana yapılan erişimleri azaltmaya — indekslerle, önbelleklerle, veriyi bir arada tutmakla — adanmıştır.

1.2.2 Diskin acımasız kuralları: blok, sayfa ve sıralı erişim

Diskin belleğe benzemeyişi yalnızca yavaşlığından ibaret değildir; çalışma **granülerliği** de farklıdır. Bellekte tek bir baytı okuyup yazabilirsiniz. Disk ise veriyi yalnızca sabit boyutlu **bloklar** (block) ya da **sayfalar** (page) halinde — tipik olarak dört kilobaytlık birimlerle — okur ve yazar. Tek bir baytı değiştirmek isteseniz bile, donanım o baytın içinde bulunduğu tüm bloğu okur, değiştirir ve geri yazar. Buna **oku-değiştir-yaz** (read-modify-write) döngüsü denir ve küçük, dağınık güncellemelerin neden bu kadar pahalı olduğunun kökünde bu yatar.

Buradan iki temel ders çıkar; ikinci kısımdaki depolama motoru tasarımı baştan sona bu iki derse dayanır. Birincisi, **sıralı erişim, rastgele erişimden kat kat hızlıdır**. Diskte arka arkaya duran blokları okumak, oraya buraya sıçrayarak okumaktan çok daha ucuzdur; dönen disklerde okuma kafasının fiziksel hareketi yüzünden, SSD'lerde ise iç paralelliğin ancak ardışık erişimde doyurulabilmesi yüzünden. Bu yüzden iyi bir depolama motoru, veriyi mümkün olduğunca **ardışık** yazmaya çalışır; altıncı bölümde göreceğimiz "ekleme-only" (append-only) günlük yapısının arkasındaki sezgi tam olarak budur. İkincisi, yazmaları **gruplamak** (batching) tek tek yazmaktan ucuzdur, çünkü blok başına düşen sabit maliyet birçok kayda bölünür.

Bütün bunların üstüne bir de bellekle disk arasında oturan birçok ara katman biner: işletim sisteminin sayfa önbelleği, diskin kendi tampon belleği, hatta sürücünün üzerindeki uçucu önbellek. Bu katmanlar performansa yardım eder ama dayanıklılık açısından bir tuzak kurar: "diske yaz" dediğinizde veri çoğu zaman yalnızca bu uçucu tamponlardan birine ulaşır, henüz kalıcı yüzeye işlenmemiştir. Veriyi gerçekten kalıcı kılmak için, sistemden bu tamponların tümünü kalıcı katmana **boşaltmasını** açıkça istemek gerekir — bu isteğe genellikle `fsync` denir ve pahalıdır, çünkü tüm o hızlı ara katmanları atlayıp en yavaş, en kalıcı yüzeye inmeyi zorlar. Altıncı bölümde dayanıklılığın bedelinin neden büyük ölçüde bu tek çağrıda toplandığını ayrıntısıyla göreceğiz. Bölümün başında belirttiğimiz o ilke — veriyi kalıcı kılmanın, onu diske göndermek değil oraya gerçekten ulaştığından emin olmak demek olduğu — işte bu ara katmanlar yüzünden bu kadar incelik taşır.

1.3 Neden sadece dosya kullanmıyoruz

Mademki veriyi diske yazacağız, neden bir veritabanına ihtiyaç duyalım? Sonuçta işletim sistemi bize dosyalar sunar; veriyi bir dosyaya yazıp gerektiğinde okuyabiliriz. Birçok basit uygulama tam olarak bunu yapar ve bu yeterlidir. Sorunlar, veri büyüdükçe, birden çok kullanıcıyla paylaşıldıkça ve üzerinde karmaşık sorular sormaya başladıkça ortaya çıkar. Bir dosyayla baş başa kaldığınızda, aşağıdaki sorunların her biriyle tek tek, elle uğraşmak zorunda kalırsınız.

Arama sorunu. Diyelim ki bir dosyada bir milyon müşteri kaydı var ve belirli bir e-posta adresine sahip olanı arıyorsunuz. Yapısı olmayan bir dosyada tek seçeneğiniz, dosyayı baştan sona okumaktır. Bir milyon kaydı taramak yavaştır; üstelik aramayı her tekrarladığınızda baştan taramanız gerekir. Bir veritabanı, indeks adı verilen yardımcı yapılarla bu sorunu çözer: aradığınız kaydın yerini, dosyayı taramadan, doğrudan bulmanızı sağlar. Yedinci bölüm tümüyle bu mekanizmaya ayrılmıştır.

Eşzamanlılık sorunu. Tek bir kullanıcı varken her şey basittir. Ama iki kişi aynı anda aynı kaydı değiştirmeye kalktığında ne olur? Biri kaydı okur, üzerinde çalışırken diğeri aynı kaydı değiştirir ve kaydeder; sonra

ilk kişi kendi değişikliğini kaydeder ve diğerinin değişikliğini hiç haberi olmadan ezer. Buna "kayıp güncelleme" denir ve düz dosyalarla çalışırken sessizce, fark edilmeden gerçekleşir. Bir veritabanı, eşzamanlı erişimi düzene sokan mekanizmalarla bu tür bozulmaları engeller. Onuncu ve on birinci bölümler bu konunun derinine iner.

Bütünlük sorunu. Verinin yalnızca saklanması değil, **tutarlı** kalması da gerekir. Bir bankada para transferi, bir hesaptan düşme ve diğerine ekleme olmak üzere iki adımdan oluşur. Bu iki adımın arasında sistem çökerse, para bir hesaptan çıkmış ama diğerine girmemiş olabilir — yani buharlaşmıştır. Ya iki adım da olmalı ya da hiçbiri olmamalıdır; arada bir durum kabul edilemez. Düz dosyalarla bu "ya hep ya hiç" garantisini kurmak son derece zordur. Veritabanı, işlem (transaction) kavramıyla bunu sistematik biçimde sağlar.

Çökmeden kurtulma sorunu. Sistem tam bir yazma işleminin ortasında çökerse ne olur? Dosyanın yarısı yeni, yarısı eski veriyle, belki de bir kısmı bozuk kalabilir. Bu, soyut bir kaygı değil; somut bir başarısızlık kipidir. Diskin yalnızca tam bloklar halinde yazdığını söylemiştik; bir kayıt birden fazla bloğa yayılıyorsa ve elektrik tam ikinci blok yazılmadan kesilirse, ortaya **yarım kalmış yazma** (torn write) çıkar: kaydın ilk yarısı yeni, ikinci yarısı eski veriden oluşan, hiçbir zaman var olmamış, anlamsız bir melez. Düz bir dosyayla çalışan bir program, yeniden açıldığında bu melezi sağlam bir kayıt sanır ve onun üzerinden işlem yapmaya kalkar. Daha sinsi bir kip, işletim sisteminin yazmaları **yeniden sıralamasıdır**: programınız önce veriyi, sonra "veri geçerli" işaretini yazsa bile, sistem bunları diske ters sırada işleyebilir; o aralıkta bir çökme, geçerli işareti taşıdığı halde içeriği yazılmamış bir kayıt bırakır. Yeniden açıldığında veritabanı, kendisini tutarlı bir duruma getirmeyi bilmek zorundadır. Bunu, yazma-öncesi günlük (write-ahead log) ve bütünlük sağlamaları (checksum) gibi tekniklerle yapar; altıncı bölümün konusu budur.

Soyutlama sorunu. Son olarak, düz bir dosyayla çalışırken verinin diskte tam olarak nasıl yerleştiğini — hangi baytın nerede durduğunu, kayıtların nasıl ayrıldığını — bizzat düşünmek zorundasınızdır. Bir veritabanı bu ayrıntıları gizler. Siz "şu koşula uyan kayıtları getir" dersiniz; verinin fiziksel yerleşimiyle uğraşmazsınız. Bu soyutlama, hem işinizi kolaylaştırır hem de veritabanının altta yerleşimi değiştirip eniyileme yapmasına olanak tanır.

İşte bir veritabanını basit bir dosyadan ayıran şey budur: bu beş sorunu — arama, eşzamanlılık, bütünlük, kurtarma ve soyutlama — sizin yerinize, sistemli ve güvenilir biçimde çözmesidir.

Bu sorunların birbiriyle örülü olduğunu görmek önemlidir; tek tek çözülemezler. Çökmeden kurtulma, yazmaların hangi sırayla diske indiğine bağlıdır; eşzamanlılık denetimi, iki yazmanın birbirinin günlüğünü bozmamasını gerektirir; bütünlük güvencesi, hem eşzamanlı erişime hem de çökmeye karşı aynı anda ayakta kalmak zorundadır. Düz dosyayla çalışan bir geliştirici, bu sorunların her birini ayrı ayrı çözmeye kalktığında, çözümlerin birbirini bozduğunu keşfeder; çünkü doğru çözüm, hepsini birlikte ele alan bütünleşik bir tasarımdır. Bir VTYS'nin asıl değeri de buradadır: bu beş sorunu birbirini gözeterek, tek bir tutarlı mimari içinde çözer.

1.4 Veritabanı ile veritabanı yönetim sistemi

Günlük konuşmada "veritabanı" sözcüğünü iki ayrı şey için kullanırız ve bu iki anlamı ayırmak yararlıdır. Dar anlamıyla **veritabanı**, düzenlenmiş verinin kendisidir — diskte duran o kayıt yığını. **Veritabanı yönetim sistemi** (kısaca VTYS) ise o veriyi yöneten yazılımdır: yazma ve okuma isteklerini karşılayan, indeksleri tutan, eşzamanlılığı düzenleyen, çökmeden kurtaran program. Bu kitapta asıl ilgilendiğimiz şey, ikincisidir: veriyi yöneten makinenin içindeki çarklardır. "OxiDB" dediğimizde de aslında bu makineden, yani VTYS'den söz ederiz.

Bir VTYS'yi, bir kütüphanenin işleyişine benzetebiliriz. Kitaplar (veri) raflarda (disk) durur. Ama bir kütüphaneyi yalnızca raflar değil, işleyiş kurar: kitapların hangi düzene göre dizildiği (veri modeli), aradığınız kitabı fişlerden bulmanızı sağlayan katalog (indeks), aynı kitabı iki kişiye birden vermemeyi sağlayan ödünç defteri (eşzamanlılık denetimi), yangın çıktığında en değerli ciltleri koruyan yedekleme düzeni (dayanıklılık ve kurtarma). VTYS, tüm bu işleyişi yürüten görünmez kütüphanecidir.

1.4.1 Soyutlama katmanları: mantıksal, fiziksel ve aradaki perde

Bir VTYS'nin en kalıcı katkılarından biri, veriyi birden çok **soyutlama katmanında** (abstraction layer) sunmasıdır. Bu katmanlı düşünce, ilişkisel modeli ortaya koyan çalışmada açıkça dile getirilmiş ve sonraki tüm veritabanı tasarımına sinmiştir.[1] En üstte **mantıksal katman** durur: verinin sizin gözünüzde aldığı biçim — tablolar, belgeler, alanlar. Bu, "veri neye benziyor" sorusunun yanıtıdır ve bir sonraki bölümün konusu olan veri modelidir. En altta **fiziksel katman** vardır: baytların diskte tam olarak nasıl yerleştiği, hangi bloğun nerede durduğu, hangi indeks yapısının kullanıldığı. Bu, "veri nasıl saklanıyor" sorusunun yanıtıdır.

Bu iki katmanı birbirinden ayırmanın gücü, **bağımsızlık** sağlamasıdır. Fiziksel katmanı — diskteki yerleşimi, kullanılan indeksleri, sıkıştırma biçimini — tümüyle değiştirebilirsiniz ve mantıksal katmanda hiçbir şey değişmez; sorgularınız aynı kalır. Veritabanı, "şu koşula uyan kayıtları getir" isteğini, altta veriyi nasıl tuttuğundan bağımsız olarak yanıtlar. Bu ayrım, ikinci ve üçüncü kısımda OxiDB'nin aynı mantıksal belge modelini hem bellek ağırlıklı hem disk öncelikli bir fiziksel yerleşimle nasıl sunabildiğini gördüğümüzde somutlaşacak: dışarıdan bakan için belge aynıdır, içeride yerleşim bambaşkadır.

Bu kitabın yapısı da bu katmanlara göre kurulmuştur. Kısım I, mantıksal katmanda kalır: veriyi nasıl düşündüğümüzü konuşur. Kısım II, perdeyi aralayıp fiziksel katmana iner: belgenin diske nasıl yazıldığını, indekslendiğini, çökmeden nasıl korunduğunu anlatır. Soyutlama, ikisi arasındaki perdedir; ve bir VTYS, o perdeyi sizin yerinize gergin tutan yazılımdır.

1.5 Gömülü kütüphane mi, sunucu mu

Bir VTYS'nin uygulamanızla nasıl konuştuğu, baştan verilen önemli bir tasarım kararıdır ve iki uçtan biriyle başlar. Birinci uçta **gömülü** veritabanı vardır: VTYS, uygulamanızın içine bir kütüphane olarak yerleşir, onunla aynı süreçte çalışır. Veriye erişim, bir ağ üzerinden değil, doğrudan fonksiyon çağrılarıyla olur; bu çok hızlıdır ve hiçbir sunucu kurulumu gerektirmez, ama veritabanı yalnızca o tek uygulamaya hizmet eder. İkinci uçta **sunucu** veritabanı vardır: VTYS ayrı bir süreç, hatta ayrı bir makine olarak çalışır; uygulamalar ona ağ üzerinden bağlanır. Bu, aynı veriye birçok uygulamanın aynı anda erişmesini sağlar, ama her isteğin ağdan geçmesinin bir bedeli ve sunucuyu yönetmenin bir yükü vardır.

Bu kitapta ele alacağımız OxiDB, ilginç biçimde her iki uçta da çalışabilen bir tasarıma sahiptir; üçüncü kısımda bunun nasıl mümkün olduğunu göreceğiz. Şimdilik yalnızca şunu bilmek yeterli: aynı temel mekanizmalar — depolama, indeks, işlem, kurtarma — bir VTYS gömülü de çalışsa, ağ üzerinden de hizmet verse, özünde aynıdır. Bu kitabın ikinci kısmında anlatacağımız ilkeler, bu yüzden ikisine de aynı ölçüde uygulanır.

1.6 Bu kitabın kuracağı zihinsel harita

Buraya kadar, bir veritabanının çözmek zorunda olduğu sorunları sıraladık. Kitabın geri kalanı, bu sorunların her birinin nasıl çözüldüğünü tek tek açar. Yol haritamızı şöyle özetleyebiliriz. Önce verinin nasıl **modellendiğini** göreceğiz: ham baytları anlamlı yapılara dönüştürmenin tarih boyunca denenmiş yolları ve belge modelinin bu yolların neresinde durduğu. Ardından bir belge veritabanının iç **mimarisine** ineceğiz: verinin diske nasıl **yazıldığı** (depolama motoru), bu yazmanın çökmeye karşı nasıl **güvence altına alındığı** (dayanıklılık), aradığımızı taramadan nasıl **bulduğumuz** (indeksleme), sorularımızın nasıl **yanıtlandığı** (sorgu ve toplama), verinin nasıl **tutarlı** kaldığı (işlemler ve eşzamanlılık) ve sistemin tek makineyi aşınca nasıl **büyüdüğü** (replikasyon ve sharding). Son olarak, tüm bu parçaların gerçek bir motorda, OxiDB'de nasıl bir araya geldiğini somut olarak izleyeceğiz.

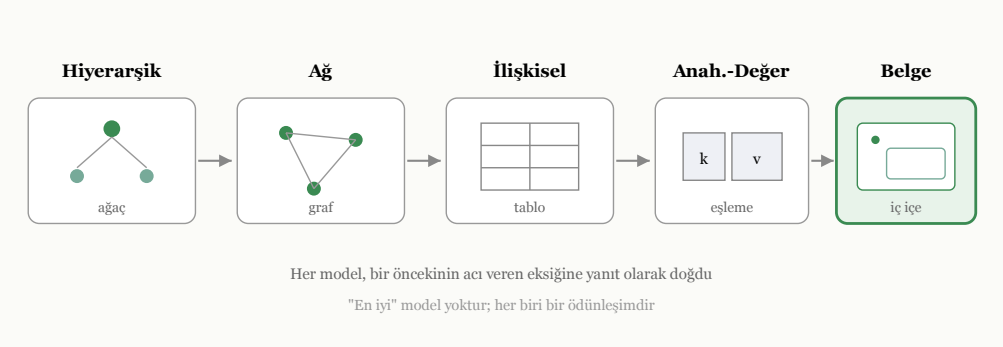
Bir sonraki bölümde ilk büyük soruyla, veri modeli sorusuyla başlıyoruz: aynı veriyi düzenlemenin birçok yolu vardır ve bu yolların her birinin kendine özgü güçleri ve zayıflıkları olmuştur. Belge modelinin neden ortaya çıktığını anlamak için, önce ondan önce gelenleri — ve onların hangi sıkıntılarına yanıt olarak doğduğunu — görmemiz gerekiyor.

[1] E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM* 13(6), 1970.

Bölüm 2

Veri Modelleri: Hiyerarşikten Belgeye

Bir önceki bölümde, bir veritabanının değerinin yalnızca sakladığı baytlardan değil, o baytları yapılandırma biçiminden geldiğini söylemiştik. Verinin nasıl yapılandırıldığına **veri modeli** denir ve bu, bir veritabanı sisteminin kimliğini belirleyen en temel karardır. Bu bölümde, veri modellerinin tarih boyunca nasıl evrildiğini izleyeceğiz. Bu bir tarih dersi değil; her modelin, kendinden öncekinin acı veren bir eksiğine yanıt olarak doğduğunu görmek, belge modelinin neden ortaya çıktığını ve hangi sorunları çözmeyi vaat ettiğini anlamanın en sağlam yoludur.



Şekil 2.1: Veri modellerinin evrimi; her biri bir öncekinin eksiğine yanıt olarak doğdu.

2.1 Veri modeli neyi belirler

Bir veri modeli üç soruyu birden yanıtlar. Birincisi: veriyi hangi **biçimde** tutarız — düz bir liste mi, bir tablo mu, iç içe geçmiş bir yapı mı? İkincisi: veri parçaları arasındaki **ilişkileri** nasıl ifade ederiz — bir siparişin bir müşteriye ait olduğunu, bir yazının birçok etikete sahip olduğunu nasıl söyleriz? Üçüncüsü: bu veriye hangi **işlemlerle** erişiriz — onu nasıl arar, filtreler, birleştirir, değiştiririz?

Bu üç sorunun yanıtları birbirine sıkıca bağlıdır. Veriyi tutma biçiminiz, ilişkileri ifade etme imkânlarınızı belirler; ilişkileri ifade etme biçiminiz de hangi soruları kolayca, hangilerini zorlukla sorabileceğinizi belirler. Tarih boyunca her veri modeli, bu üç soruya farklı bir yanıt vermiş; her yanıt bazı işleri kolaylaştırırken bazılarını zorlaştırmıştır. Hiçbir model her açıdan üstün değildir; her biri belirli bir ödünleşimi temsil eder. Yolculuğumuzun özünde bu ödünleşimi izlemek yatıyor.

2.2 Hiyerarşik model: dünya bir ağaçtır

Bilgisayarların iş dünyasına girdiği ilk dönemde, en doğal yapı ağaç biçiminde görünüyordu. Birçok gerçek dünya ilişkisi gerçekten de hiyerarşiktir: bir şirketin departmanları, departmanların çalışanları vardır; bir siparişin kalemleri vardır. **Hiyerarşik model**, veriyi tam da böyle, bir ağaç olarak düzenler. Tepede bir kök kayıt durur; onun altında çocuk kayıtlar, onların altında torunlar uzanır. Her kaydın tek bir ebeveyni vardır ve veriye erişmek, bu ağaçta kökten yapraklara doğru ilerlemek demektir.

Bu model, hiyerarşik olan veriyi temsil etmekte zarif ve hızlıdır. Ama gerçek dünya her zaman temiz bir ağaç değildir; model işte tam da burada zorlanır. İki tür sıkıntı ortaya çıkar. Birincisi **çoktan-çoğa ilişkilerdir**. Bir öğrencinin birçok dersi, bir dersin birçok öğrencisi vardır. Bunu bir ağaçla temsil etmek için ya öğrencileri derslerin altına ya da dersleri öğrencilerin altına yerleştirmek zorunda kalırsınız; hangisini seçerseniz seçin, diğer taraftaki veriyi **çoğaltarak** tekrar yazmanız gerekir. Bu çoğaltma hem yer israfıdır hem de tehlikelidir: aynı bilginin iki kopyası zamanla birbirinden ayrı düşer, biri güncellenir diğeri unutulur ve veri tutarsız hale gelir.

İkinci sıkıntı **gezinmenin katılığıdır**. Hiyerarşik bir veritabanında bir kayda ulaşmak için izleyeceğiniz yol baştan bellidir: ağacın yapısının dikte ettiği güzergâhtan gitmek zorundasınız. Eğer veriye, tasarımcının öngörmediği bir açıdan — diyelim ki "şu dersi alan tüm öğrenciler" yerine "şu öğrencinin aldığı tüm dersler" — bakmak isterseniz, ya çok dolambaçlı bir yol izlersiniz ya da hiç yapamazsınız. Sorgu, verinin fiziksel yapısına tutsaktır.

Hiyerarşik modelin en bilinen somut örneği, IBM'in 1960'ların sonunda Apollo uzay programı için geliştirdiği ve onlarca yıl boyunca büyük kuruluşların belkemiği olan IMS (Information Management System) adlı sistemdir. IMS'te veri, **bölümlenmiş ağaçlar** (segment) halinde tutulur ve bir kayda erişmek için ağacın yukarıdan aşağıya **önsıra** (preorder) gezinme düzeninde ilerlenir; yani fiziksel disk yerleşimi, ağacın gezinme sırasını birebir izler. Bu, ağacın "doğru" ucundan girip aradığı yaprağa inen bir sorgu için son derece hızlıdır — erişim, ağaç derinliği kadar adımda, yani veri büyüklüğünün logaritması mertebesinde tamamlanır. Ama yanlış uçtan, örneğin bir yaprağın tüm "kardeş" ebeveynlerini bulmak isteyen bir sorgu için, tüm ağacı baştan sona taramaktan başka çare yoktur. Maliyet, sorgunun ağacın doğal yönüyle ne kadar uyuştuğuna bağlıdır; bu da modelin katılığının doğrudan bir sonucudur.

Hiyerarşik modelin bize öğrettiği ders şudur: iç içe geçmiş, ağaç biçimli veriyi bir arada tutmak güçlü ve doğaldır — bu fikri belge modelinde yeniden göreceğiz — ama tek bir katı hiyerarşiye mahkûm olmak ve gezinme yolunu yapıya bağlamak, esnekliği öldürür.

2.3 Ağ modeli: bağlantılar her yöne

Hiyerarşik modelin çoktan-çoğa ilişkilerdeki sıkıntısına bir yanıt olarak **ağ modeli** doğdu. Bu modelin kuramsal çerçevesini ve adlandırmasını, 1960'ların sonunda CODASYL adlı komitenin standartlaştırdığı veri tabanı görev grubu ortaya koydu; modelin fikir babası sayılan Charles Bachman, bu katkısıyla 1973 Turing Ödülü'nü alacak ve ders konuşmasında veritabanını "programcının gezindiği bir uzay" olarak tarif edecekti.[1] Bu "gezgin programcı" eğretilemesi, ağ modelinin hem gücünü hem de lanetini özetler. Buradaki fikir, kayıtları katı bir ağaca hapsetmek yerine, aralarına istenen her yöne **işaretçiler** koymaktır. Bir öğrenci kaydı, aldığı her derse bir bağlantı taşır; bir ders kaydı, kendisini alan her öğrenciye. Bağlantılar, "set" denilen, bir sahip-kayıttan birçok üye-kayda uzanan adlandırılmış zincirler halinde düzenlenir. Böylece çoktan-çoğa ilişkiler veriyi çoğaltmadan ifade edilebilir.

Bu, ifade gücü açısından bir ilerlemeydi, ama ağır bir bedelle geldi: **karmaşıklık.** Veriye erişmek, artık zihinde tuttuğunuz bir bağlantı labirentinde tek tek işaretçileri izlemek demekti. Programcı, aradığı veriye ulaşmak için "şu kayıttan şu bağlantıyı izle, oradan şuna geç" diye, adım adım bir gezinme yolu yazmak zorundaydı; üstelik bu gezinme, bir imleç gibi davranan "geçerli kayıt" işaretçisini elle ileri geri taşıyarak yürütülürdü. Bir işaretçiyi izlemek tek başına ucuzdur — diskte bir konuma doğrudan atlamaktır — ama

[1] Charles W. Bachman, "The Programmer as Navigator," *Communications of the ACM* 16(11), 1973.

bir sorgu yüzlerce kaydı dolaşmayı gerektirdiğinde, bu yüzlerce **rastgele** disk erişimine dönüşür; ve birinci bölümde gördüğümüz gibi, rastgele erişim diskin en sevmediği şeydir. Maliyet, gezilen bağlantı sayısına doğrudan bağlıdır ve sorguyu yazan programcının seçtiği yola tabidir; sistemin daha akıllı bir yol bulma imkânı yoktur, çünkü yolu seçen programcının kendisidir.

Daha derin sorun ise bakımdı. Veri yapısı değiştiğinde — yeni bir bağlantı türü eklendiğinde ya da bir set'in yönü değiştiğinde — bu gezinme kodunun büyük bölümü kırılırdı. Veriye nasıl ulaşılacağı bilgisi, uygulamanın içine derinlemesine gömülmüştü; mantıksal yapı ile fiziksel yapı birbirine kenetlenmişti.

Hem hiyerarşik hem de ağ modelinin ortak kusuru buydu: ikisinde de **veriye erişim yolu, verinin yapısına sıkıca bağlıydı**. Ne sorabileceğiniz ve nasıl sorabileceğiniz, tasarımcının baştan kurduğu yapının elverdiğiyle sınırlıydı. Bu kenetlenmeyi kırmak, bir sonraki büyük fikrin işi olacaktı.

2.4 İlişkisel model: büyük ayrışma

1970'te, bu kenetlenmeden rahatsız olan bir araştırmacı — E. F. Codd — radikal bir öneri getirdi.[2] Verinin fiziksel olarak nasıl saklandığı ile mantıksal olarak nasıl düşünüldüğünü birbirinden tamamen **ayırmayı** önerdi. Bu fikir, **ilişkisel model** olarak bilinir ve onlarca yıl boyunca veritabanı dünyasına egemen oldu.

İlişkisel modelin temel yapısı şaşırtıcı derecede sadedir: her şey bir **tablodur**. Bir tablo, sütunları (alanlar) ve satırları (kayıtlar) olan bir ızgaradır; tıpkı bir hesap çizelgesi gibi. Müşteriler bir tabloda, siparişler başka bir tabloda durur. Peki bir siparişin hangi müşteriye ait olduğunu nasıl söyleriz? İşaretçiyle değil — bir **değerle**. Sipariş satırında, ait olduğu müşterinin kimliğini taşıyan bir alan bulunur. İlişki, fiziksel bir bağlantı değil, paylaşılan bir değerdir. Bu inceliğin sonuçları derindir: ilişkiler veriye gömülü işaretçilerden kurtulduğu için, veriyi istediğiniz gibi yeniden düzenleyebilir, yeni ilişkiler keşfedebilirsiniz.

Bu ayrışmanın iki büyük armağanı oldu. Birincisi, sorguların **bildirimsel** hale gelmesidir. Artık veriye nasıl ulaşılacağını adım adım tarif etmek yerine, yalnızca **ne** istediğinizi söylersiniz: "şu koşullara uyan müşterilerle şu koşullara uyan siparişleri eşleştir." Verinin fiziksel olarak nerede durduğunu, hangi indekslerin kullanılacağını sistem kendisi karar verir. Programcı "ne", sistem "nasıl" sorusuyla ilgilenir. Bu ayrım, sorgu eniyileyici denen ve verinin en hızlı nasıl getirileceğini hesaplayan bileşeni doğurdu; sekizinci bölümde bu fikre döneceğiz.

İkinci armağan **normalleştirmedir**. İlişkisel model, her bilginin tek bir yerde durmasını teşvik eder. Bir müşterinin adresi yalnızca müşteri tablosunda yazar; o müşterinin yüzlerce siparişi olsa da, adres her siparişin içinde tekrarlanmaz, yalnızca müşteri kimliğiyle ona atıfta bulunulur. Böylece adres değiştiğinde tek bir satırı güncellemek yeterli olur; çoğaltmadan kaynaklanan tutarsızlık tehlikesi ortadan kalkar. Hiyerarşik modelin en büyük derdine — veri çoğaltma — temiz bir yanıttı bu.

İlişkisel model, sağlam matematiksel temeli, bildirimsel sorgu gücü ve veri bütünlüğüne verdiği önemle haklı bir egemenlik kurdu ve hâlâ dünyanın verisinin büyük kısmını taşıyor. Ama her güçlü fikir gibi, onun da bir gölgesi vardı.

2.5 İlişkisel modelin gölgesi: parçalanma ve uyumsuzluk

İlişkisel modelin verdiği şey — her bilgiyi ayrı, düzgün tablolara bölmek — aynı zamanda onun zorlandığı yeri yaratır. Çünkü uygulamalarda düşündüğümüz "şeyler" çoğu zaman tek bir tabloya sığmaz; birçok tabloya **parçalanmıştır**.

Bir örnek üzerinden düşünelim. Bir e-ticaret siparişi, kavramsal olarak tek bir bütündür: bir müşterisi, bir teslimat adresi, birçok kalemi, her kalemin bir ürünü ve miktarı vardır. İlişkisel model bunu düzgünce ayrı tablolara böler: siparişler, sipariş kalemleri, ürünler, adresler. Bu, depolama açısından temizdir. Ama o siparişi ekranda **bütün olarak** görmek istediğinizde, bu parçaları yeniden bir araya getirmeniz gerekir. Dağılmış

[2] E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM* 13(6), 1970.

satırları paylaşılan değerler üzerinden eşleştirip birleştiren bu işleme **birleştirme** (join) denir. Birleştirme güçlü bir araçtır, ama bedeli vardır: ne kadar çok tabloyu birleştirirseniz, sorgu o kadar karmaşıklaşır ve çoğu zaman o kadar yavaşlar. Bir siparişi göstermek için yarım düzine tabloyu birleştirmek, sık karşılaşılan bir yüktür.

İkinci ve daha sinsi bir sürtünme, programlama dünyasıyla veritabanı dünyası arasındaki uyumsuzluktan doğar. Modern programlarda veriyi, iç içe geçmiş nesneler olarak düşünürüz: bir "sipariş" nesnesinin içinde bir "müşteri" nesnesi, bir "kalemler" listesi durur. Oysa ilişkisel veritabanında bu, düz tablolara serilmiştir. Programdaki zengin, iç içe nesne ile veritabanındaki düz satırlar arasında sürekli bir çeviri yapmak gerekir: nesneyi parçalayıp satırlara dağıtmak, sonra satırları toplayıp nesneyi yeniden kurmak. Bu sürekli çeviri yüküne **nesne-ilişkisel uyumsuzluk** denir. Yıllarca, bu çeviriyi otomatikleştirmek için koca yazılım katmanları geliştirildi; ama uyumsuzluğun kökü, iki dünyanın veriyi farklı biçimlerde düşünmesinde yatıyordu.

İşte belge modelini doğuran soru tam burada belirir: *Madem uygulamalar veriyi iç içe geçmiş bütünler olarak düşünüyor, veritabanı da onu neden öyle saklamasın?*

2.6 Anahtar-değer modeli: en yalın biçim

Belge modeline geçmeden önce, onun en yakın akrabasına, en yalın veri modeline bakalım. **Anahtar-değer modeli**, bir veritabanını dev bir sözlük gibi düşünür: her verinin bir **anahtarı** vardır ve o anahtarla, ona bağlı **değeri** geri alırsınız. Tıpkı bir vestiyer fişi gibi — fişi verirsin, paltonu alırsın.

Bu modelin gücü sadeliğindedir. Anahtarla erişim son derece hızlıdır: çoğu anahtar-değer sistemi, anahtarı bir özet (hash) işlevinden geçirip değerin yerine neredeyse doğrudan ulaşır, yani erişim, veri ne kadar büyürse büyüsün sabit zamanda — büyüklükten bağımsız olarak — tamamlanır. Bu sabit-zamanlı erişim, modelin damgasıdır. Aynı sadelik, dağıtmayı da kolaylaştırır: anahtarın özetine bakarak bir kaydın hangi makineye düşeceğine tek başına karar verilebilir, başka hiçbir kayda danışmaya gerek yoktur; on ikinci bölümde göreceğimiz parçalama (sharding) fikrinin en temiz hali budur.

Ama bunun bir kısıtı vardır: sistem, değerin **içine bakmaz**. Değer, sistem açısından anlamsız bir bayt yığınıdır. Bu yüzden "değeri şu koşula uyan kayıtları getir" diyemezsiniz; yalnızca anahtarını bildiğiniz kaydı alabilirsiniz. Değerin içeriğine göre arama, filtreleme ya da gruplama yapmak mümkün değildir, çünkü sistem o içeriğin yapısından habersizdir. İçeriğe göre arama tek seçenek olduğunda, geriye tüm kayıtları baştan sona taramak kalır — yani veri büyüklüğüyle doğru orantılı, doğrusal bir maliyet. Anahtar-değer modeli, hızı bu takastan kazanır: içeriği görmezden gelmeyi kabul ettiği için, gördüğü tek şey olan anahtarda kusursuz olur.

Anahtar-değer modeli, "anahtarıyla tek kayıt al" ihtiyacının baskın olduğu durumlarda kusursuzdur. Ama çoğu uygulama, verinin içeriğine göre de soru sormak ister. Belge modeli, tam da bu noktada devreye girer: anahtar-değer modelinin sadeliğini korur, ama değeri sisteme **anlaşılır** kılar.

2.7 Belge modeli: yapılandırılmış bütünler

Belge modeli, anahtar-değer modelinin değerini opak bir bayt yığını olmaktan çıkarıp, sistemin anlayabildiği, **yapılandırılmış** bir nesne haline getirir. Bu nesneye **belge** denir. Bir belge, alanlardan oluşur; her alanın bir adı ve bir değeri vardır. Değer basit olabilir — bir sayı, bir metin — ya da **kendisi de iç içe geçmiş** bir belge ya da bir liste olabilir. Yani belgeler ağaç gibi derinleşebilir: bir siparişin içinde müşteri bilgisi, onun içinde adres, sipariş kalemlerinin oluşturduğu bir liste, her kalemin içinde ürün ayrıntıları durabilir — hepsi tek bir belgede, tek bir bütün olarak.

Bu, ilişkisel modelin parçalama derdine doğrudan bir yanıttır. Uygulamanın "bir sipariş" diye düşündüğü şey, veritabanında da tek bir belge olarak durur. Onu okumak için yarım düzine tabloyu birleştirmeniz gerekmez; belgeyi olduğu gibi alırsınız ve uygulamanızdaki nesneye neredeyse birebir karşılık gelir. Nesne-ilişkisel

uyumsuzluk büyük ölçüde erir, çünkü iki dünya da veriyi artık aynı biçimde — iç içe geçmiş bütünler olarak — düşünmektedir.

Belge modelinin ikinci ayırt edici özelliği, hiyerarşik modelin tersine, **katı bir şema dayatmamasıdır**. İlişkisel bir tabloda her satır aynı sütunlara sahip olmak zorundadır; sütunlar baştan tanımlanır ve değiştirmek zahmetlidir. Belge modelinde ise her belge kendi alanlarını taşıyabilir. Aynı koleksiyondaki iki belgenin alanları farklı olabilir; bir belgeye yeni bir alan eklemek için önceden bir tanım değiştirmeniz gerekmez. Bu esneklik, verinin biçiminin zamanla geliştiği, her kaydın birbirinin aynı olmadığı durumlarda büyük rahatlık sağlar. Dördüncü bölümde bu esnekliğin hem armağanlarını hem de tuzaklarını ayrıntısıyla ele alacağız; çünkü "şema yok" demek, "şema düşünmek gerekmiyor" demek değildir.

Belge modelinin ödünleşimi de tam buradan doğar. İlişkisel modelin her bilgiyi tek bir yerde tutma disiplinini gevşetir; bir bilgiyi, ait olduğu belgenin içine gömerek tekrarlamayı kabul eder. Bu, okumayı hızlandırır — her şey tek yerde — ama hiyerarşik modelin eski derdini, çoğaltmayı geri çağırır. Belge modelinde verinizi tasarlarken sürekli bir soruyla yüzleşirsiniz: bu bilgiyi belgenin içine **gömmeli** miyim, yoksa ayrı tutup ona **atıfta** mı bulunmalıyım? Bu soru, belge modeliyle çalışmanın kalbinde durur ve dördüncü bölümü tümüyle ona ayıracağız.

2.8 Diğer modeller ve manzaranın bütünü

Tamlık için, manzaranın birkaç parçasından daha söz edelim. **Nesne-yönelimli** (object-oriented) model, 1980'lerde, programlama dillerindeki nesneleri hiçbir çeviri yapmadan, olduğu gibi diske kalıcı kılma vaadiyle ortaya çıktı; iç içe nesneleri, hatta nesneler arası işaretçileri doğrudan saklardı. Az önce değindiğimiz nesne-ilişkisel uyumsuzluğa en doğrudan saldırı buydu, ama işaretçiye dayalı yapısı onu ağ modelinin bazı katılıklarına geri sürükledi ve geniş çapta tutunamadı; yine de bıraktığı fikir — veriyi uygulamanın düşündüğü biçimde saklamak — belge modelinde yeniden hayat bulacaktı.

Graf modeli, ilişkilerin kendisini birinci sınıf bir varlık haline getirir; verinin değil, veriler arasındaki **bağlantıların** ön planda olduğu durumlar için — sosyal ağlar, öneri sistemleri, yol haritaları gibi — biçilmiş kaftandır. Bir bakıma ağ modelinin, bildirimsel sorgu gücüyle yeniden doğmuş, olgun halidir. Ayırt edici gücü şudur: "şu kişinin arkadaşlarının arkadaşları" gibi, ilişki zincirlerinde derinlemesine yürüyen sorgular, graf modelinde her adımda yalnızca komşu düğümlere bakarak ucuzca ilerler. Aynı sorgu, ilişkisel modelde her derinlik düzeyi için yeni bir birleştirme gerektirir ve maliyet zincir uzadıkça katlanarak artar; çünkü ilişkisel model, ilişkiyi değerle ifade ettiği için her sıçramada o değeri yeniden eşleştirmek zorundadır. Graf modeli ilişkiyi fiziksel bir bağ olarak tuttuğundan, sıçrama doğrudan bir işaretçi izlemeye indirgenir.

Geniş-sütun (wide-column) modeli ise, çok büyük ölçekli ve yazma-yoğun yükler için, tabloları satır yerine sütun grupları halinde düzenleyerek farklı bir ödünleşim sunar. Veriyi satır satır değil, sütun sütun bir arada tutmanın somut kazancı, yalnızca birkaç sütuna bakan toplama (aggregation) sorgularında ortaya çıkar: bir milyon kaydın yalnızca "tutar" sütununu toplamak isterken, satır temelli bir yerleşim her kaydın tüm sütunlarını diskten okumaya zorlar; sütun temelli yerleşimde ise yalnızca tutar sütunu, diskte yan yana, ardışık okunur. Birinci bölümdeki sıralı erişim kuralı sayesinde bu, kat kat ucuzdur. Karşılığında, tek bir kaydın tüm alanlarını birlikte okumak — satır temelli yerleşimin kolayca yaptığı şey — geniş-sütun modelinde dağınık sütunları toplamayı gerektirir.

Bütün bu modeller arasında "en iyi" yoktur; yalnızca **farklı ödünleşimler** vardır. Hiyerarşik ve ağ modelleri, veriye erişimi yapıya kenetleyerek esneklik ödedi. İlişkisel model, mantığı fizikten ayırarak büyük bir esneklik ve sorgu gücü kazandı, ama veriyi parçalayarak birleştirme yükünü ve nesne uyumsuzluğunu getirdi. Anahtar-değer modeli, sadelik ve ölçek uğruna içeriğe-göre-sorgudan vazgeçti. Belge modeli, ilişkisel modelin sorgu gücünü büyük ölçüde korurken, veriyi uygulamaların düşündüğü gibi iç içe bütünler halinde tutarak parçalama ve uyumsuzluk dertlerini hafifletti — bunun karşılığında ise çoğaltma ve şema disiplini sorumluluğunu geliştiricinin omuzlarına yükledi.

Bu kitabın geri kalanı belge modeline odaklanıyor; ama bu odağın bir bedel değil, bilinçli bir ödünleşim olduğunu görmek önemlidir. Belge modeli bir gümüş kurşun değildir; belirli bir denge noktasıdır ve o denge noktasının nerede işe yarayıp nerede yetersiz kaldığını bilmek, onu iyi kullanmanın ön koşuludur.

Bir sonraki bölümde, manzaranın en çok karşılaşılan geçişine yakından bakacağız: ilişkisel düşünceden belge düşüncesine geçiş. Bu geçişin neden, ne zaman ve hangi bedellerle yapıldığını anlamak — ve ne zaman *yapılmaması* gerektiğini görmek — belge veritabanlarını gerçekten kavramanın anahtarıdır.

Bölüm 3

İlişkisel Modelden Belge Modeline: Neden ve Ne Zaman

Önceki bölümde veri modellerinin manzarasını çizdik ve hiçbirinin mutlak anlamda "en iyi" olmadığını, her birinin bir ödünleşimi temsil ettiğini söyledik. Pratikte en sık karşılaşılan karar, ilişkisel düşünceden belge düşüncesine geçiş kararıdır. Bu bölüm, o kararın anatomisini açar: ekipler neden belge modeline yönelir, bu geçiş hangi durumlarda kazandırır, hangi durumlarda kaybettirir ve karar verirken hangi soruları sormak gerekir. Amacımız bir modeli yüceltmek değil; ne zaman hangisinin doğru araç olduğunu görebilecek bir muhakeme kazandırmaktır. Çünkü yanlış araçla iyi yapılmış bir iş, yine de yanlış iştir.

Şekil 3.1: Aynı sipariş verisi: ilişkisel parçalama (join) ile belge gömme yan yana.

3.1 Kral değil hizmetkâr: modeli erişim örüntüleri belirler

Bu bölümün üzerine kurulu olduğu tek bir fikir vardır ve onu en baştan söylemek gerekir: **veri modeli, erişim örüntülerini izler.** Yani veriyi nasıl yapılandıracağınıza, soyut bir "doğruluk" ilkesine göre değil, o veriyi nasıl okuyup yazacağınıza göre karar verirsiniz. Aynı gerçek dünya bilgisi — diyelim bir müşteri ve siparişleri — bir uygulamada ilişkisel olarak, bambaşka bir uygulamada belge olarak modellenebilir ve ikisi de doğru olabilir; çünkü iki uygulama o veriye farklı sorular soruyordur.

Bu yüzden "ilişkisel mi, belge mi" sorusu, soyut olarak yanıtlanamaz. Yanıt, şu somut soruların içinde gizlidir: Veriyi hangi birimler halinde okuyorum? En sık hangi sorguları soruyorum? Bir şeyi değiştirdiğimde nereleri

güncellemem gerekiyor? Bir işlem, kaç farklı varlığa birden dokunuyor? Bu bölümün geri kalanı, bu soruların yanıtlarının modeli nasıl belirlediğini gösteriyor.

3.2 Normalleştirme: bilgiyi nereye koyacağının disiplini

İlişkisel modelle belge modeli arasındaki ödünleşimi gerçekten anlamak için, ilişkisel dünyanın "doğru tasarım" anlayışının kalbindeki kavrama — **normalleştirmeye** (normalization) — biraz daha yakından bakmak gerekir. Bu kavramı ortaya koyan da, modelin kendisini öneren araştırmacıydı; bir dizi **normal form** (normal form) tanımlayarak, bir tablonun ne zaman "iyi biçimlenmiş" sayılacağını matematiksel bir kesinliğe bağladı.[1] Bu formların arkasındaki sezgiyi kavramak, belge modelinde gömme-referans kararını verirken neyi tarttığınızı görmenizi sağlar; çünkü gömmek, çoğu zaman bu disiplini bilinçli olarak gevşetmektir.

Birinci normal form (1NF), en temel kuralı koyar: her hücre **tek bir atomik değer** taşımalı; bir hücrenin içine virgülle ayrılmış bir liste ya da iç içe bir yapı tıkıştırılmamalıdır. İlişkisel modelin düz, ızgaramsı doğası tam da buradan gelir — ve dikkat edin, bu kural belge modelinin en temel özelliğini, yani bir alanın bir liste ya da iç içe bir belge olabilmesini, doğrudan yasaklar. Belge modeli, bir anlamda, 1NF'yi bilinçli olarak terk etmektir.

İkinci ve üçüncü normal formlar (2NF, 3NF) ile onların daha sıkı kuzeni Boyce-Codd normal formu (BCNF), tek bir ortak sezgiyi farklı keskinliklerle ifade eder: **her olgu tam olarak bir kez, ait olduğu yerde yazılmalıdır.** Bir sipariş satırına müşterinin adresini de yazarsanız, o adres o müşterinin her siparişinde tekrarlanır; oysa adres, siparişin değil müşterinin bir olgusudur. Bu formlar, böyle "yanlış yere konmuş" olguları ayıklayıp her birini yalnızca kendi tablosuna yerleştirmeyi buyurur. Sonuç, az önce gördüğümüz çoğaltmasız dünyadır: adres değişince tek bir satır güncellenir, tutarsızlık doğamaz.

Normalleştirmenin armağanı budur — güncelleme tutarlılığı ve çoğaltmasızlık. Ama bedeli, bu bölümün baştan beri işaret ettiği şeydir: bilgi ne kadar çok ayrı tabloya bölünürse, onu bütün olarak geri toplamak için o kadar çok birleştirme gerekir. Belge modeli, tam da bu noktada normalleştirmeye karşı bir bahis oynar: bazı olguları bilinçli olarak çoğaltıp ait oldukları varlığın içine gömerek, okuma için ödenen birleştirme bedelini, yazma için ödenen çoğaltma bedeline takas eder. Bu yüzden belge modelini "normalleştirilmemiş" (denormalized) tasarımın doğal evi olarak düşünmek yerinde olur — yeter ki bu gevşetmenin bilinçli bir karar olduğu, bir ihmal değil, akılda tutulsun.

3.3 Belge modeline geçişin dört itici gücü

Ekipleri ilişkisel modelden belge modeline yönelten gerekçeler, genellikle dört başlık altında toplanır. Bunların her biri, önceki bölümde değindiğimiz bir ilişkisel sürtünmeye verilen bir yanıttır.

Birincisi, erişim yerelliğidir. Bir uygulamanın en sık yaptığı iş, çoğu zaman tek bir varlığı bütün olarak okumaktır: bir kullanıcının profilini, bir ürünün tüm ayrıntılarını, bir siparişin tamamını. İlişkisel modelde bu varlık birçok tabloya dağılmıştır ve onu bütün olarak getirmek birleştirme gerektirir. Belge modelinde ise o varlık tek bir belgede, yan yana durur; bir okuma işlemiyle, dağınık parçaları toplamadan elinize gelir. Veriyi bir arada tutmaya **yerellik** (locality) denir ve okuma-yoğun uygulamalarda en büyük kazançlardan biridir; çünkü tek bir yerden okumak, birçok yerden toplayıp birleştirmekten neredeyse her zaman hızlıdır.

Bu kazancın kökü, birinci bölümdeki disk kurallarına dayanır. Bir belge diske **ardışık** baytlar halinde yazıldığında, onu okumak tek bir sıralı erişimdir — diskin en sevdiği erişim biçimi. Buna karşılık bir birleştirme, doğası gereği, bir tablodaki satırı bulup, taşıdığı kimlikle başka bir tablodaki ilgili satıra **atlamayı** gerektirir; ve bu atlama, çoğu zaman diskte bambaşka bir yere yapılan rastgele bir erişimdir. Bir siparişi altı tablodan toplamak, en kötü durumda her kalem için ürün tablosuna ayrı bir rastgele sıçrama demektir. Birleştirme algoritmalarının kendisi de bedavadan gelmez: iki kümeyi eşleştirmek için ya her iki tarafı da birleştirme

[1] E. F. Codd, "Further Normalization of the Data Base Relational Model," *Data Base Systems, Courant Computer Science Symposia Series* 6, Prentice-Hall, 1972.

alanına göre sıralamak (sıralama maliyeti veri büyüklüğünün logaritmasıyla çarpımı kadar) ya da bir tarafın tamamını bellekte bir özet tablosuna kurmak (bellek maliyeti) gerekir. Yerellik bu maliyetlerin hepsini birden ortadan kaldırır: birleştirilecek bir şey olmadığında, birleştirmenin algoritmik bedeli de yoktur.

İkincisi, geliştirme hızıdır. İlişkisel bir tabloda yeni bir alan eklemek ya da yapıyı değiştirmek, önceden tanımlı bir şemayı değiştirmeyi gerektirir; bu, büyük tablolarda zahmetli ve riskli bir işlemdir. Belge modelinde her belge kendi alanlarını taşıyabildiği için, yeni bir alan eklemek çoğu zaman yalnızca onu yazmaya başlamak kadar kolaydır. Gereksinimlerin hızla değiştiği, ürünün biçiminin sürekli evrildiği erken aşama projelerinde bu esneklik, hızı belirgin biçimde artırır. (Dördüncü bölümde bu esnekliğin bedelini de göreceğiz; şimdilik yalnızca itici gücün ne olduğunu saptıyoruz.)

Üçüncüsü, nesne uyumsuzluğunun azalmasıdır. Önceki bölümde, programlardaki iç içe nesnelerle ilişkisel tablolar arasındaki sürekli çeviri yükünden söz etmiştik. Bu uyumsuzluğun kökü yüzeysel bir biçim farkı değil; iki dünyanın veriyi temelden farklı **şekillerde** düşünmesidir. Bir programlama dilindeki nesne, doğası gereği bir ağaçtır — hatta işaretçilerle bir grafiktir: içinde başka nesneler, listeler, listelerin içinde başka nesneler barındırır. İlişkisel model ise, az önce gördüğümüz birinci normal form yüzünden, düzdür: iç içelik yasaktır, her şey ızgaraya serilmek zorundadır. Bir ağacı düz bir ızgaraya sığdırmak için onu parçalara ayırmak, parçalara yapay kimlikler vermek ve ilişkileri yabancı anahtarlarla kurmak gerekir; geri okurken de bu parçaları toplayıp ağacı yeniden örmek. Bu iki yönlü çeviriyi otomatikleştirmek için geliştirilen nesne-ilişkisel eşleyici (object-relational mapper) katmanları, sorunu gizler ama yok etmez; çünkü kök, iki veri şeklinin yapısal uyuşmazlığında yatar. Belge modeli, veriyi tam da uygulamanın düşündüğü biçimde — iç içe geçmiş bütünler olarak — sakladığı için bu çeviri büyük ölçüde ortadan kalkar. Programdaki nesne ile veritabanındaki belge neredeyse aynı şekle sahip olur; ağaç, ağaç olarak saklanır ve aradaki sürtünme erir.

Dördüncüsü, dağıtıma yatkınlıktır. Veri tek bir makineye sığmaz hale geldiğinde, onu birden çok makineye bölmek gerekir; on ikinci bölümde bunu ayrıntısıyla ele alacağız. Kendi içinde bütün olan, başka kayıtlara işaretçilerle bağlı olmayan belgeler, bu bölmeye doğal olarak yatkındır: her belge bir bütün olduğu için, hangi makineye gideceğine kolayca karar verilebilir ve onu okumak için başka makinelerdeki parçaları toplamak gerekmez. İlişkisel modelin birçok tabloyu birleştiren sorguları ise, veri makinelere dağıldığında çok daha pahalı hale gelir; çünkü birleştirilecek parçalar farklı makinelerde olabilir.

3.4 Belge modelinin parladığı durumlar

Bu dört itici güç bir araya geldiğinde, belge modelinin açık ara doğru seçim olduğu bir profil belirir. Eğer veriniz büyük ölçüde **kendi içinde bütün** varlıklardan oluşuyorsa — yani bir varlığı okuduğunuzda neredeyse her zaman tamamını istiyorsanız ve onu değiştirdiğinizde değişiklik o varlığın içinde kalıyorsa — belge modeli bu veriyi olduğu gibi, doğal biçimde tutar. Bir blog yazısı ve onun yorumları, bir ürün ve onun teknik özellikleri, bir kullanıcı ve onun ayarları; bunların hepsi tek bir belgeye sığan, birlikte okunup birlikte yazılan bütünlerdir.

Belge modeli ayrıca verinin biçiminin **kayıttan kayda değiştiği** durumlarda parlar. Bir ürün kataloğu düşünün: bir kitabın sayfa sayısı, bir gömleğin bedeni ve rengi, bir yazılımın lisans türü vardır. İlişkisel bir tabloda bunların hepsini barındırmak ya devasa ve çoğu boş sütunlu bir tablo gerektirir ya da zahmetli dolaylı yapılar. Belge modelinde her ürün, yalnızca kendisine ait alanları taşır; biçim değişkenliği bir sorun değil, doğal bir durumdur.

Son olarak, **okuma-ağırlıklı ve yerel** iş yükleri belge modeline çok uygundur. Veri bir kez yazılıp çok kez okunuyorsa ve okumalar varlığın tamamını birden istiyorsa, veriyi bir arada tutmanın okuma kazancı, çoğaltmanın yazma maliyetine fazlasıyla baskın gelir.

3.5 Belge modelinin zorlandığı durumlar

Şimdi madalyonun öteki yüzüne, yani belge modelinin **kötü** bir seçim olduğu durumlara gelelim. Bunları görmezden gelmek, belge veritabanlarıyla ilgili en yaygın hataların kaynağıdır; çünkü model, kötü oturduğu

yerlerde sessizce değil, acı verici biçimde zorlar.

Yoğun biçimde birbirine bağlı veri. Eğer veriniz, her yönden sorgulanan çoktan-çoğa ilişkilerle örülüyse — sosyal bir ağdaki arkadaşlıklar, bir bilgi grafındaki kavram bağlantıları gibi — belge modeli zorlanır. Çünkü belge modeli bir varlığı bir bütün olarak tutmakta iyidir, ama varlıklar arasındaki zengin, çok yönlü ilişkileri ifade etmekte ilişkisel modelin (ya da graf modelinin) gerisinde kalır. Önceki bölümde gördüğümüz çoktan-çoğa derdi, belge modelinde de yeniden belirir.

Gelişigüzel, varlıklar arası sorgular. Belge modeli, veriyi belirli bir okuma örüntüsüne göre düzenlemenizi teşvik eder. Ama uygulamanız zamanla, baştan öngörülmemiş açılardan sorular sormaya başlarsa — "şu koşula uyan tüm müşterilerin, şu koşula uyan tüm siparişlerini, şu ürünle eşleştir" gibi — ilişkisel modelin bildirimsel birleştirme gücü çok değerli hale gelir. Belge modelinde bu tür varlıklar arası sorgular ya zahmetlidir ya da veriyi baştan farklı düzenlemeyi gerektirir.

Sık güncellenen, paylaşılan bilgi. Yerellik için belgeye gömdüğünüz bir bilgi, eğer birçok belgede tekrarlanıyorsa ve sık değişiyorsa, sizi belaya sokar. Bir satıcının adını her siparişin içine gömdüğünüzü düşünün; satıcı adını değiştirdiğinde, o satıcının yüzlerce, binlerce siparişindeki kopyayı tek tek güncellemeniz gerekir. İlişkisel modelin normalleştirmesi — bilgiyi tek bir yerde tutmak — tam da bu sorunu çözmek için vardı. Bilgi ne kadar paylaşılır ve ne kadar sık değişirse, onu gömmenin bedeli o kadar artar.

Çok varlığa dokunan işlemler. Bir işlemin tutarlı kalması gereken birim, tek bir belgeden büyükse, dikkatli olmak gerekir. Belge veritabanları tek bir belge üzerindeki işlemleri kolayca atomik kılar; ama bir işlem birçok ayrı belgeye, ya da dağıtık bir kurulumda birçok makinedeki belgeye birden dokunuyorsa, tutarlılık güvencesi karmaşıklaşır. (Bu konuyu onuncu ve on ikinci bölümlerde derinlemesine ele alacağız.)

Bu dört durumun ortak noktası şudur: hepsinde de veri, **tek bir bütün olarak okunup yazılan bağımsız varlıklar** profilinden uzaklaşır ve **birbirine bağlı, paylaşılan, çok yönlü sorgulanan** bir ağa dönüşür. Belge modeli ilkinde güçlüyken, ikincisinde ilişkisel model öne geçer.

3.6 Çoğaltma ödünleşimine yeniden bakış

Belge modeline geçişin kalbinde, önceki bölümde değindiğimiz bir ödünleşim yatar ve burada onu netleştirmekte yarar var: **çoğaltma**. İlişkisel model, her bilgiyi tek bir yerde tutarak güncellemeyi ucuzlatır ve tutarsızlığı önler, ama okumayı pahalılaştırır — çünkü dağınık parçaları her okumada toplamak gerekir. Belge modeli, bilgiyi ait olduğu belgeye gömerek okumayı ucuzlatır, ama çoğaltmayı kabul ettiği ölçüde güncellemeyi pahalılaştırır ve tutarsızlık riskini geri çağırır.

Bu yüzden geçiş kararı, aslında bir okuma-yazma dengesi kararıdır. Veriniz çok okunup az yazılıyorsa ve okumalar bütünleri istiyorsa, belge modelinin gömme yaklaşımı kazandırır. Veriniz sık güncelleniyor ve paylaşılan bilgiler taşıyorsa, ilişkisel modelin normalleştirmesi kazandırır. Çoğu gerçek uygulama bu iki ucun arasında bir yerdedir; bu yüzden belge modeliyle çalışırken bile, neyi gömüp neye atıfta bulunacağınıza dair sürekli bir muhakeme yürütürsünüz. Dördüncü bölüm tümüyle bu muhakemeye ayrılmıştır.

3.7 "Ya hep ya hiç" yanılgısı ve melez gerçeklik

Bu tartışmayı bir savaş gibi sunmak — "ilişkisel mi, belge mi" — gerçeği yansıtmaz. Pratikte üç önemli incelik vardır.

Birincisi, çoğu ciddi sistem **tek bir veritabanı modeline mahkûm değildir**. Bir uygulamanın kullanıcı profillerini bir belge veritabanında, finansal işlemlerini ilişkisel bir veritabanında, oturum bilgilerini bir anahtar-değer deposunda tutması olağandır. Her veriyi, erişim örüntüsüne en uygun araçta saklama yaklaşımına "çok-dilli kalıcılık" denir. Doğru soru çoğu zaman "hangi model" değil, "bu veri parçası için hangi model" sorusudur.

İkincisi, modeller arasındaki sınırlar **bulanıklaşmıştır**. Modern belge veritabanları, başlangıçta yokken, birleştirme benzeri işlemler, çok-belgeli işlemler ve güçlü toplama yetenekleri kazanmıştır; bu kitapta OxiDB üzerinde tam da bunları göreceğiz. Aynı şekilde ilişkisel veritabanları da belge benzeri, iç içe alanları doğrudan saklayıp sorgulayabilen yetenekler edinmiştir. İki dünya, birbirinin güçlü yanlarını ödünç alarak yakınlaşmıştır. Dolayısıyla "belge modeli birleştirme yapamaz" gibi mutlak ifadeler, artık olduğundan daha keskindir.

Üçüncüsü, doğru karar **zamanla değişebilir**. Bir uygulamanın erken aşamasında, gereksinimler belirsizken belge modelinin esnekliği paha biçilmez olabilir; sistem olgunlaştıkça ve erişim örüntüleri sabitlendikçe, bazı parçalar için daha katı bir yapı tercih edilebilir. Model seçimi tek seferlik, geri dönülmez bir yemin değildir; sistemin yaşamı boyunca yeniden gözden geçirilen bir mühendislik kararıdır.

3.8 Karar vermeden önce sorulacak sorular

Bu bölümü, geçiş kararını verirken kendinize soracağınız somut sorularla toplayalım. Bu sorular, soyut tercihi pratik bir muhakemeye çevirir.

Verimi hangi **birimler** halinde okuyup yazıyorum; bir varlığı çoğunlukla bütün olarak mı, yoksa parçalarına ayrı ayrı mı erişiyorum? Tutarlı kalması gereken **birim** ne kadar büyük; bir işlem tek bir varlığa mı, yoksa birçok varlığa birden mi dokunuyor? Verim ne kadar **birbirine bağlı**; ilişkiler tek yönlü ve sahiplik temelli mi, yoksa her yönden sorgulanan çok-çok ilişkiler mi? Hangi bilgiler **paylaşılıyor** ve ne sıklıkta **değişiyor**; bunları gömersem kaç yere kopyalamış olurum? Okuma mı yoksa yazma mı baskın; bu denge gelecekte nasıl değişebilir? Sorgularım baştan **belli** mi, yoksa zamanla öngörülemeyen yeni açılardan sorular bekliyor muyum?

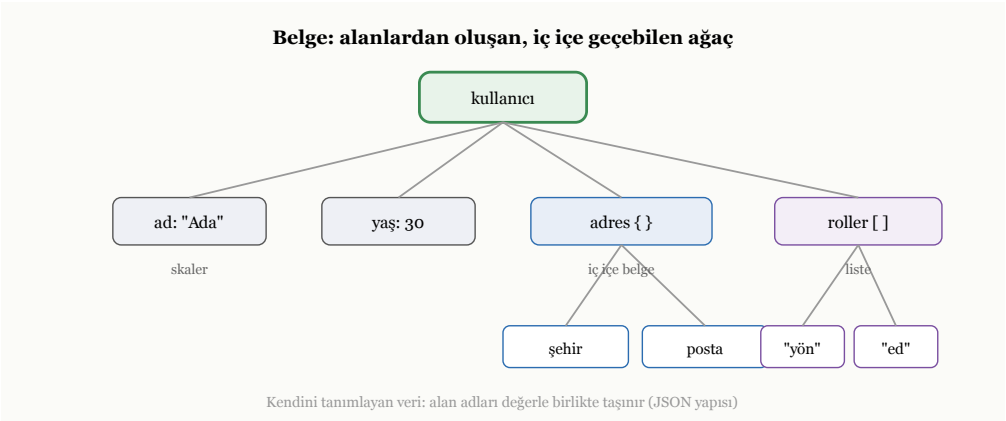
Bu soruların yanıtları, modeli sizin yerinize seçer. Yanıtlar "bağımsız bütünler, yerel erişim, az paylaşım, okuma ağırlıklı, belli sorgular" yönüne işaret ediyorsa belge modeli evinizdir. "Birbirine bağlı, paylaşılan, sık güncellenen, gelişigüzel sorgulanan veri" yönüne işaret ediyorsa ilişkisel model daha rahat edecektir. Çoğu zaman yanıt karışıktır ve bu, çok-dilli kalıcılığa ya da modellerin melezleşmiş yeteneklerine başvurmanın işaretidir.

Buraya kadar belge modelini, diğer modellerle karşılaştırarak, dışarıdan tanıdık. Artık onun **içine** girme zamanı. Bir sonraki bölümde belge modelini yakın plana alıyoruz: bir belgenin tam olarak neyden yapıldığını, JSON gibi gösterimlerin neden bu kadar yaygınlaştığını, şema esnekliğinin hem armağanını hem tuzağını ve belge modeliyle çalışmanın kalbindeki o tekrar eden kararı — gömmek mi, atıfta bulunmak mı — derinlemesine inceleyeceğiz.

Bölüm 4

Belge Modeli Derinlemesine: JSON, Şema Esnekliği, Gömme ve Referans

Önceki üç bölümde belge modelini dışarıdan, diğer modellerle karşılaştırarak tanıdık. Artık onun içine giriyoruz. Bu bölümde bir belgenin tam olarak neyden yapıldığını, JSON gibi gösterimlerin neden bu kadar yaygınlaştığını, "şemasızlık" denen şeyin gerçekte ne anlama geldiğini ve belge modeliyle çalışmanın kalbindeki o tekrar eden kararı — gömmek mi, atıfta bulunmak mı — inceleyeceğiz. Bu bölüm, Kısım I'in son durağı; onu bitirdiğimizde belgenin *ne* olduğunu tam olarak bilecek ve Kısım II'de bir veritabanının bu belgeleri içeride *nasıl* sakladığına geçmeye hazır olacağız.



Şekil 4.1: Belge, alanlardan oluşan ve iç içe geçebilen bir ağaçtır.

4.1 Bir belge neyden yapılır

Bir belge, özünde, **adlandırılmış alanların** bir koleksiyonudur. Her alanın bir adı (örneğin `ad`, `yas`, `adres`) ve bir değeri vardır. Değer birkaç temel türden biri olabilir. En basitleri **skaler** değerlerdir: bir metin, bir sayı, bir doğru/yanlış (mantıksal) değeri ya da "değer yok" anlamına gelen boş (null) değer. Ama belge modelini güçlü kılan, değerin yalnızca skaler olmak zorunda olmamasıdır.

Bir değer, bir **liste** (dizi) olabilir: sıralı bir öğeler topluluğu, örneğin bir yazının etiketleri ya da bir siparişin kalemleri. Daha da önemlisi, bir değer **başka bir belge** olabilir: alanları olan, iç içe geçmiş bir nesne. Bir kullanıcının `adres` alanı, içinde `sokak`, `sehir`, `posta_kodu` alanları olan bir alt-belge olabilir. Ve bu iç içe

geçme istediğiniz kadar derinleşebilir: listelerin içinde belgeler, belgelerin içinde listeler. İşte belge modelinin ağaç biçimli, hiyerarşik doğası buradan gelir; ikinci bölümde hiyerarşik modelin iç içe veriyi tutmaktaki gücünden söz etmiştik — belge modeli o gücü, ama tek bir katı hiyerarşiye mahkûm olmadan, devralır.

Bu yapı sayesinde, uygulamanızın zihnindeki bir "şey" — bir sipariş, bir profil, bir ürün — tek bir belgede, kendi doğal biçimiyle durabilir. Belgenin şekli, nesnenin şekline neredeyse birebir uyar. Üçüncü bölümde sözünü ettiğimiz nesne uyumsuzluğunun erimesi, tam da bu yapısal benzerlikten kaynaklanır.

4.2 Kendini tanımlayan veri

Belge modeliyle ilişkisel model arasındaki en derin yapısal fark, **şemanın nerede durduğudur**. İlişkisel bir tabloda alan adları ve türleri yalnızca bir kez, tablo tanımında yazar; satırların kendisi yalnızca değerleri taşır, hangi değerin hangi sütuna ait olduğu tablonun yapısından bilinir. Belge modelinde ise her belge, alan adlarını **kendi içinde** taşır. Yani bir belgeye baktığınızda, ayrı bir tanıma bakmaya gerek kalmadan, hangi değerin neyi ifade ettiğini görürsünüz. Buna **kendini tanımlayan** veri denir.

Bu özelliğin iki yüzü vardır. Armağan tarafı: veri taşınabilir ve esnektir. Bir belgeyi tek başına anlamlandırabilirsiniz; her belge kendi alanlarını taşıdığı için farklı belgeler farklı alanlara sahip olabilir ve yeni bir alan eklemek için merkezi bir tanımı değiştirmek gerekmez. Bedel tarafı: alan adlarının her belgede tekrarlanması yer kaplar — bir milyon belgenin her birinde `kullanici_adi` metnini taşımak, o adı bir kez tanımlamaktan daha savurgandır — ve adların tutarlılığını güvence altına alan merkezi bir otorite yoktur. Bir belgede `telefon`, başkasında `tel`, bir üçüncüsünde `telefon_no` yazarsa, sistem bunu bir hata olarak görmez; bu tutarsızlıkla baş etmek size kalır. Bu, şema esnekliği tartışmasının kalbinde yatan gerilimdir ve birazdan ona döneceğiz.

4.3 JSON nedir ve nereden geldi

Belgeyi bir bilgisayarda tutmak başka şeydir, onu insanlar ve programlar arasında **yazıya dökerek aktarmak** başka şeydir. Bir belgeyi bir dosyada saklamak, bir ağ üzerinden göndermek ya da bir kayıt üzerine yazmak için, onun metinsel bir gösterimine ihtiyaç duyarız. İşte JSON, tam da bu işi yapan bir **gösterim biçimidir**. Adı, İngilizce "JavaScript Object Notation" ifadesinin kısaltmasıdır; Türkçeye "JavaScript Nesne Gösterimi" olarak çevrilebilir. Adında JavaScript geçmesine karşın, JSON belirli bir programlama diline bağlı değildir; herhangi bir dilde okunup yazılabilen, dilden bağımsız bir veri değiş-tokuş biçimidir.

JSON'un yapısı şaşırtıcı derecede sadedir ve tam olarak bu bölümün başında anlattığımız belge yapısına karşılık gelir. Yalnızca birkaç yapı taşı vardır. Bir **nesne**, süslü parantezler içinde, ad-değer çiftlerinden oluşan bir topluluktur — yani bir belgenin ta kendisi. Bir **dizi**, köşeli parantezler içinde, sıralı bir değerler listesidir. Ve değerlerin kendisi ya bir metin (çift tırnak içinde), bir sayı, bir mantıksal değer (doğru/yanlış), boş değer (null), ya da yine bir nesne veya dizi olabilir. Değerlerin nesne ve dizi olabilmesi sayesinde JSON, tıpkı belgeler gibi, istenildiği kadar iç içe geçebilir. Yani JSON'un dilbilgisi, belge modelinin yapısının yazıya dökülmüş hâlinden başka bir şey değildir.

JSON'un dilbilgisini bu denli sağlam kılan şey, **özyinelemeli** (recursive) olarak tanımlanmış olmasıdır. Tüm dilbilgisi tek bir başlangıç kavramına, yani "değer"e dayanır; ve bir değer ya bir yaprak (metin, sayı, doğru/yanlış, null) ya da içinde yine değerler barındıran bir dal (nesne ya da dizi) olabilir. Bu özyineleme sayesinde, yalnızca bir avuç kuralla sınırsız derinlikte iç içe yapılar tanımlanabilir; dilbilgisinin tamamı tek bir sayfaya sığar, ama ürettiği yapıların çeşitliliği sonsuzdur. Sadeliğin ardındaki güç budur. Bu sadeliğin bir bedeli de vardır ve farkında olmak gerekir: JSON, kendi içinde yalnızca bu birkaç türü tanır; örneğin tam olarak hangi sayısal kesinliğin korunacağı — çok büyük tamsayıların ya da ondalıkların bit bit aynı kalıp kalmayacağı — biçimin kendisinde değil, onu okuyan tarafın yorumuna bırakılmıştır. Bu küçük boşluk, birazdan göreceğimiz ikili biçimlerin doğmasının nedenlerinden biridir.

JSON'un kökeni öğreticidir. JavaScript dili 1990'ların ortasında, web tarayıcılarında çalışmak üzere doğdu ve bu dilin, nesneleri ve listeleri kısaca yazmaya yarayan kendine özgü bir sözdizimi vardı — buna nesne

ve dizi "değişmez gösterimi" (literal) denir. 2000'lerin başında, web sayfalarının sunucularla veri alışverişi yapma ihtiyacı hızla arttı. O dönemde bu iş için yaygın olarak XML adlı, etiketlere dayalı ve oldukça ağır bir biçim kullanılıyordu. Douglas Crockford adlı bir mühendis, JavaScript'in zaten var olan o sade nesne gösteriminin, veri aktarımı için XML'den çok daha hafif ve kullanışlı bir biçim olarak kullanılabileceğini fark etti; bu fikri derleyip toparlayarak, ona bir ad (JSON) ve açık bir tanım kazandırdı ve `json.org` adresinde yayımladı.[1] Yani JSON icat edilmedi denebilir; daha çok, JavaScript'in içinde zaten saklı duran sade bir alt küme keşfedilip bağımsız bir biçim olarak öne çıkarıldı. JSON'un nesnelere bu kadar doğal uymasının nedeni de budur: o, bir programlama dilinin nesne yazma biçiminin doğrudan kendisinden türemiştir.

JSON'un yükselişi, web'in olgunlaşmasıyla el ele gitti. Tarayıcıların sayfayı baştan yüklemeden sunucuyla arka planda veri alışverişi yapabildiği tekniklerin yaygınlaşmasıyla, bu alışverişte taşınacak hafif bir biçime ihtiyaç doğdu ve JSON, ağır XML'in yerini hızla aldı. Zamanla, biçimin herkesçe aynı yorumlanması için resmî standartlara da bağlandı: hem yazılım endüstrisi standartları hem de internet standartları, JSON'un dilbilgisini kesin biçimde tanımlayan belgeler yayımladı.[2] Böylece JSON, bir kişinin derlediği pratik bir fikirden, tüm dünyanın üzerinde anlaştığı resmî bir veri değiş-tokuş biçimine dönüştü ve bugün web servislerinden yapılandırma dosyalarına, kayıt sistemlerinden — bu kitabın konusu olan — belge veritabanlarına kadar her yerde karşımıza çıkıyor.

4.4 JSON neden kazandı

JSON'un ne olduğunu ve nereden geldiğini gördükten sonra, onu belge dünyasının ortak diline dönüştüren özellikleri toparlayabiliriz; bu özellikler öğreticidir. JSON, insan tarafından okunabilir düz metindir; bir belgeye bakıp ne olduğunu hemen anlarsınız. Yapısı, tam da az önce anlattığımız belge yapısına uyar: adlandırılmış alanlar, skaler değerler, listeler ve iç içe nesneler. Ve belki en önemlisi, modern programlama dillerindeki nesnelere neredeyse doğrudan karşılık gelir; bir programdaki nesneyi JSON'a çevirmek ve geri almak kolaydır. Bu üç özellik — okunabilirlik, belge yapısına uygunluk ve nesnelere yakınlık — JSON'u belge dünyasının ortak dili yaptı.

Ama JSON'un, bir veritabanının iç gösterimi olarak doğrudan kullanmak için iki önemli eksiği vardır. Birincisi **tür yoksulluğudur**. JSON yalnızca birkaç temel tür tanır; örneğin tarih, zaman ya da ikili (binary) veri için ayrı bir türü yoktur. Bir tarihi JSON'da saklamak için onu bir metne ya da sayıya çevirmek gerekir ve bu çevrimin anlamı, yorumlayan tarafın bilgisine kalır. İkincisi **verimsizliktir**: JSON düz metindir, bu yüzden hem yer açısından savurgandır hem de bir alanı bulmak için metni baştan ayrıştırmak gerekir; sayıları bile metin olarak yazıp yeniden okumak zorundadır.

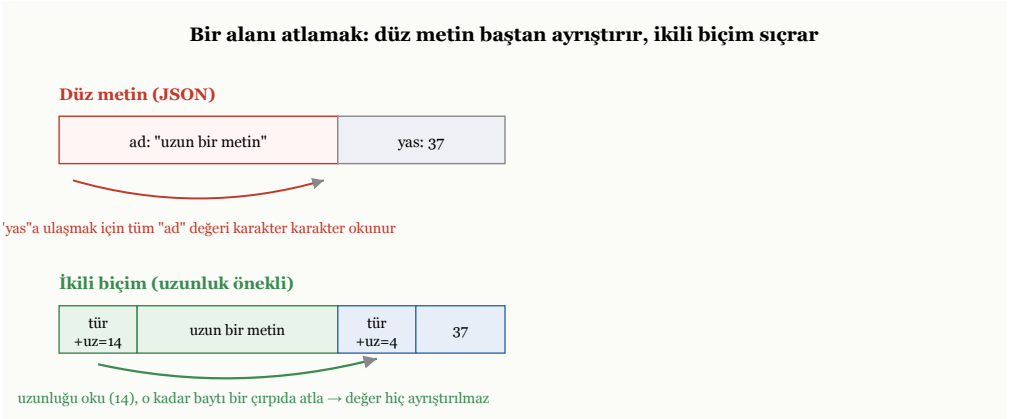
Bu yüzden ciddi belge veritabanları, JSON'u dışarıya — kullanıcıyla iletişimde — kullanırken, **içeride** daha zengin ve daha verimli bir **ikili gösterime** (binary encoding) çevirir. Bu ikili biçimler, JSON'un tanımadığı türleri (tarih, ikili veri, kesin sayılar) ekler ve veriyi, ayrıştırmadan doğrudan üzerinde işlem yapılabilecek şekilde kodlar; örneğin bir alanın değerine, tüm belgeyi metin olarak okumadan, doğrudan atlayabilirsiniz.

Bu ailenin nasıl çalıştığını anlamak öğreticidir, çünkü hepsi aynı temel hileyi kullanır: **uzunluk öneki** (length prefix). Düz metin JSON'da bir değerin nerede bittiğini bulmanın tek yolu, onu karakter karakter okuyup kapanış parantezini ya da tırnağı görmektir; yani bir alanı atlamak için bile onu baştan sona ayrıştırmanız gerekir. İkili biçimler ise her değerin önüne, türünü ve baytça uzunluğunu yazar. Böylece bir alanı atlamak isteyen okuyucu, uzunluğu okur ve o kadar baytı bir çırpıda atlar — değeri hiç ayrıştırmadan. Bir milyon belgenin yalnızca tek bir alanını okuyan bir sorguda bu farkın somut anlamı, gereksiz ayrıştırmadan tümüyle kurtulmaktır; yedinci ve sekizinci bölümlerde sorgu hızının büyük bölümünün böyle "atlamalardan" geldiğini göreceğiz.

Bu ikili biçimlerin en bilinenleri öğretici biçimde farklı dengeler tutar. BSON (Binary JSON), belge veritabanları dünyasında yaygınlaşan biçimdir; JSON'a tarih, ikili veri ve farklı tamsayı türleri ekler ve belge ile dizi uzunluklarını öne yazarak alan atlamayı kolaylaştırır — yer açısından metin JSON'dan her zaman daha

[1] Douglas Crockford, "Introducing JSON," json.org, https://www.json.org/.

[2] *ECMA-404: The JSON Data Interchange Syntax*, Ecma International, 2. baskı, 2017; T. Bray (ed.), *RFC 8259: The JavaScript Object Notation (JSON) Data Interchange Format*, IETF, 2017.



Şekil 4.2: Uzunluk öneki sayesinde ikili biçim, atlanacak alanı hiç ayrıştırmadan geçer.

küçük değildir, ama gezilmesi ucuzdur.[3] MessagePack, asıl amacı en küçük baytı sıkıştırmak olan, ağ üzerinde yer kazanmaya odaklı bir biçimdir; aynı veriyi JSON'dan belirgin biçimde daha az baytla taşır. CBOR (Concise Binary Object Representation), benzer bir hedefi bir internet standardı titizliğiyle izler ve kısıtlı, gömülü aygıtlarda bile güvenle ayrıştırılabilecek, genişletilebilir bir biçim olarak tanımlanmıştır.[4] Bu üçü, "JSON'un kavramsal modelini koru, ama metnin verimsizliğini at" fikrinin üç ayrı yorumudur.

Üçüncü kısımda OxiDB'nin tam olarak böyle bir iç ikili gösterim kullandığını ve bunun sorgu hızına nasıl katkı sağladığını göreceğiz. Şimdilik akılda tutulacak nokta şu: JSON belge modelinin yüzüdür, ama bir veritabanının içinde yaşayan biçim genellikle JSON'un daha zengin, daha sıkı bir akrabasıdır.

4.5 Belgeler ve koleksiyonlar

Belgeler tek başlarına durmaz; bir arada gruplanırlar. Bu gruplara genellikle **koleksiyon** denir — ilişkisel dünyadaki tablonun belge dünyasındaki karşılığı. Ama önemli bir farkla: bir tablo, içindeki tüm satırların aynı sütunlara sahip olmasını dayatır; bir koleksiyon ise içindeki belgelerin birbirine benzemesini **zorunlu kılmaz**. Aynı koleksiyonda, farklı alanlara sahip belgeler bir arada durabilir. Pratikte bir koleksiyondaki belgeler genelde benzer biçimdedir — çünkü hepsi aynı tür "şeyi" temsil eder — ama bu benzerlik, sistemin dayattığı bir kural değil, uygulamanın gözettiği bir gelenektir.

Her belgenin, kendisini koleksiyon içinde benzersiz biçimde tanımlayan bir **kimliği** vardır. Bu kimlik, belgeye doğrudan, taramadan erişmenin anahtarıdır; ikinci bölümde anahtar-değer modelinden söz ederken değindiğimiz "anahtarla erişim" fikri burada da yaşar. Belge veritabanları, bir belgeyi kimliğiyle bulmayı her zaman çok hızlı kılacak şekilde tasarlanır. Diğer alanlara göre arama ise — yedinci bölümde göreceğimiz — indekslerin işidir.

4.6 "Şemasızlık" yanılgısı: şema-okumada ve şema-yazmada

Belge veritabanları sık sık "şemasız" diye anılır, ama bu niteleme yanıltıcıdır ve düzeltilmesi gerekir. Veri her zaman bir şemaya sahiptir — yani bir biçime, bir beklentiye. Soru, o şemanın **nerede yaşadığı** ve **ne zaman dayatıldığıdır**.

İlişkisel modelde şema **yazma anında** dayatılır: bir satırı yazmadan önce tablonun tanımına uymak zorundadır, aksi halde yazma reddedilir. Buna "şema-yazmada" denir. Şema merkezîdir, açıktır ve veritabanı

[3] *BSON (Binary JSON) Specification*, MongoDB Inc., bsonspec.org.
[4] C. Bormann ve P. Hoffman, *RFC 8949: Concise Binary Object Representation (CBOR)*, IETF, 2020.

tarafından zorlanır. Avantajı, verinin her zaman beklenen biçimde olmasıdır; veriyi okuyan hiç kimse "acaba bu alan var mı, türü doğru mu" diye endişelenmek zorunda değildir. Dezavantajı, biçimi değiştirmenin pahalı olmasıdır: yeni bir alan eklemek, var olan tüm satırları ilgilendiren bir değişiklik gerektirir.

Belge modelinde ise şema tipik olarak **okuma anında** anlam kazanır: veritabanı, yazdığınız belgenin biçimine karışmaz; biçimi yorumlama işi, veriyi okuyan uygulamaya düşer. Buna "şema-okumada" denir. Şema kaybolmamıştır; yalnızca veritabanından uygulamaya, açıktan örtüye taşınmıştır. Avantajı esnekliktir: biçimi değiştirmek için merkezi bir tanım güncellemek gerekmez; farklı belgeler farklı biçimlerde olabilir; veri zamanla evrilebilir. Dezavantajı, şemayı zorlama sorumluluğunun uygulamaya geçmesidir.

Bu devrin pratik sonuçları vardır. Esneklik, bir armağandır: gereksinimler değiştiğinde göç (migration) denen o zahmetli, tüm veriyi dönüştüren işlemleri yapmadan, yeni alanları yazmaya başlayabilirsiniz. Ama aynı esneklik bir tuzaktır: veritabanı sizi tutarlı tutmadığı için, zamanla **biçim kayması** oluşabilir — aynı koleksiyonda kimi belgelerde bir alan vardır, kimilerinde yoktur; kiminde sayıdır, kiminde metin; kiminde `tel`, kiminde `telefon`. Veriyi okuyan kod, bu çeşitliliğe karşı **savunmacı** olmak, eksik ya da beklenmedik alanlara hazırlıklı olmak zorunda kalır. "Şemasızlık", "şema düşünmek gerekmiyor" demek değildir; tersine, şema disiplinini veritabanı yerine sizin taşımanız gerektiği anlamına gelir. İyi kullanılan belge veritabanlarında bu disiplin yok olmaz; uygulama katmanında, bilinçli biçimde sürdürülür.

Bu yüzden belge dünyası, şemayı tümüyle terk etmek yerine, onu **isteğe bağlı** ve **kademeli** kılan bir orta yol geliştirdi. Bunun en yaygın aracı, JSON Schema adlı, bir belgenin uyması beklenen biçimi yine JSON'la tanımlayan bir betimleme dilidir: hangi alanların zorunlu olduğunu, bir alanın hangi türde olması gerektiğini, bir sayının hangi aralıkta kalacağını ya da bir metnin hangi örüntüye uyacağını ifade eder.[5] Bu yaklaşımın inceliği, şemanın **ne zaman** dayatılacağını seçilebilir kılmasıdır. İsterseniz onu yazma anında bir doğrulama (validation) kuralı olarak veritabanına bağlar, kurala uymayan belgeleri reddedersiniz — böylece ilişkisel modelin şema-yazmada güvencesine yaklaşırsınız. İsterseniz şemayı yalnızca okuma anında, gelen veriyi denetlemek için kullanırsınız. Önemli olan kavrayış şudur: belge modelinde şema yok olmaz, yalnızca **zorlama anı ve yeri seçime bırakılır**. Olgun bir belge sisteminde tasarımcı, bu seçimi her koleksiyon için bilinçli olarak yapar: kararlı, kritik veride şemayı sıkar; deneysel, hızla evrilen veride gevşek bırakır.

4.7 Belge modelinin kalbindeki karar: gömmek mi, atıfta bulunmak mı

Belge modeliyle veri tasarlarken karşılaşacağınız en sık ve en belirleyici karar şudur: birbiriyle ilişkili iki veri parçasını, birini diğerinin **içine gömerek** mi tek bir belgede tutmalı, yoksa ayrı belgelerde tutup birinden diğerine **atıfta bulunarak** (referansla) mı bağlamalı? Bu kararın iyi verilmesi, belge modelini ustaca kullanmanın özüdür. Üçüncü bölümdeki çoğaltma ödünleşimi, burada somut bir tasarım kararına dönüşür.

Gömmek, ilişkili veriyi ana belgenin içine yerleştirmektir. Bir blog yazısının yorumlarını yazının kendi belgesi içinde bir liste olarak tutmak buna örnektir. Gömmenin armağanları, üçüncü bölümde saydığımız yerellik kazançlarıdır: veriyi tek bir okumayla, parçaları toplamadan alırsınız; ve bir belgeyi değiştirmek atomik olduğu için, gömülü veriyi ana veriyle birlikte tutarlı biçimde güncelleyebilirsiniz. Bedeli ise iki katmanlıdır. Birincisi, eğer gömülü bilgi başka belgelerde de tekrarlanıyorsa, çoğaltmanın güncelleme maliyetini ve tutarsızlık riskini üstlenirsiniz. İkincisi ve daha sinsi olanı, **sınırsız büyüme** tehlikesidir: yorumları gömdüğünüz bir yazı yıllar içinde on binlerce yorum biriktirirse, o belge devasa büyür; her okumada — yalnızca başlığı görmek isteseniz bile — tüm o yorum yığınını taşımak zorunda kalırsınız. Sınırı belli olmayan, sürekli büyüyen listeleri gömmek, belge modelinin en yaygın tuzaklarından biridir.

Atıfta bulunmak, ilişkili veriyi ayrı bir belgede tutup, ona yalnızca kimliğiyle işaret etmektir — ilişkisel modelin paylaşılan-değerle bağ kurma fikrinin belge dünyasındaki karşılığı. Bunun armağanları normalleştirmenin armağanlarıdır: paylaşılan bilgi tek bir yerde durur, çoğaltma olmaz, değiştirmek ucuzdur ve belgeler şişmez. Bedeli ise yerelliğin kaybıdır: ilişkili veriyi almak için birden fazla okuma ya da uygulama tarafında bir birleştirme gerekir.

[5] *JSON Schema: A Media Type for Describing JSON Documents*, JSON Schema Org., taslak spesifikasyon, json-schema.org.

Bu iki seçenek arasında karar verirken birkaç yol gösterici ilke yardımcı olur. İlişkili veri, ana varlığın **ayrılmaz bir parçasıysa** ve onunla **birlikte okunup yazılıyorsa** — bir adresin bir kullanıcıya, bir sipariş kaleminin bir siparişe ait olması gibi — ve büyümesi **sınırlıysa**, gömmek doğaldır. İlişkili veri **birçok varlık tarafından paylaşılıyorsa**, **bağımsız olarak sorgulanıyorsa**, **sık değişiyorsa** ya da **sınırsız büyüyebiliyorsa**, atıfta bulunmak daha sağlamdır. Ve sık karşılaşılan çoktan-çoğa ilişkilerde — ikinci bölümden beri peşimizi bırakmayan o zorlu örüntü — neredeyse her zaman atıf tercih edilir, çünkü gömmek kaçınılmaz biçimde çoğaltmaya yol açar.

Bu kararın tek ve kesin bir doğru yanıtı yoktur; doğru yanıt, üçüncü bölümde gördüğümüz gibi, erişim örüntülerinize bağlıdır. Aynı "kullanıcı ve adres" ilişkisi, bir uygulamada gömülecek, başka birinde referansla bağlanacaktır. Belge modelini iyi kullanmak, bu kararı her ilişki için bilinçli biçimde, erişim örüntülerine bakarak vermektir.

4.8 Atomikliğin sınırı belgedir

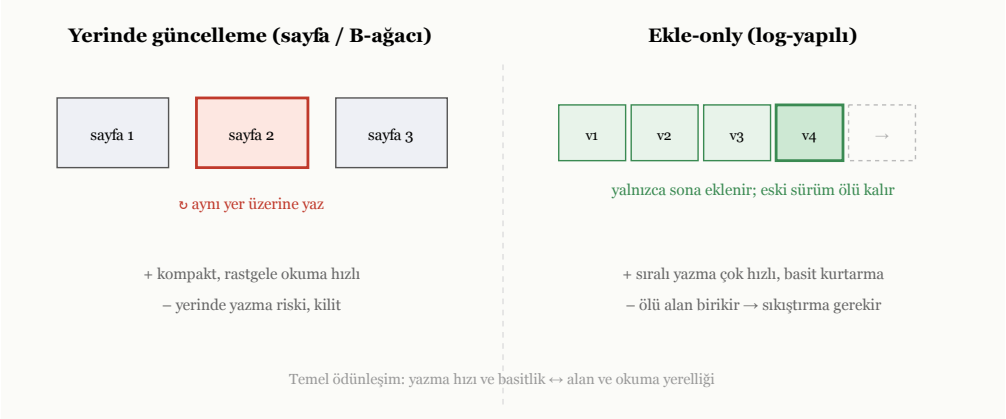
Son olarak, ileride önemli olacak bir noktayı şimdiden tohumlayalım. Belge veritabanlarında, üzerinde "ya hep ya hiç" güvencesi en kolay verilen birim, tek bir belgedir. Bir belgeyi değiştirdiğinizde, o değişiklik ya tümüyle gerçekleşir ya da hiç gerçekleşmez; arada bir durum olmaz. İşte gömmenin daha önce değinmediğimiz bir avantajı buradan doğar: ilişkili veriyi tek bir belgede topladığınızda, onları birlikte, tek bir atomik işlemde güncelleyebilirsiniz. Veriyi ayrı belgelere dağıttığınızda ise, birden çok belgeyi tutarlı biçimde değiştirmek daha güçlü işlem güvenceleri gerektirir. Bu, üçüncü bölümdeki "tutarlı kalması gereken birim ne kadar büyük" sorusunun belge modelindeki yankısıdır ve onuncu bölümde işlemleri ele alırken bütün ağırlığıyla geri gelecek.

Böylece Kısım I'i tamamlamış olduk. Artık belgenin ne olduğunu — alanlardan, iç içe yapılardan, kendini tanımlayan veriden oluştuğunu; JSON'la gösterilip içeride daha zengin bir ikili biçimde yaşadığını; koleksiyonlarda gruplandığını; şemasının yazma yerine okuma anında anlam kazandığını; ve onunla çalışmanın kalbinde gömme-referans kararının durduğunu — biliyoruz. Şimdiye dek hep *mantıksal* düzeyde, yani veriyi nasıl düşündüğümüz düzeyinde kaldık. Kısım II ile birlikte perdeyi aralıyor ve *fiziksel* düzeye iniyoruz: bir belge veritabanı, tüm bu belgeleri diske tam olarak nasıl yazar, çökmeden nasıl korur, aradığımızı taramadan nasıl bulur ve sorularımızı nasıl yanıtlar? Bir sonraki bölümde işin en altından, depolama motorundan başlıyoruz.

Bölüm 5

Depolama Motorları: Sayfa Tabanlı, Append-Only, LSM ve B-Ağaçları

Kısım I boyunca hep mantıksal düzeyde kaldık: veriyi nasıl düşündüğümüzle, belgenin ne olduğuyla ilgilendik. Şimdi perdeyi aralıyor ve fiziksel düzeye iniyoruz. Bir belge veritabanı, tüm o belgeleri diske tam olarak nasıl yazar ve geri okur? Bu sorunun yanıtını veren bileşene **depolama motoru** denir ve o, bir veritabanının kalbidir. Bu bölümde, depolama motorlarının çözmek zorunda olduğu temel gerilimi ve bu gerilime tarih boyunca verilen başlıca yanıtları — sayfa tabanlı B-ağaçlarını ve append-only/log-yapılı yaklaşımları — inceleyeceğiz. Buradan sonraki birkaç bölüm daha teknik olacak, ama yöntemimiz aynı: her tasarımı, çözmeye çalıştığı somut sorundan başlayarak anlamak.



Şekil 5.1: Yerinde güncelleme ile ekle-only depolamanın temel ödünleşimi.

5.1 Depolama motoru ne yapar

Bir depolama motorunu, veritabanının geri kalanından soyutlayarak düşünmek yararlıdır. Üstündeki katmanlar — sorgu işleyici, indeksler, işlem yöneticisi — ona iki temel istekle gelir: “şu kimliğe sahip belgeyi sakla” ve “şu kimliğe sahip belgeyi geri ver.” Depolama motorunun işi, bu basit isteklerin altındaki zor sorunu çözmektir: baytları diskte nereye, hangi düzende yerleştireceğini ve onları nasıl geri bulacağını kararlaştırmak. Bu kararlar, veritabanının ne kadar hızlı yazıp okuyacağını, ne kadar yer kaplayacağını ve çökmeye karşı ne kadar dayanıklı olacağını doğrudan belirler. Bu yüzden depolama motoru tasarımı, veritabanı mühendisliğinin en belirleyici tercihidir.

Önemli bir noktayı baştan ayıralım: depolama motoru, belgenin **içeriğiyle** ilgilenmez. Onun için bir belge, bir kimliğe bağlı bir bayt yığınıdır. Belgenin alanlarını yorumlamak, üzerinde sorgu çalıştırmak üst katmanların işidir; depolama motoru yalnızca o bayt yığınını güvenilir biçimde saklayıp geri vermekle yükümlüdür. Bu ayrım, depolama motorlarının neden farklı veri modellerinin altında — ilişkisel de olsa belge de olsa — aynı ilkelerle çalışabildiğini açıklar.

5.2 Her şeyi belirleyen kısıt: disk belleğe benzemez

Birinci bölümde, diskin belleğe hiç benzemediğine değinmiştik. Depolama motorlarını anlamak için, bu farkın somut sonuçlarını derinleştirmemiz gerekir; çünkü bu bölümdeki her tasarım kararı, doğrudan diskin doğasından doğar.

Diskin üç temel huyu vardır. Birincisi, **yavaştır**: belleğe erişim nanosaniyeler alırken, diske erişim kat be kat daha uzun sürer. Bu yüzden depolama motorunun en büyük amacı, gereksiz disk erişimlerinden kaçınmaktır. İkincisi, disk **ardışık erişimi**, dağınık erişimden çok daha iyi yapar. Diskten art arda gelen bir bölgeyi okumak ya da dosyanın sonuna eklemeye devam etmek hızlıdır; oysa diskin orasından burasından, rastgele konumlara erişmek çok daha yavaştır. Bu fark, geleneksel dönen disklerde uçurum kadar büyüktü; katı hal sürücülerinde daralsa da hâlâ önemlidir. Üçüncüsü, disk veriyi **bloklar halinde** okuyup yazar: tek bir baytı bile değiştirmek isteseniz, sistem o baytı içeren koca bir bloğu okuyup, değiştirip, geri yazar. Yani küçük, dağınık değişiklikler diskte orantısız bir maliyet taşır.

Bu üç huy, depolama motoru tasarımına tek bir buyruk dayatır: **diske ardışık, büyük ve seyrek yaz; rastgele, küçük ve sık erişimden kaçın.** Bu bölümdeki tüm yaklaşımlar, bu buyruğa farklı biçimlerde uymaya çalışan stratejilerdir.

5.3 Temel gerilim: okumaya mı, yazmaya mı eniyilemek

Depolama motoru tasarımının kalbinde tek bir gerilim yatar ve onu en baştan görmek, geri kalan her şeyi aydınlatır: bir veri yerleşimi, hem okumayı hem yazmayı aynı anda en iyi yapamaz. Bu ikisi arasında bir ödünleşim vardır.

Neden? Çünkü okumayı hızlandırmak için veriyi **düzenli** tutmak istersiniz — örneğin sıralı, dengeli bir yapıda — ki aradığınızı çabucak bulasınız. Ama veriyi düzenli tutmak, her yeni yazmada o düzeni korumak için diskin ortasına, "doğru yere" müdahale etmeyi gerektirir; bu da rastgele yazma demektir, yani diskin en sevmediği şey. Tersine, yazmayı hızlandırmak için veriyi olduğu gibi, geldiği sırada, dosyanın sonuna ardışık olarak eklersiniz; bu yazma için idealdir, ama o zaman veri düzensiz birikir ve aradığınızı bulmak zorlaşır. Düzen, okumanın dostu ama yazmanın düşmanıdır; düzensizlik ise tersi. İşte iki büyük depolama felsefesi — sayfa tabanlı B-ağaçları ve log-yapılı yaklaşımlar — bu gerilimin iki ucundan doğar.

5.4 Sayfa tabanlı B-ağaçları: yerinde güncelleme

Birinci felsefe, veriyi **düzenli tutmayı** önceler ve onlarca yıl boyunca veritabanı dünyasına egemen oldu. Bu yaklaşımda disk, sabit boyutlu **sayfalara** bölünür — her sayfa, belirli sayıda kaydı tutan, diskten tek seferde okunup yazılan bir blok. Veri, bu sayfalar üzerinde **B-ağacı** denen bir yapıyla düzenlenir.[1]

B-ağacını, devasa, çok katmanlı bir dosyalama dolabı gibi düşünebilirsiniz. En üstte, "A–M arası şu çekmecede, N–Z arası bu çekmecede" diyen bir yönlendirme katmanı vardır. Onun altında daha ince ayrımlar, en altta ise verinin kendisini tutan yapraklar bulunur. Bir kaydı aramak için tepeden başlar, her adımda doğru dala saparak yapraklara inersiniz; birkaç adımda, milyonlarca kayıt arasında bile, aradığınıza ulaşırsınız.

[1] R. Bayer ve E. M. McCreight, "Organization and Maintenance of Large Ordered Indexes," *Acta Informatica* 1(3), 1972.

Üstelik veri sıralı tutulduğu için, "şu değerle bu değer arasındaki tüm kayıtlar" gibi **aralık sorgularını** da verimli yanıtlarsınız — yapraklar arasında yan yana ilerlemek yeterlidir. B-ağacı, hem tekil hem aralık okumalarında olağanüstü iyidir; bu yüzden okuma-yoğun, sorgu-zengin sistemlerin gözdesi olmuştur.

5.4.1 Düğümün içi: fanout, dolum oranı, dallanma derinliği

Bu dosyalama dolabı benzetmesinin altındaki sayısal davranışı görmek, B-ağacının neden bu kadar etkili olduğunu açıklar. Her düğüm bir diske bir sayfaya, yani sabit boyutlu bir bloğa karşılık gelir — tipik olarak dört ya da sekiz kilobaytlık bir blok. Bir iç (ara) düğümün içine, mümkün olduğunca çok sayıda **ayrıcı anahtar** (separator key) ve onlara karşılık gelen çocuk işaretçisi sığdırılır. Bir düğümün kaç çocuğa dallandığı sayısına o ağacın **fanout'u** (çıkış genişliği) denir. Anahtarlar küçük olduğunda, tek bir sayfaya yüzlerce ayrıcı sığabilir; yani fanout yüzlerle ölçülür. Bunun derinliğe etkisi çarpıcıdır: fanout'u iki yüz olan bir ağaç, tek seviyede iki yüz, iki seviyede kırk bin, üç seviyede sekiz milyon, dört seviyede bir buçuk milyar kaydı kapsar. Yani milyarlarca kayıt arasında bile, kök düğümden bir yaprağa inmek topu topu üç-dört sayfa erişimi alır. B-ağacının vaadi tam da budur: derinliği, veri miktarının **logaritması** kadar yavaş büyür ve fanout büyük olduğu için bu logaritmanın tabanı büyüktür — yani ağaç şaşırtıcı derecede sığ kalır.

Burada çoğu modern veritabanının kullandığı belirli bir biçimi anmak gerekir: **B+ağacı**. Saf B-ağacında veri her düğümde bulunabilirken, B+ağacında tüm gerçek kayıtlar yalnızca **yapraklarda** durur; iç düğümler yalnızca yol gösteren ayrıcı anahtarları taşır. Bu ayrımın iki büyük getirisi vardır. Birincisi, iç düğümler veri taşımadığı için daha çok ayrıcı sığdırır, fanout artar, ağaç daha da sığ olur. İkincisi — ve aralık sorguları için kritik olanı — yapraklar bir **bağlı liste** ile soldan sağa birbirine zincirlenir. Bir aralığın başını bir kez bulduktan sonra, ağaca tekrar tepeden inmeden, yapraktan yaprağa yürüyerek tüm aralığı sırayla okursunuz. Bu, "şu tarihten bu tarihe kadar tüm siparişler" gibi sorguları neredeyse ardışık bir okuma hızına indirir.

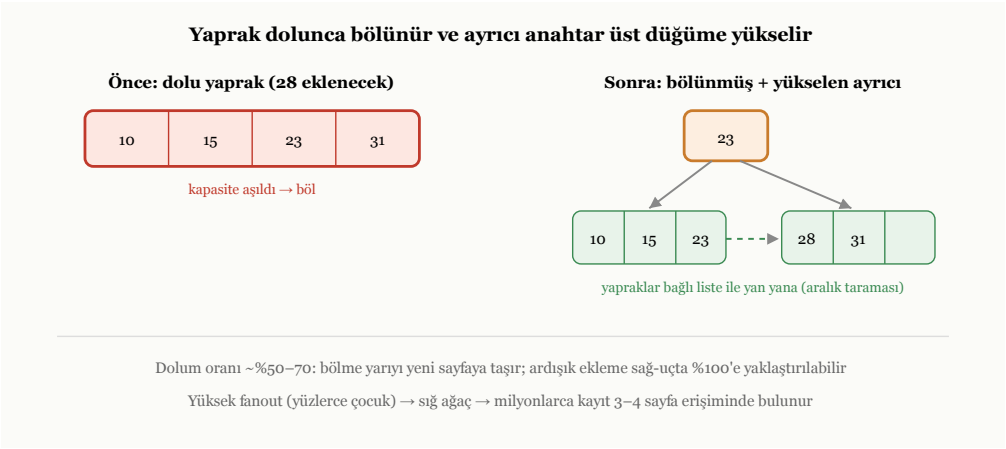
Düğümlerin ne kadar dolu tutulacağı da bir tasarım kararıdır ve buna **dolum oranı** (fill factor) denir. Bir düğüm asla tamamen boş ya da tamamen dolu tutulmaz; tipik olarak yarısıyla tamamı arasında bir doluluk hedeflenir. Çok dolu tutmak, her küçük eklemede taşma ve bölünme riskini artırır; çok boş tutmak ise yer israf eder ve ağacı gereksiz şişirir. Rastgele anahtar ekleme örüntülerinde ağaçlar pratikte yaklaşık yüzde yetmiş civarı dolulukta dengelenir; oysa anahtarlar hep artan sırada (örneğin zaman damgası ya da otomatik kimlik) geliyorsa, ekleme hep en sağdaki yaprağa düşer ve sayfalar neredeyse tamamen doldurulabilir — bu, depolamayı sıkılaştıran, bilinçli olarak istenen bir örüntüdür.

5.4.2 Bölme ve birleştirme: dengeyi korumanın bedeli

B-ağacının "dengeli" kalması, yani her yaprağa aynı sayıda adımda inilmesi, kendi kendine olmaz; ekleme ve silme sırasında ağacın kendini onarmasıyla sağlanır. Bir yaprağa yeni bir anahtar eklemek istediğinizde ve o yaprak zaten doluysa, yaprak **bölünür** (split): içindeki anahtarlar kabaca ikiye ayrılır, yarısı yeni bir sayfada kalır, ortadaki ayrıcı anahtar bir üst düğüme **yükselir**. Eğer o üst düğüm de doluysa, bölünme yukarı doğru zincirleme yayılır; en kötü durumda köke kadar çıkar ve kök bölündüğünde ağacın boyu bir artar. Tersine, silmeler bir yaprağı belirli bir alt eşiğin altına düşürürse, o yaprak bir komşusuyla **birleştirilir** (merge) ya da komşusundan anahtar **ödünç alır** (rebalance); böylece hiçbir düğüm fazla seyrek kalmaz. Bu bölme-birleştirme dansı, ağacın hep dengeli ve sığ kalmasının bedelidir.

Bedeli ise yazmada ortaya çıkar. Bir kaydı değiştirdiğinizde, B-ağacı onu diskte ait olduğu sayfada, **yerinde** günceller. Bu, az önce diskin en sevmediği şey dediğimiz rastgele yazma demektir: değiştirilecek sayfa diskin neresindeyse, oraya gidilip o sayfa okunur, değiştirilir, geri yazılır. Dahası, az önceki bölme-birleştirme dansı her seferinde birden çok sayfaya dokunur; üstelik çoğu sistem bir sayfanın yarısını değil **tüm sayfayı** yeniden yazar. Tek bir küçük değişikliğin diske birden çok blok, hatta kilobaytlarca veri yazılmasına yol açmasına **yazma büyütmesi** (write amplification) denir ve B-ağaçlarının doğal bir maliyetidir. Bir de eşzamanlılık zorluğu vardır: birçok işlem aynı sayfalara aynı anda dokunabileceği için, sayfaların dikkatlice kilitlenmesi gerekir; bunun inceliklerine on birinci bölümde döneceğiz.

Özetle B-ağacı yaklaşımı, okumayı en üst düzeye çıkarmak için veriyi düzenli ve yerinde tutar; bunun karşılığında rastgele yazma ve yazma büyütmesi maliyetini öder.



Şekil 5.2: Yaprak dolunca bölünür; ortadaki ayrıcı anahtar üst düğüme yükselir.

5.5 Append-only ve log-yapılı yaklaşımlar: asla üzerine yazma

İkinci felsefe, gerilimin öteki ucundan başlar ve **yazmayı** önceler. Temel fikri tek bir cümleyle özetlenebilir: **var olan veriyi asla yerinde değiştirme; her yazmayı dosyanın sonuna ekle.** Bunu, sürekli yeni satırlar eklediğiniz, ama hiçbir eski satırı silmediğiniz bir muhasebe defterine benzetebilirsiniz. Bir kaydı güncellemek istediğinizde, eski kaydı bulup değiştirmezsiniz; onun yeni hâlini defterin sonuna yazarsınız. En son yazılan, geçerli olandır.

Bu yaklaşımın yazmadaki üstünlüğü açıktır: her yazma, dosyanın sonuna **ardışık** bir ekleme olduğu için, diskin en sevdiği erişim biçimidir. Rastgele yazma yoktur; yazma büyütmesi en aza iner. Bu yüzden log-yapılı (append-only) motorlar, yazma- yoğun iş yüklerinde B-ağaçlarını rahatça geçer.

Ama bedava öğle yemeği yoktur; bu yaklaşım iki yeni sorun doğurur. Birincisi **okuma sorunudur**. Bir kaydın en güncel hâli defterin neresinde diye, dosyayı baştan sona taramak kabul edilemez. Bu yüzden append-only motorlar, ayrı bir **dizin** tutar: her kaydın kimliğinden, o kaydın en güncel hâlinin dosyada durduğu konuma — yani dosya başından kaç bayt ilerideki ofsete — bir eşleme. Yazarken bu dizini güncellersiniz; okurken önce dizine bakıp konumu öğrenir, sonra doğrudan o konuma gidersiniz. Bu dizin, bir karma tablo (hash) biçiminde bellekte tutulabilir — ki o zaman kimlikten konuma erişim sabit zamanlıdır, ama tüm anahtarların belleğe sığması gerekir — ya da diskte sıralı bir yapıda tutulabilir. Önemli bir özellik şudur: bu yaklaşımda **veri kaydının kendisi** kimlik, uzunluk ve içeriği gömülü olarak taşıyabilir; böylece dizin çökmede kaybolsa bile, dosyayı baştan sona bir kez tarayarak yeniden inşa edilebilir. Yani dizin, hızlandırıcı bir kestirmedir; gerçeğin tek kaynağı her zaman append-only veri dosyasının kendisidir. (Üçüncü kısımda OxiDB'nin tam olarak böyle bir kimlik-konum dizini kullandığını ve her kaydı bir durum baytı, uzunluk alanı ve yük olarak nasıl sakladığını göreceğiz.)

İkincisi **ölü alan sorunudur**. Madem eski kayıtların üzerine yazmıyoruz, onlar dosyada öylece durmaya devam eder. Bir kaydı yüz kez güncellerseniz, dosyada o kaydın yüz eski kopyası birikir; yalnızca sonuncusu geçerlidir, gerisi **ölü alandır**. Zamanla dosya, çoğu artık geçersiz kayıtlarla şişer. Bu şişkinliği gidermek için, arada bir, dosyayı baştan yazıp yalnızca yaşayan (geçerli) kayıtları yeni bir dosyaya kopyalamak gerekir; ölü kayıtlar geride bırakılır. Bu temizlik işlemine **sıkıştırma** (compaction) denir ve append-only motorların ayrılmaz bir parçasıdır. Sıkıştırmayı, defter dolup taştığında oturup yalnızca hâlâ geçerli satırları temiz bir deftere geçirmeye benzetebilirsiniz. (Üçüncü kısımda OxiDB'nin sıkıştırmayı ne zaman ve nasıl yaptığına ayrı bir bölüm ayıracağız.)

5.6 LSM ağaçları: log-yapılının olgun hâli

Append-only fikrini bir adım ileri taşıyan ve birçok modern belge ve geniş-sütun veritabanının kalbinde yatan tasarıma **LSM ağacı** denir — açılımı "log-yapılı birleştirmeli ağaç".[2] Saf append-only yaklaşımının okuma zorluğunu, akıllıca bir düzenlemeyle hafifletir.

LSM'nin fikri şudur: gelen yazmaları önce **bellekte**, sıralı bir yapıda biriktir. Bu bellek içi sıralı tampona **memtable** denir; genellikle hızlı eklemeye ve sıralı taramaya elveren bir ağaç ya da atlama listesi olarak tutulur. Bellek hızlı olduğu için bu yazmalar anında kabul edilir. (Bu noktada akla bir soru gelir: memtable bellektedir, peki çökerse ne olur? Yanıt, bir sonraki bölümün konusu olan yazma-öncesi günlüktür; LSM her yazmayı memtable'a koymadan önce ayrıca bir günlüğe ekler, böylece çökmede memtable yeniden inşa edilebilir.)

Memtable belirli bir boyuta ulaştığında, salt-okunur kılınır ve içeriği tek seferde, baştan sona **sıralı** biçimde diske bir dosya olarak boşaltılır. Bu sıralı, değişmez disk dosyasına **SSTable** (sıralı-dizili tablo) denir — yine ardışık yazma, yine diskin sevdiği biçim, üstelik bir kez yazıldıktan sonra asla değiştirilmez. Zamanla diskte böyle birçok SSTable birikir. Arka planda çalışan bir süreç, bu dosyaları periyodik olarak **birleştirir** (merge): birkaç sıralı dosyayı, sıralı listeleri birleştirme tekniğiyle tek bir büyük sıralı dosyaya kaynaştırır, bu sırada bir anahtarın eski sürümlerini ve silinmiş kayıtları ayıklar. Bu da bir sıkıştırmadır — append-only modelde az önce gördüğümüz temizliğin daha düzenli, kademeli bir biçimi.

5.6.1 İki birleştirme stratejisi: leveled ve size-tiered

Sıkıştırmanın *nasıl* düzenleneceği, LSM motorlarının ödünleşim eğrisini büyük ölçüde belirleyen iki ana okula ayrılır. Birincisi **seviyeli sıkıştırma** (leveled compaction): SSTable'lar seviyelere yerleştirilir ve her seviye, bir öncekinden kabaca on kat daha büyüktür. En alttaki seviye (L0) dışında, her seviyenin içindeki dosyaların anahtar aralıkları **birbiriyle çakışmaz**; yani bir seviyede belirli bir anahtarı tutabilecek en fazla bir dosya vardır. Bu, okumayı keskinleştirir — bir kaydı bulmak için seviye başına en fazla bir dosyaya bakılır — ama bir bedeli vardır: üst seviyeye boşaltılan veri, alt seviyenin çakışan dosyalarıyla sürekli yeniden birleştirildiği için, aynı veri ömrü boyunca defalarca yeniden yazılır; yani yazma büyütmesi yüksektir.

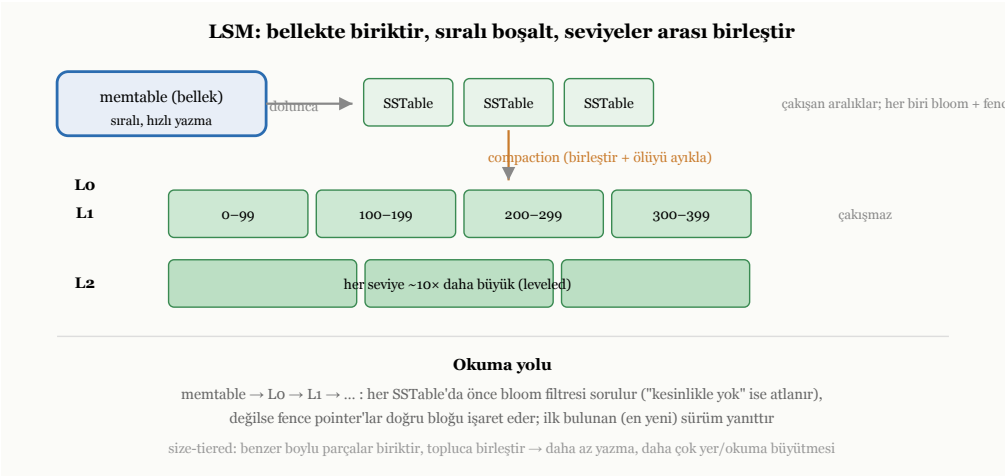
İkincisi **boyut katmanlı sıkıştırma** (size-tiered compaction): benzer boyuttaki SSTable'lar bir araya biriktirilir ve yeterince birikince hepsi tek seferde tek bir büyük dosyaya birleştirilir. Bu, her veriyi daha az kez yeniden yazdığı için yazma büyütmesini düşürür; karşılığında aynı anahtarın birden çok katmanda kopyası bulunabildiği için hem yer büyütmesi hem okuma büyütmesi artar. Kabaca: seviyeli strateji okuma-yoğun yükleri, boyut katmanlı strateji yazma-yoğun yükleri kayırır. Bu seçim, bir önceki bölümdeki "okumaya mı yazmaya mı eniyilemek" gerilimini, tek bir motorun içinde ayarlanabilir bir kadrana dönüştürür.

5.6.2 Okuma yolunu kurtaran iki yardımcı: bloom filtresi ve fence pointer

LSM, yazmada olağanüstü hızlıdır, çünkü her yazma önce belleğe gider ve diske hep ardışık yazılır. Okumada ise doğal bir bedel vardır: bir kaydı ararken, en güncel hâli hangi dosyada diye, memtable'dan başlayıp seviye seviye birçok SSTable'a bakmak gerekebilir. Bu maliyeti iki ayrı yardımcı yapı dramatik biçimde düşürür.

Birincisi **bloom filtresi** (bloom filter): her SSTable'ın yanına iliştirilen, çok küçük, olasılıksal bir üyelik testidir.[3] Bir anahtarın o dosyada olup olmadığını sorduğunuzda, bloom filtresi iki yanıttan birini verir: "kesinlikle yok" ya da "belki var". "Kesinlikle yok" yanıtı asla yanlış olmaz; bu yüzden, dosyadaki anahtarların büyük çoğunluğu için, diske hiç dokunmadan o dosyayı atlayabilirsiniz. "Belki var" yanıtı ise bazen yanlış alarm (false positive) olabilir — filtre bazen "belki var" der ama anahtar aslında yoktur — ancak bu yanlış alarm oranı, filtreye ayrılan bellek miktarıyla ayarlanabilir; tipik olarak yüzde bir dolayında tutulur. Böylece bir LSM araması, anahtarı içermeyen onlarca dosyayı, her biri için yalnızca birkaç bayttan hesaplanan bir sınama ile eler.

[2] P. O'Neil, E. Cheng, D. Gawlick ve E. O'Neil, "The Log-Structured Merge-Tree (LSM-Tree)," *Acta Informatica* 33(4), 1996.
[3] B. H. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Communications of the ACM* 13(7), 1970.



Şekil 5.3: LSM: memtable'dan SSTable'a boşaltma, seviyeler ve sıkıştırma.

İkincisi **fence pointer** (sınır işaretçileri, ya da seyrek indeks): bir SSTable sıralı olduğu için, motor o dosyanın her N kilobaytlık bloğunun başındaki anahtarı küçük bir tabloda saklar. Bloom filtresi "belki var" dediğinde, fence pointer'lar hangi bloğun aranan anahtarı içerebileceğini söyler; böylece koca dosyayı taramak yerine, tek bir bloğu okumak yeterli olur. Bloom filtresi *hangi dosyaya hiç bakmayacağını*, fence pointer *bir dosyanın neresine bakacağını* söyler; ikisi birlikte, LSM'nin okuma yolunu bir tam taramadan birkaç hedefli blok okumasına indirir. Yine de LSM, doğası gereği, yazmayı okumanın ve arka plan sıkıştırma yükünün önüne koyan bir tasarımdır.

5.7 Üç büyütme: kaçınılmaz üçgen

Bu noktada, depolama motorlarını karşılaştırmak için kullanışlı bir çerçeve ortaya çıkar. Her tasarım, üç tür "büyütme" arasında bir denge kurar ve hiçbiri üçünü birden en aza indiremez; biri azalırken genellikle bir diğeri artar.

Yazma büyütmesi, mantıksal olarak yazdığınız bir baytın diske kaç bayt olarak yazıldığıdır. B-ağaçlarında yerinde güncelleme ve sayfa bölünmeleri bunu artırır; LSM'de ise asıl yük, arka plandaki birleştirmenin aynı veriyi defalarca yeniden yazmasından gelir. **Okuma büyütmesi**, mantıksal bir okuma için diskten kaç parçaya bakıldığıdır; B-ağacı burada genellikle iyidir (birkaç sayfa), LSM ise birden çok parçaya bakmak zorunda kalabildiği için daha kötüdür. **Yer büyütmesi**, verinin diskte mantıksal boyutunun kaç katı yer kapladığıdır; append-only ve LSM'de biriken ölü kayıtlar bunu artırır, sıkıştırma azaltır.

Bu üçgenin sayısal davranışını bir an somutlaştırmak öğreticidir. Seviyeli sıkıştırmalı bir LSM'de, her seviye bir öncekinin on katıysa ve veri en üst seviyeye ulaşana dek her seviyede yaklaşık bir kez yeniden yazılıyorsa, on bir seviyelik bir ağaçta tek bir mantıksal yazma, ömrü boyunca diske onlarca kez yazılabilir — yazma büyütmesi on katı aşar. B-ağacında ise yazma büyütmesi, genellikle değiştirilen kaydın boyutu ile tüm sayfanın boyutu arasındaki orana bağlıdır: yüz baytlık bir kaydı değiştirmek için sekiz kilobaytlık bir sayfayı yeniden yazmak, seksen katlık bir büyütme demektir. Yer büyütmesi tarafında ise LSM ve append-only motorlarda biriken ölü kayıtlar, sıkıştırma yetişemediğinde veriyi mantıksal boyutunun iki-üç katına çıkarabilir; B-ağacında ise dolum oranı yüzde yetmiş dolayındaysa, sayfaların boş kalan üçte biri kalıcı bir yer büyütmesi olarak durur.

Bu üçgen, neden tek bir "en iyi" depolama motoru olmadığını net biçimde gösterir. B-ağaçları okuma büyütmesini en aza indirir, yazma ve belirli durumlarda yer büyütmesini öder; LSM ve append-only yaklaşımlar yazmayı en aza indirir, okuma ve yer büyütmesini öder. Hangisinin doğru olduğu, üçüncü bölümdeki o değişmez ilkeye geri döner: iş yükünüze, yani okuma mı yoksa yazma mı baskın olduğuna bağlıdır.

5.8 Anahtar ile değeri ayırmak: yazma büyütmesini kırmak

LSM'nin yazma büyütmesinin asıl kaynağına yakından bakınca, ince bir israf görülür. Sıkıştırma sırasında, motor bir SSTable'ı yeniden yazarken hem anahtarları hem de onlara bağlı **tüm değerleri** taşır. Oysa sıralamanın, birleştirmenin ve aramanın yalnızca **anahtarlara** ihtiyacı vardır; değerler — ki belge veritabanlarında bunlar koca koca belgelerdir — yalnızca taşınmak zorunda kalan ölü ağırlıktır. Bir kilobaytlık bir belge, anahtarı hiç değişmese bile, ağaçta yukarı taşındığı her seferde yeniden kopyalanır. Yazma büyütmesinin aslan payı buradan gelir.

Bu gözlemden doğan zarif bir fikir, **anahtar-değer ayrımıdır** (key-value separation): değerleri LSM ağacının içinde tutmak yerine, ayrı bir append-only **değer günlüğüne** yazmak ve ağaçta her anahtarın yanında yalnızca o değere bir işaretçi — küçük bir ofset — saklamak.[4] Bu sayede sıkıştırma yalnızca küçük anahtar-işaretçi çiftlerini taşır; ağır değerler yerinde durur ve hiç yeniden yazılmaz. Yazma büyütmesi çarpıcı biçimde düşer. Karşılığında iki yeni maliyet doğar: bir okuma artık önce ağaçtan işaretçiyi, sonra değer günlüğünden değeri okuduğu için iki adım gerektirir (katı hal sürücülerinde bu fazladan dağınık okuma ucuzdur, ama bedavadır da denemez); ve değer günlüğü de zamanla ölü değerlerle dolacağı için ona ayrı bir çöp toplama süreci gerekir. Bu fikir, depolama motoru tasarımının hâlâ canlı bir araştırma alanı olduğunu ve üçgenin köşelerinin sürekli yeniden müzakere edildiğini gösteren iyi bir örnektir.

5.9 Belleğin rolü ve veriyi belleğe yansıtmak

Şimdiye dek hep diskten söz ettik, ama hiçbir gerçek depolama motoru yalnızca diske güvenmez; bellekle disk arasında sürekli bir dans vardır. Sık erişilen veriyi bellekte tutmak, onu her seferinde diskten okumaktan kat kat hızlıdır. Bu yüzden depolama motorları, diskin en çok kullanılan parçalarını bellekte tutan bir **önbellek** işletir; sıcak veri bellekte kalır, soğuk veri diskte. Bu önbelleğin ne kadar büyük olacağı, hangi verinin tutulup hangisinin atılacağı, başlı başına bir tasarım konusudur ve on üçüncü bölümü tümüyle ona ayıracağız.

Belleği işin içine katmanın zarif bir yolu daha vardır ve onu burada tohumlamak yararlı olur: bir dosyayı, işletim sistemine söyleyerek, doğrudan belleğe **yansıtmak**. Bu teknikte dosya, sanki belleğin bir parçasıymış gibi davranır; ona erişmek, diskten açıkça okuma çağrısı yapmak yerine, yalnızca bellekteki bir adrese bakmak kadar basit hale gelir. Verinin gerçekten diskten belleğe ne zaman getirileceğine işletim sistemi karar verir ve bellek darda kaldığında o veriyi sessizce geri atabilir. Bu yaklaşım, depolama motorunun kendi önbelleğini elle yönetme yükünü büyük ölçüde işletim sistemine devretmesini sağlar. Üçüncü kısımda OxiDB'nin "disk-öncelikli" kipinde tam olarak bu tekniği kullandığını ve bunun bellek ayak izini nasıl küçülttüğünü ayrıntısıyla göreceğiz.

5.10 Bu bölümün bıraktığı yer ve bir sonrakine köprü

Bu bölümde, bir belge veritabanının baytları diske nasıl yerleştirdiğini — depolama motorunun işini — ve bu işin kalbindeki okuma-yazma gerilimini gördük. İki büyük felsefeyi tanıdık: veriyi düzenli ve yerinde tutarak okumayı önceleyen sayfa tabanlı B-ağaçları; ve veriyi asla üzerine yazmadan ardışık ekleyerek yazmayı önceleyen append-only ve LSM yaklaşımları. Bu yaklaşımların hiçbirinin mutlak üstün olmadığını, her birinin üç büyütme arasında farklı bir denge kurduğunu gördük.

Ama hangi felsefeyi seçerseniz seçin, henüz konuşmadığımız, sinsi bir sorun geride duruyor. Bir veriyi diske "yazdığınızda", o veri gerçekten kalıcı olmuş mudur? Sezgi "evet" der, ama gerçek çok daha incelikli. Yazma işlemleri yolda, çeşitli tamponlarda bekliyor olabilir; ve sistem tam o sırada çökerse, "yazdım" sandığınız veri buharlaşabilir — ya da daha kötüsü, yarısı yazılmış, bozuk bir hâlde kalabilir. Bir sonraki bölümde, depolama motorlarının en çetin sorumluluğuna, yani bir yazmanın gerçekten dayanıklı olduğundan emin olmaya

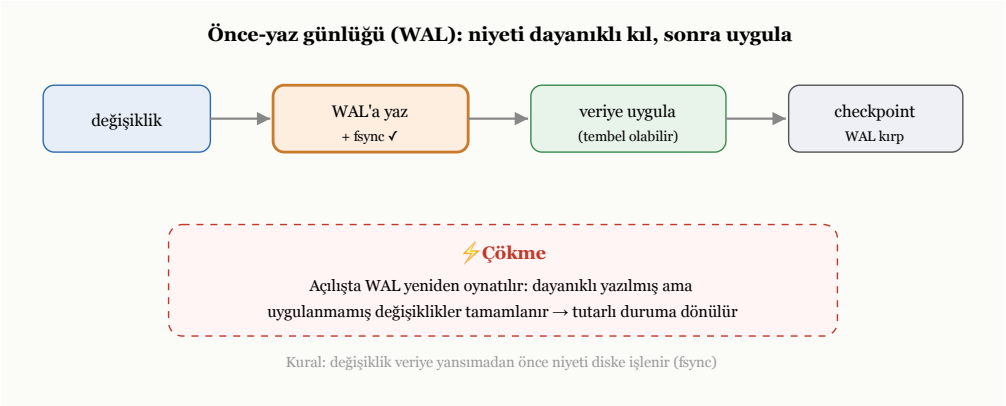
[4] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau ve R. H. Arpaci-Dusseau, "WiscKey: Separating Keys from Values in SSD-Conscious Storage," *14th USENIX Conference on File and Storage Technologies (FAST '16)*, 2016.

ve çökmenin ortasından tutarlı biçimde geri dönmeye — yazma-öncesi günlüğe ve fsync'in inceliklerine — eğileceğiz.

Bölüm 6

Dayanıklılık: Write-Ahead Log, fsync ve Çökmeden Kurtarma

Önceki bölümün sonunda sinsi bir soru bırakmıştık: bir veriyi diske "yazdığınızda", o veri gerçekten kalıcı olmuş mudur? Sezgi rahatça "evet" der. Ama bu sezgi yanlıştır ve bu yanlışın bedeli, kaybolan ya da bozulan veridir. Bu bölüm, bir veritabanının en çetin ve en az takdir edilen sorumluluğunu ele alır: bir yazmanın gerçekten dayanıklı olduğundan emin olmak ve sistem en beklenmedik anda çökerse, enkazın içinden tutarlı biçimde geri dönmek. Bu, veritabanını basit bir dosyadan ayıran en derin farklardan biridir.



Şekil 6.1: Önce-yaz günlüğü (WAL) akışı ve çökmeden kurtarma.

6.1 "Yazdım" demek neden yetmez

Bir programın diske yazma isteği, sandığınız gibi tek adımda diskin yüzeyine ulaşmaz. Aradan birçok katman geçer ve her katman, hız uğruna veriyi geçici olarak bekletir. Programınız veriyi yazdığında, o veri önce işletim sisteminin tuttuğu bir tampona düşer; işletim sistemi, performans için, bu veriyi hemen diske göndermeyebilir — biriktirip sonra topluca yazmayı tercih edebilir. Veri diske gönderildiğinde bile, diskin kendi denetleyicisinin de bir önbelleği vardır; veri orada bekleyip henüz fiziksel ortama işlenmemiş olabilir. Yani "yazdım" dediğiniz an ile verinin gerçekten kalıcı ortama oturduğu an arasında, görünmez bir gecikme ve birçok bekleme noktası vardır.

Sıradan zamanlarda bu katmanlar fark edilmez; veri eninde sonunda diske oturur ve her şey yolundadır. Sorun, tam da bu bekleme noktalarının birinde sistem çökerse — elektrik kesilirse, makine donarsa, süreç

aniden ölürse — ortaya çıkar. O an, henüz fiziksel ortama ulaşmamış olan her şey buharlaşır. Programınız "kaydettim" demiş, kullanıcıya "işlem tamam" demiş olabilir; ama o veri katmanların birinde beklerken yok olmuştur. Kullanıcı, yaptığını sandığı işlemin hiç gerçekleşmediğini sonradan öğrenir. Bir bankada, bir siparişte, bir tıbbi kayıtta bunun ne anlama geldiğini düşünün.

6.2 fsync: veriyi gerçekten diske zorlamak

Bu gecikmenin bir panzehiri vardır. İşletim sistemine, "bu veriyi şimdi, bu an, gerçekten kalıcı ortama yaz ve bunu bana onaylamadan geri dönme" diyebilirsiniz. Bu açık emre, geleneksel adıyla **fsync** denir. fsync döndüğünde, daha önce yazdığınız verinin gerçekten dayanıklı hale geldiğinden emin olabilirsiniz; artık çökme onu silemez.

Ama fsync pahalıdır. Çünkü tam da diskin en sevmediği şeyi yapmaya zorlar: beklemeyi bırakıp veriyi fiilen fiziksel ortama işlemek. Bu, milisaniyeler mertebesinde sürebilir — bilgisayar ölçeğinde upuzun bir zaman. Bir veritabanı her küçük yazma için fsync çağırsaydı, korkunç yavaş olurdu. İşte buradan, bu bölümün geri kalanını biçimlendiren gerilim doğar: **dayanıklılık güvenlidir ama yavaştır; onsuz hızlıdır ama tehlikelidir.** İyi bir veritabanı, bu ikisi arasında akıllıca bir denge kurmak zorundadır.

6.2.1 Boşaltma çağrılarının ailesi: fsync, fdatasync ve gerçek boşaltma

Burada, çoğu kitabın atladığı ama mühendislikte hayati olan bir ayrıma inmek gerekir; çünkü "diske yazdığını garanti et" emri tek bir şey değil, bir ailedir ve üyeleri farklı güvenceler verir. Sıradan boşaltma çağrısı, bir dosyanın hem **içeriğini** hem de o dosyaya ilişkin defter bilgisini — boyutu, son değişiklik zamanı gibi **üstveriyi** (metadata) — birlikte kalıcı kılar. Üstveriyi de boşaltmak fazladan bir disk işlemi gerektirir. Oysa bir veritabanı günlüğüne ekleme yaparken çoğu zaman dosyanın boyutu dışında bir üstveri umursanmaz; işte bu durum için, yalnızca veri içeriğini (ve boyutu güncellemek için gereken kadarını) boşaltan, üstverinin gerisini atlayan daha hafif bir çağrı vardır. Geleneksel adıyla **fdatasync** olarak bilinen bu çağrı, her commit'te bir disk işlemini kısarak gözle görülür bir hız kazandırabilir; çünkü saniyede on binlerce commit yapan bir sistemde, commit başına atlanan tek bir disk işlemi bile büyük fark yaratır.

İşin daha sinsi tarafı şudur: bazı sistemlerde ve bazı disk türlerinde, sıradan boşaltma çağrısı bile veriyi yalnızca **disk denetleyicisinin uçucu önbelleğine** ulaştırmakla yetinir, onun fiziksel ortama işlendiğini garanti etmez. Yani çağrı geri dönmüş, "kalıcı oldu" demiş olabilir; ama elektrik tam o an kesilirse, denetleyicinin önbelleğindeki o veri yine de kaybolur. Gerçek, sıfır kayıplı bir dayanıklılık için, denetleyiciye "önbelleğini de fiziksel ortama boşalt" diyen daha güçlü, daha da pahalı bir emir gerekir — bazı işletim sistemlerinde buna ayrı bir "tam boşaltma" bayrağıyla ulaşılır. Bir veritabanı mühendisinin verdiği en kritik ve en kolay gözden kaçan kararlardan biri, tam olarak bu güçlü boşaltmayı mı yoksa hafif olanı mı kullanacağıdır; çünkü bu seçim, "kaydettim" sözünün gerçekte ne kadar tuttuğunu belirler. (Üçüncü kısımda OxiDB'nin tam da bu güçlü boşaltmayı kullandığını ve bunun tek-kayıt güncellemelerinin hızına nasıl yansıdığını göreceğiz.)

6.3 İkinci tehlike: yarım kalmış yazma

Çökmenin yalnızca veriyi yok etmek gibi bir tehlikesi yoktur; ondan daha sinsi bir tehlikesi vardır. Bir yazma işleminin **ortasında** çökme olursa ne olur? Disk veriyi bloklar halinde yazdığı için, büyük bir kaydın bir kısmı yeni değeriyle, bir kısmı eski değeriyle, hatta arada bir yerde tümüyle bozuk kalabilir. Buna **yarım kalmış yazma** (torn write) denir. Sonuç, yalnızca eksik değil, **tutarsız** bir veridir: yarısı bir dünyaya, yarısı başka bir dünyaya ait bir kayıt. Böyle bir kaydı sonradan okumaya çalışmak, anlamsız ya da yanıltıcı sonuçlar doğurabilir.

Yarım kalmış yazmanın neden kaçınılmaz olduğunu görmek için, ölçeklerin uyumsuzluğuna bakmak gerekir. Bir veritabanı sayfası tipik olarak dört ya da sekiz kilobayttır; oysa bir disk, dayanıklılığı yalnızca çok daha

küçük bir birim — geleneksel olarak beş yüz on iki baytlık ya da dört kilobaytlık bir **sektör** — için **atomik** olarak garanti eder. Yani sekiz kilobaytlık bir sayfayı yazmak, aslında arka arkaya birkaç sektör yazmaktır ve elektrik tam ortada kesilirse, ilk birkaç sektör yeni değerle, gerisi eski değerle kalır. Sonuç, ne eski ne yeni olan, melez ve bozuk bir sayfadır.

Bu tehlikeye karşı, yerinde-güncelleyen motorların başvurduğu klasik bir savunma **çift yazmadır** (double-write). Fikir şudur: motor, bir sayfayı asıl yerine yazmadan önce, onun tam bir kopyasını ayrılmış, ayrı bir "çift yazma tamponuna" yazar ve orayı boşaltır; ancak ondan sonra sayfayı asıl yerine yazar. Eğer asıl yere yazma sırasında çökme olur ve sayfa yırtılırsa, kurtarma sırasında sağlam kopya çift yazma tamponunda durduğu için, yırtık sayfa oradan eksiksiz yeniden yazılarak onarılır. Bedeli açıktır: her sayfa diske iki kez yazılır — bir tampona, bir asıl yere — yani bu, dayanıklılık uğruna bilinçle ödenen bir yazma büyütmesidir. (Bir önceki bölümün append-only motorlarının bu derde hiç düşmediğine dikkat edin: onlar var olan veriyi hiç yerinde değiştirmediği için, yarım kalan bir ekleme yalnızca dosyanın en sonundaki son kaydı bozar, eski veriye dokunmaz; bu, az sonra göreceğimiz sağlama toplamıyla kolayca yakalanır.)

Birinci bölümde para transferi örneğini hatırlayın: bir hesaptan düş, diğerine ekle. Bu iki adımın arasında çökme olursa, para buharlaşmıştı. Şimdi sorunun daha da temel bir katmanını görüyoruz: yalnızca iki ayrı adım arasında değil, **tek bir kaydın yazılması sırasında** bile, çökme bizi tutarsız bir duruma düşürebilir. Bir veritabanı, hem "ya hep ya hiç" güvencesini vermek hem de yarım yazmalara karşı kendini korumak zorundadır. Bu iki ihtiyaca birden, şaşırtıcı derecede zarif tek bir fikir yanıt verir.

6.4 Çözüm: önce niyetini yaz

Bu fikrin adı **yazma-öncesi günlük** — İngilizcesiyle write-ahead log, kısaca WAL. Özü tek bir cümlede toplanır: **asıl veriyi değiştirmeden önce, ne yapacağını ayrı bir günlüğe yaz ve o günlüğü dayanıklı kıl.** Yani önce niyetini kaydedersin, sonra eylemi gerçekleştirirsin.

Bunu günlük hayattan bir benzetmeyle düşünelim. Önemli bir işe girişmeden önce, ne yapacağınızı bir deftere not aldığınızı varsayın: "şu hesaptan şu kadar düş, şu hesaba şu kadar ekle." Bu notu aldıktan, yani niyetinizi kalıcı biçimde kaydettikten sonra, işi yapmaya başlarsınız. İşin tam ortasında bayılıp düşerseniz ne olur? Ayıldığınızda defterinizi açar, ne yapmaya niyet ettiğinizi okur ve işi baştan, eksiksiz tamamlarsınız. Defter, niyetinizin kanıtıdır; eylem yarım kalsa bile, defterdeki kayıt sayesinde onu tamamlayabilirsiniz.

WAL tam olarak böyle çalışır. Veritabanı bir değişiklik yapmadan önce, o değişikliğin bir kaydını günlüğe **ekler** ve o kaydı fsync ile dayanıklı kılar. Günlüğe yazmanın güzelliği, önceki bölümde gördüğümüz append-only fikrinin tam da kendisidir: günlük yalnızca sona eklenir, yani diskin sevdiği o hızlı, ardışık yazma biçiminde büyür. Bir değişiklik günlüğe dayanıklı biçimde yazıldığı an, o değişiklik artık **kalıcı kabul edilir** — asıl veri dosyaları henüz güncellenmemiş olsa bile. Çünkü bir çökme olsa dahi, günlükteki kayıt sayesinde o değişikliği yeniden uygulayabiliriz.

İşte WAL'ın asıl zekâsı buradadır. Asıl veri yapısını — ister B-ağacı ister append-only depo olsun — değiştirmek, rastgele yazma içerebilen, yavaş ve riskli bir iştir. WAL, dayanıklılık güvencesini bu yavaş işten **ayırır**: dayanıklılığı hızlı, ardışık günlük yazmasıyla hemen sağlar; asıl veri yapısını güncellemeyi ise sonraya, acele etmeden yapılacak bir işe bırakır. Hız ve güvenlik, böylece aynı anda elde edilir: günlük hızlı ve güvenlidir, asıl güncelleme ise tembel ve huzurludur.[1]

6.4.1 Günlüğe ne yazılır: yineleme, geri-alma ve sıra numarası

WAL'ın bir kaydında tam olarak ne durduğu, kurtarmanın neyi yapabileceğini belirler. İki tür bilgi düşünülebilir. **Yineleme bilgisi** (redo), bir değişikliğin *yeni* hâlini ya da onu yeniden üretecek yeterli bilgiyi taşır; sayesinde, asıl veriye henüz yansımamış bir değişikliği çökmeden sonra yeniden uygulayabiliriz. **Geri-alma bilgisi** (undo), bir değişikliğin *eski* hâlini taşır; sayesinde, tamamlanmamış kalan bir değişikliğin etkisini

[1] J. Gray ve A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1992.

söküp atabiliriz. Tam donanımlı bir günlük sistemi her ikisini de tutar: yineleme, "tamamlandı" denmiş ama diske inmemiş işi bitirmek için; geri-alma, "tamamlandı" denmeden yarıda kalmış işi temizlemek için.

Bu kayıtları birbirine bağlayan kritik bir kavram, her günlük kaydına verilen artan, benzersiz bir numaradır — **günlük sıra numarası** (log sequence number, LSN). LSN, günlükteki olaylara değişmez bir zaman çizgisi kazandırır: hangi değişikliğin hangisinden önce geldiğini kesin biçimde söyler. Daha da önemlisi, asıl veri sayfalarının her birine, o sayfaya en son uygulanan değişikliğin LSN'i damgalanır. Bu damga, az sonra göreceğimiz kurtarmanın kalbinde yatar: bir sayfanın taşıdığı LSN, bir günlük kaydının LSN'inden büyük ya da eşitse, o değişikliğin o sayfaya **zaten uygulandığını** kesin biçimde anlarız ve onu yeniden uygulamaktan kaçınırız. Bu, az sonra kurtarmayı anlatırken adını koyacağımız o kritik özelliğin — bir değişikliği kaç kez tekrarlarsan tekrarla sonucun değişmemesinin — somut mekanizmasıdır.

6.5 Kurtarma: enkazın içinden geri dönmek

WAL'ın asıl sınavı, çökmeden sonra verdiği vaattir. Sistem yeniden açıldığında, veritabanı kendini tutarlı bir duruma getirmek zorundadır ve bunu günlüğü **okuyarak** yapar. Bu sürece **kurtarma** (recovery) denir. Bu kurtarma düzeninin klasik, etkili biçimi, veritabanı literatüründe ARIES yöntemi olarak bilinir.[2]

ARIES, kurtarmayı tek bir geçişte değil, birbirini izleyen **üç fazda** yürütür; bu üç fazı tanımak, sağlam bir kurtarma tasarımının iskeletini görmek demektir.

Şekil 6.2: ARIES kurtarmanın üç fazı: analiz, yineleme, geri-alma.

Birinci faz **analizdir**. Veritabanı, en son güvenli denetim noktasından başlayarak günlüğü ileriye doğru tarar ve çökme anındaki durumun haritasını yeniden çıkarır: çökerken hangi işlemler henüz tamamlanmamıştı (bunlar sonradan geri alınacaktır) ve diske henüz inmemiş olabilecek "kirli" sayfalar hangileriydi. Bu kirli sayfaların listesine **kirli sayfa tablosu** (dirty page table) denir ve önemi şudur: yineleme fazının günlükte tam olarak nereden başlaması gerektiğini, yani en eski yansımamış değişikliğin LSN'ini bu tablo söyler. Analiz fazı tek başına hiçbir şey değiştirmez; yalnızca sonraki iki fazın yol haritasını çizer.

İkinci faz **yinelemedir**. Veritabanı, analiz fazının belirlediği başlangıç noktasından itibaren günlüğü ileriye doğru oynatır ve gördüğü her değişikliği — tamamlanmış işlemlere ait olsun ya da olmasın — asıl veriye yeniden uygular. Bu, sezgiye aykırı görünebilir: neden tamamlanmamış işlemlerin değişikliklerini de uygulayalım? ARIES'in cevabı "tarihi olduğu gibi tekrarla" ilkesidir; önce çökme anındaki tam durumu birebir yeniden kuruyoruz, ardından gelen üçüncü faz tamamlanmamışları temizleyecek. Yinelemenin güvenli olması, az önce anlattığımız LSN damga karşılaştırmasına dayanır: sayfaya zaten yansımış bir değişiklik yeniden

[2] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh ve P. Schwarz, "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging," *ACM TODS* 17(1), 1992.

uygulanmaz. İşte yinelemeyi **etkisiz tekrarlanabilir** (idempotent) kılan budur — aynı kurtarma yarıda kesilip baştan başlasa bile sonuç değişmez.

Üçüncü faz **geri-almadır**. Şimdi sıra, analiz fazında "tamamlanmamış" diye işaretlenen işlemlerin etkilerini söküp atmaya gelir. Veritabanı, günlüğü bu kez **geriye** doğru tarar ve bu işlemlerin yaptığı her değişikliği, günlükteki geri-alma bilgisini kullanarak tersine çevirir. İncelik şudur: geri-almanın kendisi de günlüğe yazılır — buna **telafi kaydı** (compensation log record) denir. Böylece kurtarmanın *ortasında* yeniden bir çökme olursa, hangi geri-almaların zaten yapıldığı yine günlükten bilinir ve geri-alma baştan değil, kaldığı yerden sürdürülür. Bu üç faz tamamlandığında, çökmeden hemen önce "tamamlandı" denmiş her işlem asıl veride eksiksiz durur, tamamlanmamış her işlemin izi silinmiştir ve veritabanı tutarlı bir duruma dönmüştür — sanki hiç çökme olmamış gibi.

6.6 Yarım yazmayı yakalamak: sağlama toplamları

Peki kurtarma, günlüğün kendisinin yarım kalmış bir kaydını nasıl fark eder? Sonuçta çökme, en son günlük kaydının tam yazılmasının ortasında da olabilir. İşte burada **sağlama toplamı** (checksum) devreye girer. Veritabanı, her günlük kaydının yanına, o kaydın içeriğinden hesaplanmış küçük bir doğrulama değeri ekler. Kurtarma sırasında her kaydı okurken, bu değeri yeniden hesaplar ve kayda iliştirilmiş olanla karşılaştırır. İkisi uyuşuyorsa kayıt sağlamdır; uyuşmuyorsa, kayıt çökme sırasında yarım kalmış ya da bozulmuştur. Tipik olarak yalnızca günlüğün **en sonundaki** kayıt böyle bozulabilir — çünkü çökme anına dek öncekiler zaten tamamlanmıştı — ve veritabanı o bozuk son kaydı güvenle atıp, ondan önceki son sağlam noktadan devam eder. Böylece yarım kalmış yazma, sessizce bozuk veriye dönüşmek yerine, açıkça fark edilip temizlenir.

6.7 Günlük sonsuza dek büyüyemez: denetim noktası

WAL'ın bir sorunu vardır: günlük yalnızca eklenerek büyüdüğü için, zamanla devasa olur. Eğer hiç temizlenmezse, hem yer kaplar hem de kurtarma sırasında baştan sona okunması gereken bir dev haline gelir. Çözüm, günlüğü periyodik olarak güvenle kısaltmaktır.

Bu işleme **denetim noktası** (checkpoint) denir. Mantığı şudur: günlükteki değişiklikler eninde sonunda asıl veri dosyalarına yansıtılır. Veritabanı, belirli aralıklarla durup şundan emin olur: "şu ana kadarki tüm günlük kayıtlarının temsil ettiği değişiklikler, artık asıl veriye güvenle işlenmiş ve dayanıklı hale gelmiştir." Bu güvence sağlandıktan sonra, o noktaya kadarki günlük kayıtlarına artık ihtiyaç kalmaz — çünkü kurtarma gerekse bile, onların temsil ettiği veri zaten asıl depoda durmaktadır. İşte o eski kayıtlar artık atılabilir ya da günlük yeniden kullanılabilir. Denetim noktası, günlük ile asıl depo arasındaki **senkronizasyon anıdır**: o ana kadar her şeyin yerli yerinde olduğunu mühürler ve günlüğün baştan büyümesine izin verir.

Burada saf yaklaşımın bir tuzağı vardır ve onu görmek, neden gerçek sistemlerin daha incelikli davrandığını açıklar. En basit denetim noktası, tüm yazma etkinliğini durdurup, bellekteki tüm kirli sayfaları diske boşaltıp, sonra günlüğe "buraya kadar her şey güvende" diye bir işaret koymaktır. Ama bu, veritabanını o boşaltma süresince — saniyeler sürebilir — tümüyle dondurmak demektir; yüksek hacimli bir sistemde kabul edilemez bir duraklamadır. Bunun yerine ARIES, **bulanık denetim noktası** (fuzzy checkpoint) kullanır: denetim noktası anında sistemi durdurmaz; yalnızca o anki kirli sayfa tablosunun ve aktif işlemlerin bir fotoğrafını günlüğe yazar, sonra kirli sayfaları **arka planda, telaşsızca** boşaltmayı sürdürür. Karşılığında, kurtarma yineleme fazına bu bulanık denetim noktasının kaydettiği en eski yansımamış değişiklikten başlar — bu yüzden "buraya kadar her şey kesinlikle diskte" diyemese de, "yinelemenin nereden başlaması gerektiğini biliyorum" diyebilir. Bulanık denetim noktası, çalışma zamanındaki duraklamayı, kurtarma sırasında biraz daha fazla yineleme işi karşılığında ortadan kaldıran bir takastır.

6.8 Dayanıklılığın tayfı: katı, gevşek ve grup commit

Bölümün başında, dayanıklılığın güvenli ama yavaş olduğunu söylemiştik. Gerçek sistemler, bu ödünleşimde tek bir noktaya saplanmaz; bir tayf üzerinde tercih yaparlar.

Bir uçta **katı dayanıklılık** vardır: her değişiklik, "tamamlandı" denmeden önce günlüğe yazılır ve fsync ile dayanıklı kılınır. Bu, en güçlü güvencedir — "tamamlandı" dedikten sonra hiçbir çökme veriyi geri alamaz — ama her değişiklik bir fsync beklediği için en yavaş olandır. Öteki uçta **gevşek dayanıklılık** vardır: değişiklikler günlüğe yazılır ama fsync, her değişiklikte değil, arada bir, toplu olarak yapılır. Bu çok daha hızlıdır; ama bir çökme olursa, henüz fsync edilmemiş son birkaç değişiklik kaybolabilir. Yani gevşek kip, hızı küçük bir veri kaybı riski karşılığında satın alır.

İki ucun arasında zarif bir orta yol vardır: **grup commit** (group commit). Fikri şudur: aynı anda birçok değişiklik "tamamlandı" olmayı bekliyorsa, onları tek tek fsync etmek yerine, hepsini bir araya getirip **tek bir fsync** ile birden dayanıklı kılmak. Bir fsync, ister bir değişikliği ister yüz değişikliği boşaltsın, kabaca aynı süreyi alır — çünkü maliyetin asıl kaynağı, boşaltılan veri miktarı değil, diskten "fiilen işledim" onayını bekleme gecikmesidir; bu yüzden yüz değişikliği tek fsync'e toplamak, gerçek dayanıklılık güvencesinden hiç ödün vermeden verimi kat kat artırır. Bunu, postaneye her mektup için ayrı ayrı koşmak yerine, eldeki tüm mektupları toplayıp tek seferde götürmeye benzetebilirsiniz.

Şekil 6.3: Grup commit: bekleyen birçok commit tek fsync'e toplanır.

Grup commit'in ince bedeli, bir gecikme takasıdır: bir grubu oluşturabilmek için, sistem ilk gelen commit'i hemen boşaltmak yerine, kısa bir süre bekleyip yanına birkaç commit daha toplanmasını umar. Bu küçük bekleme, tek bir işlemin gecikmesini azıcık artırır; ama sistem genelindeki **iş hacmini** (throughput) çarpıcı biçimde yükseltir. Yük arttıkça grup commit kendiliğinden daha da verimli olur — çünkü ne kadar çok commit aynı anda bekliyorsa, tek fsync'e o kadar çoğu sığar. Üçüncü kısımda OxiDB'nin varsayılan olarak katı dayanıklılığı seçtiğini — her commit'i fsync ettiğini — ama toplu eklemelerde bu maliyeti tek bir fsync'e amortize ettiğini ve istendiğinde gevşek kipe geçilebildiğini göreceğiz.

6.9 WAL nereye oturur

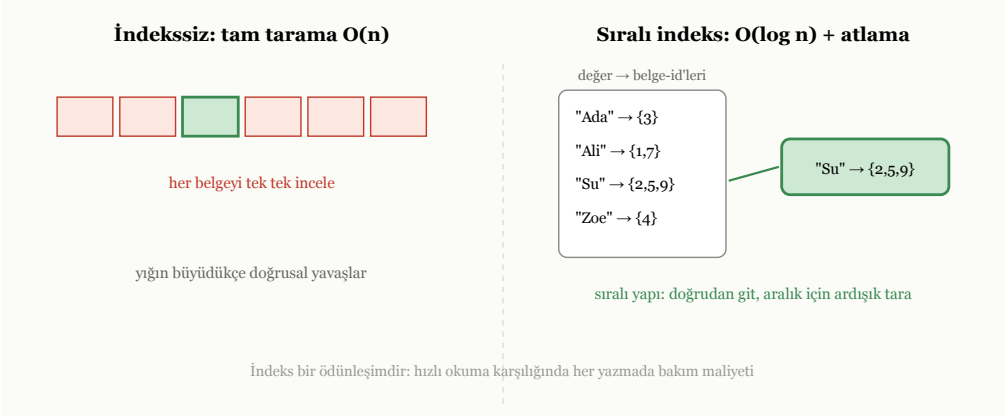
Bu bölümü, önceki bölümle bağını netleştirerek toparlayalım. WAL, beşinci bölümdeki depolama felsefelerinin bir alternatifi değildir; onların **önünde** duran, dayanıklılıktan ve atomiklikten sorumlu ayrı bir katmandır. İster veriyi sayfa tabanlı bir B-ağacında yerinde güncelleyen bir motor olsun, ister append-only bir depo olsun, ikisi de dayanıklılık ve çökme güvenliği için bir WAL'a yaslanabilir. Asıl veri yapısı "nasıl saklayacağım" sorusunu yanıtlar; WAL ise "değiştirirken çökersem ne olacak" sorusunu yanıtlar. İkisi birbirini tamamlar.

Böylece, beşinci ve altıncı bölümler birlikte, bir belge veritabanının en alt katmanını — veriyi diske güvenli biçimde yazma ve çökmeden geri dönme yeteneğini — tamamlamış oldu. Artık verimiz hem kalıcı hem de tutarlı biçimde diskte duruyor. Ama elimizde, çözülmemiş büyük bir sorun daha var. Birinci bölümde saymıştık: bir milyon belge arasından aradığımız tek kaydı, diski baştan sona taramadan nasıl buluruz? Veriyi güvenle saklamayı öğrendik; şimdi onu hızla **bulmayı** öğrenmemiz gerekiyor. Bir sonraki bölümde, veritabanlarını taramaya mahkûm olmaktan kurtaran o yardımcı yapılara — indekslere — eğiliyoruz.

Bölüm 7

İndeksleme: B-Ağacı İndeksleri, Bileşik İndeksler ve Ters İndeks

Önceki iki bölümde veriyi diske güvenle yazmayı ve çökmeden geri dönmeyi öğrendik. Artık verimiz kalıcı ve tutarlı biçimde duruyor. Ama birinci bölümde saydığımız sorunlardan en görünür olanı hâlâ çözülmedi: bir milyon belge arasından aradığımız tek kaydı, diski baştan sona taramadan nasıl buluruz? Bu bölüm, veritabanlarını taramaya mahkûm olmaktan kurtaran o yardımcı yapılara — indekslere — eğiliyor. İndeksler, bir veritabanının hızının büyük bölümünün geldiği yerdir; onlarsız her soru, tüm veriyi okumayı gerektirirdi.



Şekil 7.1: Tam tarama ile sıralı indeks erişiminin karşılaştırması.

7.1 Tarama sorunu

Bir koleksiyonda bir milyon kullanıcı belgesi olduğunu ve belirli bir e-posta adresine sahip olanı aradığımızı düşünelim. Hiçbir yardımcı yapı yoksa, tek seçeneğimiz belgeleri tek tek okuyup e-posta alanına bakmaktır. Aradığımız belge şanslıysa başlarda, şanssızsak en sonda; ortalama olarak yarım milyon belgeyi okumamız gerekir. Buna **tam tarama** (full scan) denir ve maliyeti veriyle birlikte büyür: veri iki katına çıkınca arama da iki kat yavaşlar. Üstelik aynı aramayı her tekrarladığınızda baştan tararsınız; sistem hiçbir şey öğrenmez.

Bu, kabul edilemez bir durumdur. Bir milyon, hatta bir milyar kayıt arasından aradığımızı, neredeyse anında bulabilmeliyiz. Bunun için, verinin yanı sıra, aramayı hızlandıran ayrı bir yapı tutmamız gerekir.

7.2 İndeks nedir: kitabın arkasındaki dizin

İndeksin ne olduğunu anlamak için, elinizdeki bu kitabın arkasındaki dizini düşünün. "fsync" kavramının nerede geçtiğini bulmak istediğinizde, kitabı baştan sona okumazsınız; arkadaki dizine bakar, "fsync" sözcüğünün yanında yazan sayfa numarasını görür ve doğrudan o sayfaya gidersiniz. Dizin, kitabın içeriğinin bir kopyası değildir; ondan **türetilmiş**, yalnızca "hangi kavram hangi sayfada" bilgisini sıralı biçimde tutan, küçük ve aranabilir bir yapıdır.

Bir veritabanı indeksi tam olarak budur: belirli bir alanın değerlerinden, o değere sahip belgelerin **konumuna** giden bir eşleme. E-posta alanı için bir indeksimiz varsa, aradığımız e-posta adresini bu indekste buluruz ve indeks bize o adrese sahip belgenin nerede olduğunu doğrudan söyler. Bir milyon belgeyi taramak yerine, küçük ve düzenli indeks yapısında birkaç adımda sonuca ulaşırız. İndeks, asıl veriden türetilmiştir — yani onda yeni bir bilgi yoktur, veri zaten belgelerin içindedir — ama o bilgiyi, aramaya uygun bir düzende yeniden düzenleyerek, bulmayı hızlandırır.

7.3 İndeksin bedeli: bedava hız yoktur

İndeksler sihirli görünür, ama bedelsiz değildir ve bu bedeli görmek, onları doğru kullanmanın ön koşuludur. İki maliyet vardır.

Birincisi **yazma maliyetidir**. İndeks, asıl veriden türetildiği için, veri her değiştiğinde indeksin de güncellenmesi gerekir. Yeni bir belge eklediğinizde, yalnızca belgeyi yazmakla kalmaz, o belgenin indekslenen her alanı için ilgili indekse de bir giriş eklersiniz. Bir belgeyi sildiğinizde ya da indekslenen bir alanını değiştirdiğinizde, indeksleri de buna göre düzeltmeniz gerekir. Dolayısıyla her indeks, okumayı hızlandırırken yazmayı bir miktar yavaşlatır. Bir koleksiyonda ne kadar çok indeks varsa, her yazma o kadar çok ek iş doğurur.

İkincisi **yer maliyetidir**: her indeks, diskte ek yer kaplar.

Bu maliyetler, basit ama önemli bir ilkeye götürür: **her alanı indekslemezsiniz; yalnızca üzerinde sık arama yaptığınız alanları indekslersiniz.** Hangi alanları indeksleyeceğiniz, üçüncü bölümdeki o değişmez ilkeye geri döner — erişim örüntülerinize bağlıdır. Hangi sorguları sık soruyorsanız, o sorguların süzdüğü alanları indekslersiniz. "Her ihtimale karşı her şeyi indeksleyelim" yaklaşımı, yazmaları boğan ve yer israf eden yaygın bir hatadır.

7.4 Sıralı indeksler: hem eşitlik hem aralık

İndeksin *ne* olduğunu anladık; peki *nasıl* yapılır? En yaygın ve en güçlü indeks türü, değerleri **sıralı** tutan yapılardır — beşinci bölümde tanıdığımız B-ağacının indeks olarak kullanılan biçimi. Burada B-ağacı, asıl veriyi değil, **indekslenen alanın değerlerini** sıralı biçimde tutar; her değerin yanında, o değere sahip belgelerin konumu durur.

Değerleri sıralı tutmanın üç ayrı kazancı vardır ve bu, sıralı indeksleri bu kadar değerli kılan şeydir. Birincisi **eşitlik aramasıdır**: belirli bir değere sahip kayıtları, sıralı yapıda hızlıca buluruz. İkincisi **aralık aramasıdır**: "şu değerle bu değer arasındaki tüm kayıtlar" sorusunu, sıralı yapıda o aralığın başına gidip yan yana ilerleyerek yanıtlarız — değerler zaten sıralı olduğu için bu doğaldır. Üçüncüsü ve çoğu zaman gözden kaçanı, **sıralı getirmedir**: bir sorgu sonuçları belirli bir alana göre sıralı isterse ve o alanın sıralı bir indeksi varsa, sıralama işini ayrıca yapmaya gerek kalmaz — indeks zaten o sırayı tutmaktadır. "Şu alana göre sıralı ilk on kayıt" gibi bir istek, tüm veriyi sıralamak yerine, indeksin başından on adım yürüyerek yanıtlanabilir. Üçüncü kısımda OxiDB'nin sıralamayı tam da böyle, indeks üzerinden, taramadan yaptığını göreceğiz.

Bir önceki bölümde B+ağacının iç yapısını gördük; bir indeks olarak kullanıldığında o yapı neredeyse aynı kalır, yalnızca yapraklarda asıl belge yerine **anahtar artı belge konumu** çiftleri durur. Yüksek fanout sayesinde indeks ağacı da sığ kalır: milyonlarca farklı değer arasında bile, aranan değere ya da bir aralığın

başına üç-dört düğüm erişimiyle ulaşılır. Yaprakların bağlı liste ile zincirlenmesi, "şu değerden büyük ilk yüz kayıt" ya da "şu iki tarih arası tüm kayıtlar" gibi sorguları, ağaca tekrar tepeden inmeden, yapraktan yaprağa yürüyerek yanıtlamayı sağlar. Aynı yapı, "sıralı getirme"nin bedava gelmesinin de nedenidir: indeks zaten sıralı olduğu için, sıralı sonuç istemek ek bir sıralama işi doğurmaz.

Sıralı indeksin bir alternatifi, değerleri sıralı tutmayan ama eşitlik aramasını çok hızlı yapan **karma (hash) indekstir**. Karma indeks, bir değeri bir karma işleviyle doğrudan bir konuma eşler; eşitlik aramasında ortalama sabit zamanda — ağaçtaki gibi birkaç düğüm inişi bile gerekmeden — sonuca varır. Bedeli, sıranın tümüyle kaybolmasıdır: karma, yakın değerleri diskte yakın yerlere koymaz, tam tersine bilerek dağıtır; bu yüzden aralık sorgusunu ya da sıralı getirmeyi hiç yapamaz. Yalnızca tam eşleşme aradığınız ve aralık ya da sıralama gerekmediği durumlarda yeterlidir. Sıralı indeksler ise daha genel amaçlıdır; bu yüzden veritabanlarının çoğunlukla yaslandığı yapı onlardır.

Sıralı indeksi gerçeklemenin ağaca rakip, zarif bir başka yolu daha vardır: **atlama listesi** (skip list).[1] Atlama listesi, sıralı bir bağlı liste üzerine, rastgele yükseklikte "ekspres şeritler" kuran bir yapıdır. En alt şerit tüm öğeleri sırayla içerir; her üst şerit, alttakinin öğelerinden yalnızca bir kısmını — yazı-tura atar gibi olasılıkla seçilmiş bir alt kümesini — atlamalı biçimde tutar. Bir değer ararken en üst, en seyrek şeritten başlar, mümkün olduğunca uzağa zıplar, gerektiğinde bir alt şeride iner ve böylece hedefe ağaçtakine benzer logaritmik adımda ulaşırsınız. Atlama listesinin çekiciliği, dengeli bir ağacın bölme-birleştirme dansının karmaşık kilitleme mantığını gerektirmemesi, buna karşın benzer arama başarımını olasılıksal olarak sunmasıdır; bu yüzden eşzamanlı, bellek-içi sıralı yapılarda — örneğin bir önceki bölümün LSM memtable'ında — sık tercih edilir.

7.5 Seçicilik: her indeks aynı ölçüde işe yaramaz

Bir indeksin ne kadar işe yaradığı, indekslenen alanın **seçiciliğine** bağlıdır. Seçicilik, bir değerin kaç belgeyle eşleştiğiyle ilgilidir. Bir alanın her değeri yalnızca birkaç belgeyle eşleşiyorsa — örneğin e-posta adresi, ki neredeyse her biri benzersizdir — indeks muhteşem çalışır: aradığınız değeri bulur ve sizi doğrudan o birkaç belgeye götürür.

Ama bir alanın yalnızca birkaç farklı değeri varsa, indeksin yararı azalır. Bir "aktif mi" alanını düşünün: değeri ya doğru ya yanlıştır. Bu alanı indekslerseniz, "aktif olanları getir" sorgusu, belki belgelerin yarısını seçer; yarım milyon belgeyi getirmek için indeksten geçmek, neredeyse tüm veriyi taramaktan pek de hızlı değildir.

Bu sezginin altında somut bir maliyet vardır ve onu görmek önemlidir. Bir indeks, size eşleşen belgelerin konumlarını verir; ama o konumlar diskte dağınıksa, her birine gitmek ayrı bir rastgele okuma demektir. Eşleşen belge sayısı arttıkça, bu dağınık okumaların toplamı, dosyayı baştan sona ardışık okumaktan — ki disk ardışık okumayı çok daha iyi yapar — daha pahalı hale gelebilir. İşte bu yüzden, sorgunun verinin yaklaşık yüzde beş-onundan fazlasını seçtiği noktada, sorgu işleyiciler çoğu zaman indeksi bilerek bırakıp tam taramayı seçer; çünkü ardışık bir tarama, binlerce dağınık atlamadan ucuzdur. Bu eşiğe ne kadar yaklaşıldığı, tam da alanın seçiciliğine bağlıdır. Düşük seçicilikli alanları indekslemek, çoğu zaman bedelini hak etmez. İyi bir indeks, çok sayıda belge arasından **azını** ayıklayan alanlar üzerine kurulur.

7.6 Bileşik indeksler: birden çok alan birlikte

Sorgular çoğu zaman tek bir alanı değil, birkaç alanı birden süzer: "şu bölgedeki, şu yaş aralığındaki, şu durumdaki kullanıcılar." Bunun için **bileşik indeks** (composite index) kullanılır: tek bir alan yerine, birkaç alanın birleşimi üzerine kurulu bir indeks.

Bileşik indeksin inceliği, alanların **sırasının** önemli olmasıdır. Bir telefon rehberini düşünün: kayıtlar önce soyada, aynı soyad içinde ada göre sıralanmıştır. Bu rehber, "soyadı Yılmaz olanlar" ya da "soyadı Yılmaz, adı

[1] W. Pugh, "Skip Lists: A Probabilistic Alternative to Balanced Trees," *Communications of the ACM* 33(6), 1990.

Ali olanlar" sorularını kolayca yanıtlar; çünkü sıralama önce soyada göredir. Ama "adı Ali olanlar" sorusunu — soyadından bağımsız olarak — kolayca yanıtlayamaz, çünkü Ali'ler tüm rehbere dağılmıştır. Bileşik indeks de tıpkı böyledir: (bölge, yaş) üzerine kurulu bir indeks, yalnızca bölgeye göre ya da bölge artı yaşa göre aramayı hızlandırır; ama yalnızca yaşa göre aramaya pek yaramaz. Buna **önek kuralı** denir: bileşik indeks, alan sırasının baştan başlayan bir önekini kullanan sorgulara yarar.

Şekil 7.2: Bileşik indeks (bölge, yaş) ve önek kuralının işleyişi.

Önek kuralının pratik bir uzantısı, **eşitlik-sonra-aralık** dizilimidir. Bileşik indeks, önekteki alanlar üzerinde **eşitlik** koşulu olduğu sürece, sonraki alanlardaki **aralık** koşullarını da verimli süzebilir; ama bir alanda aralık koşuluna girer girmez, ondan sonraki alanlar artık sıralı dilim içinde dağıldığından, indeks onları aynı verimle süzemez. Bu yüzden iyi bir kural, eşitlikle sınanan alanları indeksin başına, aralıkla sınananı sona koymaktır. Bu yüzden bileşik indekste alanları hangi sırayla dizeceğiniz, hangi sorguları hızlandıracağınızı belirleyen önemli bir tasarım kararıdır.

7.7 Kapsayan indeksler: belgeye hiç dokunmamak

İndekslerin zarif bir gücü daha vardır. Normalde bir indeks size belgenin **konumunu** verir; sonra o konuma gidip belgeyi okursunuz. Ama bazen, sorgunun ihtiyaç duyduğu tüm bilgi zaten indeksin içinde durur. O zaman belgeye hiç gitmeye gerek kalmaz; yanıt, yalnızca indeksten üretilir.

En arı örneği saymadır. "Şu bölgede kaç kullanıcı var?" sorusunu düşünün. Eğer bölge alanının bir indeksi varsa, o indekste her bölgenin altında hangi belgelerin olduğu zaten yazılıdır; saymak için yalnızca o girişleri saymak yeterlidir, belgelerin hiçbirini açmaya gerek yoktur. Sorgunun ihtiyaç duyduğu her şeyi indeksin tek başına sağlamasına **kapsama** (covering) denir ve bir sorgunun verilebilecek en hızlı yanıtlarından birini doğurur: hiç belge okumadan, yalnızca indeksten. Üçüncü kısımda OxiDB'nin saymayı tam da böyle, belgelere hiç dokunmadan yaptığını göreceğiz.

İndekslerin bir yan faydasını da burada anmak gerekir: **benzersizlik güvencesi**. Bir alanı "benzersiz indeks" olarak işaretlerseniz, indeks o alanda aynı değerin iki kez yazılmasını engeller. Böylece indeks yalnızca aramayı hızlandırmakla kalmaz, bir veri bütünlüğü kuralını da dayatmış olur.

7.8 Kısmi ve seyrek indeksler: yalnızca işe yarayanı indeksle

İndeksin yazma ve yer maliyetini hatırlayalım: her indeks, indekslenen her belge için bir giriş tutar ve her yazmada güncellenmek zorundadır. Peki bir koleksiyondaki belgelerin yalnızca küçük bir bölümü bir sorgu için anlamlıysa, neden hepsini indeksleyelim? Bu gözlem, iki akraba inceltmeye yol açar.

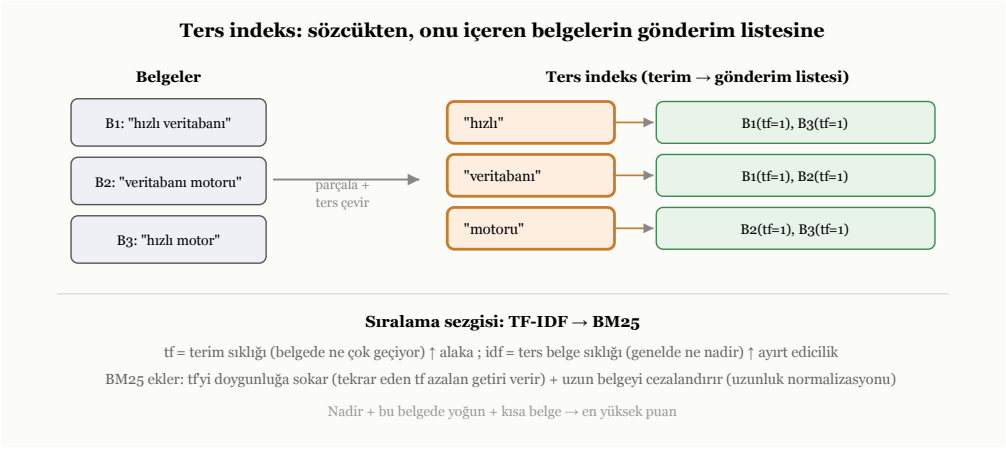
Belge veritabanlarının esnek şemasından doğan ilki **seyrek indekstir** (sparse index). Belgeler birbirinden farklı alanlara sahip olabildiği için, bir alan belgelerin yalnızca bir kısmında bulunabilir — örneğin yalnızca premium kullanıcıların bir "abonelik bitiş tarihi" alanı vardır. Seyrek indeks, o alanı **içermeyen** belgeleri indekse hiç koymaz; yalnızca alanın gerçekten var olduğu belgeler için giriş tutar. Böylece indeks, ilgisiz milyonlarca belgenin "yok" girişiyle şişmez; hem küçük kalır hem de o alanı içeren azınlığı sorgularken keskin olur.

İkincisi, daha genel olan **kısmi indekstir** (partial index): yalnızca belirli bir **koşulu** sağlayan belgeleri indeksler. Örneğin yalnızca "durum = açık" olan siparişleri indekslemek; kapanmış, arşivlenmiş siparişler — ki çoğunluk onlardır ve onlar üzerinde nadiren arama yaparsınız — indekse hiç girmez. Kısmi indeksin kazancı çift yönlüdür: indeks çok daha küçük olduğu için hem yer kazanırsınız hem de bu küçük indekse yazmak daha ucuz olur; üstelik koşulu sağlamayan belgelerin yazılması indeksi hiç meşgul etmez. Karşılığında, kısmi indeks yalnızca koşuluyla örtüşen sorguları hızlandırır; "kapalı siparişleri ara" derseniz bu indeks işe yaramaz. Hem seyrek hem kısmi indeks, aynı sağduyunun farklı yüzleridir: indeksi yalnızca onu gerçekten kullanacak sorguların kapsadığı veriyle sınırlamak.

7.9 Ters indeks: metni aranabilir kılmak

Şimdiye dek anlattığımız indeksler, bir alanın **tam değeriyle** çalışır: e-posta adresi şuna eşit, yaş şu aralıkta. Ama bambaşka bir arama türü vardır: metnin **içinde** sözcük aramak. "İçinde 'veritabanı' sözcüğü geçen tüm belgeleri getir" demek istediğinizde, alanın tam değerine bakan indeksler işe yaramaz; çünkü siz tam eşleşme değil, metnin içinde geçen bir sözcüğü arıyorsunuz. Bu ihtiyaç için tasarlanmış, tümüyle farklı bir yapı vardır: **ters indeks** (inverted index).

Ters indeksin fikri, sıradan indeksin tersini yapmaktır. Sıradan indeks "belgeden, içindeki değere" giden ilişkiyi tutarken, ters indeks "sözcükten, o sözcüğü içeren belgelere" giden ilişkiyi tutar. Bir kitabın arkasındaki dizine yine dönelim; orada her kavramın yanında, o kavramın geçtiği sayfaların listesi vardır. Ters indeks tam olarak budur, ama metindeki her anlamlı sözcük için. Bu yapıda her sözcüğe — daha doğrusu her **terime** — karşılık gelen, o terimi içeren belgelerin listesine **gönderim listesi** (posting list) denir. Gönderim listesi yalnızca belge kimliklerini değil, çoğu zaman terimin o belgede kaç kez geçtiğini ve hangi konumlarda durduğunu da tutar; bu ek bilgi, hem birazdan göreceğimiz alaka puanlamasını hem de "şu iki sözcük yan yana geçsin" gibi öbek aramalarını mümkün kılar.



Şekil 7.3: Ters indeks: terimden gönderim listesine, ve alaka puanlaması.

Bir belge eklendiğinde, metni anlamlı sözcüklere ayrılır — buna **parçalama** (tokenization) denir — ve genellikle bir dizi normalleştirme adımından geçer: büyük-küçük harf birleştirilir, çekim ekleri budanarak sözcükler köklerine indirgenebilir, "ve", "bir", "ile" gibi her belgede geçen ve ayırt edici değeri olmayan **durak sözcükler** (stop words) atılabilir. Bu işlenmiş terimlerin her birinin gönderim listesine o belge eklenir. Bir

sözcüğü aradığınızda, ters indeks size o sözcüğün gönderim listesini doğrudan verir; birden çok sözcüklü bir sorguda ise, ilgili gönderim listeleri kesiştirilerek (hepsini içeren belgeler) ya da birleştirilerek (herhangi birini içeren belgeler) sonuç kümesi üretilir — hiçbir metni baştan sona taramadan.

Metin aramanın sıradan aramadan bir farkı daha vardır: **alaka düzeyi**. Bir sözcüğü içeren yüzlerce belge olabilir, ama hepsi aynı ölçüde ilgili değildir; bu yüzden metin arama, sonuçları yalnızca bulmakla kalmaz, en alakalıdan en alakasıza **sıralar** da. Bu sıralamanın klasik sezgisi iki ölçütü birleştirir. Birincisi **terim sıklığıdır** (term frequency): bir sözcük bir belgede ne kadar çok geçiyorsa, o belge o sözcükle muhtemelen o kadar ilgilidir. İkincisi **ters belge sıklığıdır** (inverse document frequency): bir sözcük tüm koleksiyonda ne kadar *nadir* geçiyorsa, geçtiği yerde o kadar ayırt edicidir — "veritabanı" gibi nadir bir terim, "için" gibi her yerde geçen bir terimden çok daha fazla bilgi taşır. Bu ikisinin çarpımı, terim sıklığı-ters belge sıklığı (TF-IDF) puanlamasının özüdür: nadir bir terimin yoğun geçtiği belge en yükseğe çıkar.

Bu sezginin olgunlaşmış, olasılıksal akrabası **BM25** olarak bilinir ve iki ince düzeltme getirir. Birincisi **doygunluktur** (saturation): terim sıklığının katkısı sonsuza dek artmaz; bir sözcük belgede beş kez yerine elli kez geçtiğinde, bu ondan elli kat daha alakalı sayılmaz — katkı bir tavana doğru yumuşakça doyar. İkincisi **uzunluk normalizasyonudur** (length normalization): uzun bir belgede bir sözcüğün geçmesi, kısa bir belgede geçmesinden daha az şey ifade eder, çünkü uzun belge zaten her sözcüğü barındırma eğilimindedir; BM25, belge uzunluğunu koleksiyon ortalamasıyla kıyaslayarak uzun belgeleri hafifçe cezalandırır. Bu iki düzeltme, çıplak TF-IDF'in fazla bağırdığı durumları yumuşatır ve pratikte daha isabetli bir sıralama üretir. Bu konular bilgi erişimi alanının temel taşlarıdır.[2] Üçüncü kısımda OxiDB'nin tam metin aramayı, böyle bir ters indeks ve alaka puanlaması üzerine kurduğunu göreceğiz.

7.10 İndeksler de dayanıklı olmalı

Son bir bağlantı kuralım. İndeksler asıl veriden türetilmiş olsa da, her sorgudan önce baştan kurulamayacak kadar pahalıdır; bu yüzden onların da kalıcı olması gerekir. İndeksler de, asıl veri gibi, diske yazılır ve bir önceki bölümde gördüğümüz dayanıklılık kaygılarının kapsamına girer. Bir çökme sonrası, indekslerin de asıl veriyle tutarlı bir duruma getirilmesi gerekir; kimi sistemler indeksleri günlükten kurtarır, kimileri ise gerekirse asıl veriyi tarayarak yeniden kurar. İndeks ile veri arasındaki bu tutarlılığın korunması, sağlam bir veritabanının sessiz ama kritik bir görevidir.

7.11 Bu bölümün bıraktığı yer

Bu bölümde, veritabanlarını taramaya mahkûm olmaktan kurtaran indeksleri tanıdık. Bir indeksin, asıl veriden türetilmiş, aramayı hızlandıran ayrı bir yapı olduğunu; bunun karşılığında yazmayı yavaşlatıp yer kapladığını; bu yüzden seçici biçimde, erişim örüntülerine göre kurulduğunu gördük. Sıralı indekslerin — ister B+ağacı ister atlama listesi biçiminde olsun — eşitlik, aralık ve sıralı getirmeyi birden desteklediğini; karma indekslerin yalnızca eşitlikte hızlı olduğunu; seçiciliğin bir indeksin yararını belirlediğini ve bir eşik aşıldığında tam taramanın indeksten ucuza gelebildiğini; bileşik indekslerin önek kuralıyla, eşitlik-sonra-aralık dizilimiyle birden çok koşulu hızlandırdığını; kapsayan indekslerin belgeye hiç dokunmadan yanıt üretebildiğini; kısmi ve seyrek indekslerin indeksi yalnızca işe yarar veriyle sınırlayarak maliyeti düşürdüğünü; ve ters indekslerin, gönderim listeleri ve TF-IDF'ten BM25'e uzanan alaka puanlamasıyla metni sözcük düzeyinde aranabilir kıldığını öğrendik.

Ama bir indeks, tek başına, yalnızca bir araçtır. Bir kullanıcının sorusunu — "şu bölgedeki, şu yaş aralığındaki, adında şu geçen kullanıcıları, şuna göre sıralı getir" — alıp, bu soruyu hangi indekslerin işe yarayacağına karar veren, veriyi en az iş yaparak süzecek bir plana dönüştüren ayrı bir akıl gerekir. O akıl, sorgu işleyicidir. Bir

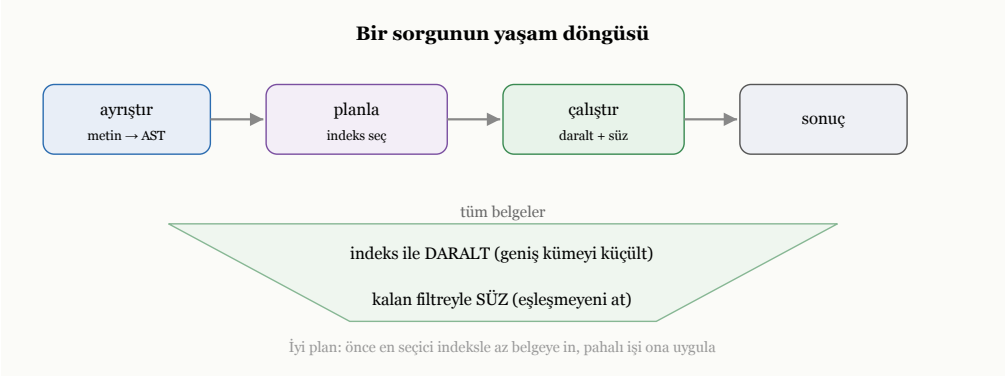
[2] C. D. Manning, P. Raghavan ve H. Schütze, *Introduction to Information Retrieval*, Cambridge University Press, 2008; S. Robertson ve H. Zaragoza, "The Probabilistic Relevance Framework: BM25 and Beyond," *Foundations and Trends in Information Retrieval* 3(4), 2009.

sonraki bölümde, bir sorunun bir yanıta nasıl dönüştüğünü — ayrıştırmadan planlamaya, indeks seçiminden yürütmeye — adım adım izleyeceğiz.

Bölüm 8

Sorgu İşleme: Ayrıştırma, Planlama ve İndeks Seçimi

Önceki bölümde indeksleri tanıdık ve bir indeksin, tek başına, yalnızca bir araç olduğunu söyledik. Bir kullanıcının sorusunu alıp, hangi indeksin işe yarayacağına karar veren, veriyi en az iş yaparak süzecek bir plana dönüştüren ayrı bir akıl gerekir. O akıl, **sorgu işleyicidir** ve bu bölümün konusu, bir sorunun bir yanıta nasıl dönüştüğüdür. İkinci bölümde, ilişkisel modelin büyük armağanının sorguları "bildirimsel" kılmak olduğunu — yani yalnızca *ne* istediğinizi söyleyip *nasıl* sorusunu sisteme bırakmak olduğunu — söylemiştik. İşte o "nasıl" sorusunu yanıtlayan bileşen, sorgu işleyicidir; ve onun bu işi ne kadar iyi yaptığı, bir veritabanının hızını belirleyen en kritik etkenlerden biridir.



Şekil 8.1: Bir sorgunun yaşam döngüsü: ayrıştır, planla, çalıştır.

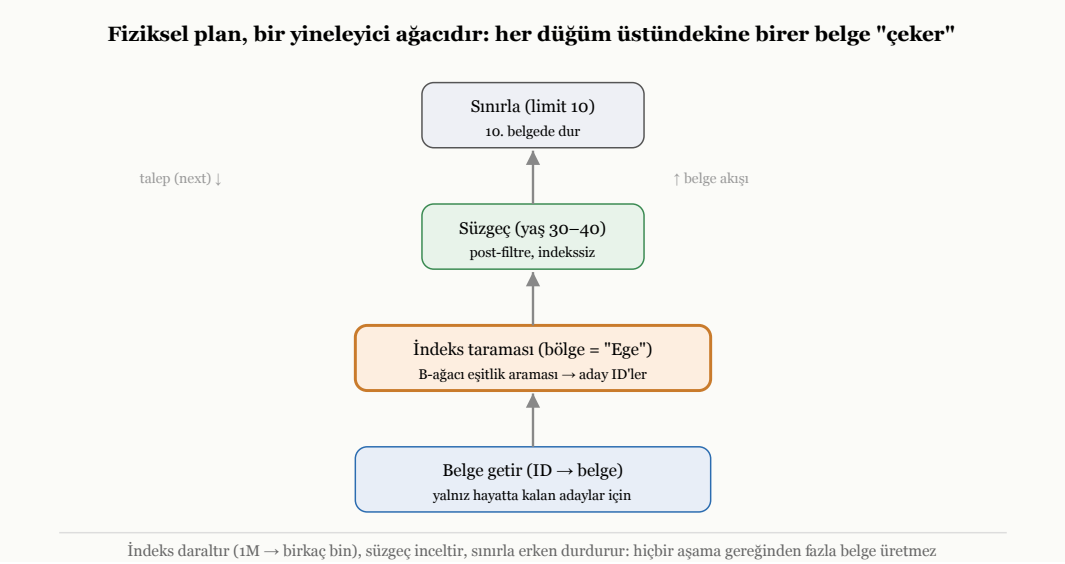
8.1 Soru ile yol arasındaki fark

Önce bir ayrımı net kuralım. Bir **sorgu**, bir sorudur: "şu bölgedeki, şu yaş aralığındaki, adında şu geçen kullanıcıları, şuna göre sıralı getir." Bu soru, *neyi* istediğinizi söyler, ama o sonuca *nasıl* ulaşılacağını söylemez. Sonuca ulaşmanın ise birçok yolu olabilir. Önce bölgeye göre indeksi kullanıp adayları daraltıp sonra yaşa bakabilirsiniz; ya da önce yaşa göre indeksten başlayabilir; ya da hiç indeks kullanmadan tüm belgeleri tarayıp koşulları tek tek deneyebilirsiniz. Bu yolların her biri aynı yanıtı verir, ama maliyetleri taban tabana farklı olabilir — biri saniyenin binde biri, diğeri dakikalar sürebilir.

Sorgu işleyicinin görevi, soruyu bir **yola**, yani bir yürütme planına dönüştürmektir. Bunu bir yol tarifi uygulamasına benzetebilirsiniz: siz yalnızca varış noktanızı söylersiniz; uygulama, olası güzergâhları değerlendirip

en hızlısını seçer ve sizi adım adım yönlendirir. Sorgu işleyici de tıpkı böyle, olası yürütme yollarını tartar ve en ucuzunu seçer. Bu işi üç aşamada yapar: ayrıştırma, planlama ve yürütme.

Bu üç aşamanın altında, profesyonel veritabanı motorlarının onlarca yıldır biriktirdiği daha ince bir katmanlama yatar. Ham istek önce bir **soyut sözdizimi ağacına** (abstract syntax tree, AST) dönüşür — sorgunun saf yapısal iskeleti. Bu iskelet, ardından bir **mantıksal plana** (logical plan) çevrilir: hangi işlemlerin hangi sırayla yapılacağını, ama *nasıl* yapılacağını henüz söylemeyen, cebirsel bir akış. Mantıksal plan, "şu koşulla süz, sonra şuna göre sırala, sonra ilk onu al" der; ama süzmenin indeksle mi taramayla mı, sıralamanın bellekte mi indeksle mi yapılacağını belirtmez. Son adımda mantıksal plan bir ya da daha çok **fiziksel plana** (physical plan) somutlaşır: her mantıksal işlem, onu gerçekten yürütecek somut bir algoritmaya — indeks taraması, tam tarama, bellekte sıralama, şu birleştirme algoritması — bağlanır. Eniyileyicinin asıl işi, aynı mantıksal planı karşılayan birçok olası fiziksel plan arasından en ucuzunu seçmektir. Bu katmanlı ayrım — soru, cebirsel akış, somut algoritma — modern eniyileyicinin omurgasıdır ve ilk olarak IBM'in System R araştırma sisteminde, bugün hâlâ kullandığımız biçimiyle ortaya konmuştur.[1]



Şekil 8.2: Fiziksel plan, bir yineleyici ağacıdır.

8.2 Birinci aşama: ayrıştırma

İlk adım **ayrıştırmadır** (parsing). Kullanıcının sorgusu, sisteme bir biçimde — bir metin ya da yapılandırılmış bir istek olarak — gelir. Sorgu işleyici, bu ham isteği önce kendi anlayabileceği iç bir temsile çevirir: koşulların ve mantıksal bağların bir ağacına. Örneğin "bölge eşittir Ege VE yaş 30 ile 40 arasında" sorgusu, bir "VE" düğümünün altında iki koşulun durduğu küçük bir ağaca dönüşür.

Belge veritabanlarında bu koşullar zengin bir çeşitlilik taşır. En temeli **eşitliktir**: bir alanın belirli bir değere eşit olması. Ardından **karşılaştırmalar** gelir: bir alanın bir değerden büyük, küçük ya da bir aralıkta olması. **Üyelik** koşulları, bir alanın belirli değerler kümesinden birine eşit olup olmadığını sorar. **Mantıksal bağlar** — ve, veya, değil — koşulları birleştirir. Belgeler iç içe ve listeli olabildiği için, belgeye özgü koşullar da vardır: bir alanın **var olup olmadığı**, bir listenin belirli bir öğeyi **içerip içermediği**, bir metnin bir kalıba **uyup uymadığı**. Ayrıştırma aşaması, tüm bu koşulları tanır, geçerliliğini denetler ve onları işlenebilir bir yapıya kavuşturur. Bu aşamanın sonunda, sorgu artık bir metin değil, sistemin üzerinde akıl yürütebileceği yapısal bir nesnedir.

[1] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie ve T. Price, "Access Path Selection in a Relational Database Management System," *Proc. ACM SIGMOD*, 1979.

8.3 İkinci aşama: planlama

Asıl zekâ ikinci aşamada, **planlamada** belirir. Artık sorgunun yapısını bilen sistem, onu nasıl yürüteceğine karar vermelidir. Buradaki temel karar, yedinci bölümde tanıdığımız araçla doğrudan ilgilidir: **hangi indeksi kullanmalı, ya da hiç indeks kullanmamalı mı?**

Mantık şöyle işler. Sorgudaki her koşula bakılır ve o koşulun süzdüğü alanın bir indeksi olup olmadığı kontrol edilir. Eğer bir koşul, indeksli ve seçici bir alan üzerindeyse — örneğin e-posta adresi gibi — bu koşul bir **fırsattır**: indeksi kullanarak, tüm belgeleri taramadan, yalnızca o koşula uyan az sayıda **aday** belgeyi doğrudan bulabiliriz. Plan, bu indeksten yararlanmak üzere kurulur.

Burada zarif bir iş bölümü ortaya çıkar. İndeks, adayları **daraltır**; ama sorguda indeksin kapsamadığı başka koşullar da varsa, o koşullar adaylar üzerinde **süzgeç** olarak uygulanır. Diyelim ki bölge alanının indeksi var ama yaş alanının yok. Sistem, önce bölge indeksinden o bölgedeki adayları çeker — belki bir milyondan birkaç bine iner — sonra bu birkaç bin aday üzerinde yaş koşulunu tek tek dener. Tüm belgeleri taramak yerine, indeksin daralttığı küçük aday kümesini süzmek, kıyaslanamayacak kadar ucuzdur. Bu "indeks daraltır, süzgeç inceltir" iş bölümü, sorgu yürütmenin kalbinde yatar.

8.4 Birden çok koşulu birleştirmek ve en seçiciyi seçmek

Birden çok indeksli koşul olduğunda, plan daha da inceltilir. "VE" ile birleştirilmiş koşullarda, sistem her koşulun ne kadar seçici olduğunu tahmin eder ve **en seçici** olanından başlar — çünkü en az sayıda aday üreten koşul, sonraki süzgeçlerin işini en aza indirir. Bölge bir milyon belgeden bini seçiyor, durum ise yarısını seçiyorsa, plan bölgeden başlar; çünkü binlik bir aday kümesini süzmek, yarım milyonluk bir kümeyi süzmekten çok daha hızlıdır. Bazı sistemler bir adım daha ileri gider ve birden çok indeksin sonuçlarını **kesiştirerek** adayları daha da daraltır.

"VEYA" ile birleştirilmiş koşullarda mantık tersine döner: her koşulun adayları ayrı ayrı bulunur ve **birleştirilir**, çünkü koşullardan herhangi birine uyan her belge sonuca dahildir.

Bu kararların hepsinin altında tek bir soru yatar: hangi yol daha ucuz? Sistem bunu, **maliyet tahmini** yaparak yanıtlar. Her alanın değerlerinin ne kadar dağıldığına dair tuttuğu kabaca istatistiklere bakarak, bir indeksin kaç aday üreteceğini, bir süzgecin ne kadar iş gerektireceğini tahmin eder ve en ucuz görünen planı seçer. Bu tahminler kusursuz değildir; gerçeği yanlış tahmin ederse, sistem kötü bir plan seçip yavaşlayabilir. İyi bir sorgu işleyici, tahminlerini olabildiğince isabetli tutmaya çalışır, ama tahminin doğası gereği hata payı her zaman vardır.

8.5 Kardinalite, seçicilik ve histogramlar

Yukarıda durmadan "seçici" sözcüğünü kullandık; şimdi onu sayısal bir zemine oturtalım, çünkü eniyileyicinin tüm kararları bu kavramın üzerine kuruludur. Bir koşulun **seçiciliği** (selectivity), o koşulun belgelerin ne kadar küçük bir oranını geçirdiğidir: sıfıra yakınsa çok seçici (az belge geçer), bire yakınsa seçici değil (çoğu belge geçer). Bir koşulun, bir koleksiyona uygulandığında kaç belge döndüreceğinin tahminine ise **kardinalite tahmini** (cardinality estimation) denir. Seçicilik ile koleksiyon boyutunun çarpımı, beklenen sonuç kardinalitesini verir; eniyileyici planındaki her aşamanın çıktı boyutunu işte bu çarpımla kestirir.

Peki sistem, bir belgeyi okumadan, "bölge = Ege" koşulunun bir milyon belgeden kaçını geçireceğini nasıl tahmin eder? Bunu **istatistikler** aracılığıyla yapar. En kaba istatistik, bir alandaki **ayrık değer sayısıdır** (distinct değer): eğer bölge alanı yedi farklı değer alıyorsa, eşitlik koşulunun kabaca belgelerin yedide birini geçireceği varsayılır — buna *tekdüze dağılım varsayımı* denir. Ama gerçek veri nadiren tekdüzedir; bir bölgede nüfusun yarısı, ötekinde yüzde biri yaşayabilir. Bu çarpıklığı yakalamak için sistemler **histogram** tutar: alanın değer aralığını kovalara böler ve her kovaya kaç belgenin düştüğünü sayar. Bir aralık koşulunun ("yaş 30 ile 40 arası") seçiciliği, o aralığa düşen kovaların sayımlarının toplamından kestirilir. Sık görülen

birkaç değerin tek başına büyük yer kapladığı çarpık alanlarda, sistemler ayrıca bu **uç değerleri** (most common values) tek tek listeleyip geri kalanı histogramla modelleyebilir; böylece hem çarpıklık hem de kuyruk birlikte yakalanır.

Birden çok koşulun birleşik seçiciliğini tahmin etmek daha da çetrefildir. En yaygın varsayım, koşulların **bağımsız** olduğudur: "bölge = Ege" belgelerin yüzde onunu, "yaş < 40" yüzde yarısını geçiriyorsa, ikisinin birden seçiciliği 0,10 ile 0,50'nin çarpımı olan 0,05 sayılır. Bu varsayım pratik olsa da tehlikelidir; çünkü gerçek alanlar çoğu zaman **bağımlıdır** (örneğin posta kodu ile şehir). Bağımlılık göz ardı edilince tahmin gerçeğin çok altına ya da üstüne düşebilir, ve kötü kardinalite tahmini, kötü plan seçiminin bir numaralı nedenidir. İyi eniyileyiciler bu yüzden, birden çok alan üzerinde birlikte istatistik tutmaya ya da çalışma anında ölçüm yapıp planı düzeltmeye çalışır; ama tahmin hatası, sorgu eniyilemesinin kalıcı ve çözülmemiş zorluğudur.

8.6 Maliyet modeli: planları sayıya dökmek

Kardinalite tahmini, "her aşamadan kaç belge çıkar" sorusunu yanıtlar; ama eniyileyicinin asıl sorusu, "bu planı yürütmek ne kadar **iş** gerektirir"dir. Bu işi ölçen şey, **maliyet modelidir** (cost model). Maliyet modeli, bir planın tahmini bedelini soyut bir sayıya indirger ve genellikle iki tür temel işi ağırlıklandırarak toplar: **disk erişimi** (sayfa okuma/yazma) ve **işlemci işi** (belge çözme, koşul değerlendirme, karşılaştırma). Klasik System R modeli, bir erişim yolunun maliyetini "okunan sayfa sayısı + bir ağırlık çarpı işlenen kayıt sayısı" gibi bir formülle hesaplar; çünkü diskten bir sayfa getirmek, bellekte bir karşılaştırma yapmaktan kat kat pahalıdır.

Bu modelin neden bu kadar önemli olduğunu somut bir örnekle görelim. Diyelim bir koşul belgelerin yüzde otuzunu geçiriyor. Bunun için bir indeks var; ama indeksi kullanmak, eşleşen her belgenin kimliğini bulup sonra her birini diskten ayrı ayrı getirmek demektir — ve bu ayrı ayrı getirmeler, diskte dağınık, **rastgele** erişimlerdir. Oysa tam tarama, tüm belgeleri diskten **sıralı** okur; sıralı okuma, rastgele okumadan kat kat hızlıdır. İşte bu yüzden, bir koşul yeterince seçici *değilse*, indeksli olsa bile tam tarama daha ucuz olabilir: yüzde otuzu rastgele toplamaktansa, yüzde yüzü sıralı okuyup süzmek daha hızlı çıkar. Maliyet modeli, tam da bu sezgiyi sayıya döker ve eniyileyicinin "indeksi kullanmamak" gibi ilk bakışta tuhaf görünen, ama doğru kararları vermesini sağlar. Bu "her erişim yolunu maliyetlendir, en ucuzunu seç" yaklaşımı, System R'den bu yana maliyet-tabanlı eniyilemenin temelidir.

8.7 İndeks yoksa: tarama ve süzgeç

Bazen, sorgudaki hiçbir koşul indeksli bir alana denk gelmez. O zaman sistemin elinde tek seçenek kalır: **tarama**. Tüm belgeleri sırayla okur ve her birinde sorgunun koşullarını dener; uyanları sonuca alır, uymayanları atar. Bu, yedinci bölümde kaçınmaya çalıştığımız tam tarama maliyetidir; ama uygun bir indeks yoksa, kaçınılmazdır.

Burada, çoğu zaman gözden kaçan ama önemli bir incelik vardır. Taradığınız her belge için koşulu denemek, belgeyi okuyup içindeki ilgili alanı çözmeyi gerektirir. Oysa belge, beşinci bölümde değindiğimiz gibi, diskte sıkıştırılmış ya da kodlanmış bir biçimde durabilir; onu denetlemek için önce bu biçimden çözmek gerekir. Eğer milyonlarca belgenin **hepsini** baştan sona çözüp sonra çoğunu eler atarsanız, çok büyük bir emek boşa gider. Akıllı bir tasarım, bir belgenin koşula uyup uymadığını, onu tümüyle çözmeden, kodlanmış biçim üzerinde hızlıca kestirmeye çalışır; yalnızca koşula uyanları tam olarak çözer. Üçüncü kısımda OxiDB'nin tam da böyle, eşleşmeyen belgeleri hiç çözmeden atlayan bir "bayt düzeyinde süzme" yolu kullandığını ve bunun büyük taramalarda hem hızı hem belleği nasıl iyileştirdiğini göreceğiz.

8.8 Sıralama, atlama ve sınırlama

Çoğu sorgu, yalnızca "uyanları getir" demez; sonuçların belirli bir düzende, belirli bir miktarda gelmesini de ister: "şu alana göre sıralı, ilk on kayıt." Bu istekler, sorgu planını önemli ölçüde etkiler.

Sıralama, yedinci bölümdeki sıralı indekslerle doğrudan bağlantılıdır. Eğer sorgu, sıralı bir indeksi olan bir alana göre sıralı sonuç istiyorsa, sıralama işini ayrıca yapmaya gerek kalmaz; indeks zaten o sırayı tutar ve sonuçları o sırada gezerek üretebiliriz. Ama sıralanacak alanın indeksi yoksa, sistem önce tüm sonuçları toplar, sonra onları belleğe alıp sıralar — ki bu, büyük sonuç kümelerinde pahalı bir iştir.

Sınırlama (limit), yani "yalnızca ilk on" demek, sıralı bir indeksle birleştiğinde özellikle güçlüdür. Çünkü indeks sonuçları zaten sıralı verdiği için, sistem onuncu sonucu bulduğu an durabilir; gerisini hiç üretmeye gerek yoktur. Buna **erken sonlanma** denir ve milyonlarca kayıt arasından en büyük ya da en küçük birkaçını bulmayı neredeyse anlık hale getirir. **Atlama** (skip) ise sonuçların baştan belirli bir kısmını geçip kalanını döndürür; çoğu zaman sayfalama için sınırlamayla birlikte kullanılır.

Erken sonlanma yalnızca sıralamayla sınırlı değildir. "Şu koşula uyan **bir** kaydı getir" ya da "şu koşula uyan **ilk** kaydı güncelle" gibi sorgularda da sistem, ilk eşleşmeyi bulduğu an durabilir; kalan belgeleri taramaya gerek yoktur. Üçüncü kısımda OxiDB'nin tekil okuma, tekil güncelleme ve tekil silme işlemlerinde tam olarak bu erken sonlanmayı uyguladığını göreceğiz.

8.9 Yalnızca gerekeni döndürmek

Sorgu işleyicinin küçük ama yararlı bir görevi daha vardır. Bir kullanıcı çoğu zaman belgenin tamamını değil, yalnızca birkaç alanını ister: "kullanıcıların yalnızca adlarını ve e-postalarını getir." Sonucu yalnızca istenen alanlara indirgeme işine **izdüşüm** (projection) denir. İzdüşüm, hem ağ üzerinden taşınan veriyi azaltır hem de — yedinci bölümdeki kapsayan indeksleri hatırlayın — eğer istenen alanların hepsi zaten bir indekste varsa, belgeye hiç dokunmadan yanıt üretmeyi mümkün kılar.

8.10 İki kümeyi eşleştirmek: birleştirme algoritmaları

Şimdiye dek tek bir koleksiyondan belge süzmeyi konuştuk. Ama kimi sorular, iki ayrı kümeyi bir anahtar üzerinden **eşleştirmeyi** gerektirir: bir sipariş listesini, o siparişlerin sahibi kullanıcılarla buluşturmak gibi. Belge dünyasında bu, çoğu zaman bir sonraki bölümdeki birleştirme aşamasıyla yapılır; ama altta yatan algoritmalar, ilişkisel dünyadan ödünç alınmış üç klasik yöntemdir ve her birinin maliyet profili farklıdır.

Birincisi **iç içe döngü birleştirmesidir** (nested-loop join). En sezgisel olanıdır: dış kümedeki her belge için, iç kümenin tamamını gezip eşleşeni arar. İki küme N ve M boyutundaysa, maliyeti kabaca N çarpı M'dir — yani küçük kümelerde ucuz, büyük kümelerde felakettir. Ama bir incelik vardır: eğer iç küme tarafında, eşleştirme anahtarı üzerinde bir indeks varsa, her dış belge için tüm iç kümeyi taramak yerine indeksten doğrudan eşleşeni bulabiliriz; maliyet N çarpı M'den N çarpı (M'nin logaritması)'na iner. Bu "indeksli iç içe döngü", iç tarafta seçici bir indeks olduğunda hâlâ çok kullanışlıdır.

İkincisi **hash birleştirmesidir** (hash join). Fikri zariftir: önce iki kümeden **küçük** olanını gezip, eşleştirme anahtarına göre bir **hash tablosu** kurarsın; sonra büyük kümeyi bir kez gezip, her belgenin anahtarını bu hash tablosunda ararsın. Her arama neredeyse anlık olduğu için, toplam maliyet kabaca N artı M'dir — iç içe döngünün çarpımsal maliyetine kıyasla çok daha iyi. Hash birleştirmesi, eşitlik üzerinden yapılan birleştirmelerde ve hash tablosu belleğe sığdığında en hızlı seçenektir. Bir sonraki bölümdeki birleştirme aşamasının verimli gerçekleştirimleri, çoğu zaman tam olarak bir hash birleştirmesidir.

Üçüncüsü **sırala-birleştir birleştirmesidir** (sort-merge join). İki kümeyi de eşleştirme anahtarına göre sıralar, sonra ikisini bir fermuar gibi yan yana yürüterek eşleşenleri çıkarır. Sıralama maliyetlidir; ama kümeler zaten sıralıysa — örneğin bir indeks onları zaten sıralı veriyorsa — sıralama bedavaya gelir ve yürüme

Şekil 8.3: Üç birleştirme algoritması, üç maliyet profili.

adımı yalnızca N artı M'dir. Sırala-birleştir, eşitlik dışındaki karşılaştırmalarda (örneğin aralık birleştirmesi) ve sonucun da sıralı olması istendiğinde özellikle değerlidir. Eniyileyici, bu üç algoritma arasından, az önceki kardinalite tahminine ve maliyet modeline bakarak — kümeler ne kadar büyük, anahtarda indeks var mı, çıktının sıralı olması gerekiyor mu — birini seçer.

8.11 Yineleyici modeli: planı çalıştıran iskelet

Bir fiziksel plan seçildikten sonra, onu *çalıştıran* mekanizma da en az planın kendisi kadar zariftir. Yaygın yaklaşım, **yineleyici modeli** (iterator model) ya da kâşifinin adıyla **Volcano modelidir**.[2] Bu modelde plan, bir **yineleyici ağacı** olarak kurulur: her aşama — indeks taraması, süzgeç, sıralama, sınırlama — aynı basit arayüzü konuşan bir düğümdür ve bu arayüzün özü tek bir işlemdir: "bana bir sonraki belgeyi ver."

İş, yukarıdan aşağı bir **talep** olarak akar. En üstteki düğüme (diyelim sınırlama) "bir belge ver" denir; o, hemen altındaki düğümden (süzgeç) bir belge ister; süzgeç de altındaki indeks taramasından ister; indeks taraması bir aday üretir, süzgeç onu koşula göre kabul ya da reddeder, kabul edilen yukarı çıkar. Belgeler aşağıdan yukarı **birer birer** akar; hiçbir aşama, tüm belgeleri belleğe yığmak zorunda değildir. Bu modelin iki büyük erdemi vardır. Birincisi **bileşilebilirliktir**: her aşama aynı arayüzü konuştuğu için, onları lego gibi serbestçe üst üste dizebilirsiniz. İkincisi ise **akış halinde çalışabilmesidir**: sınırlama düğümü onuncu belgeyi aldığında "yeter" der ve talep akışını durdurur; alttaki indeks taraması da o anda durur, geri kalan milyonlarca belgeyi hiç üretmez. İşte erken sonlanma, yineleyici modelinin talep-güdümlü doğasından kendiliğinden doğar. Bu birer-birer çekme modeli, modern sorgu motorlarının büyük çoğunluğunun iskeletidir.

8.12 Bildirimsel olmanın asıl kazancı

Bu bölümü, ikinci bölümde attığımız bir tohumun nasıl meyve verdiğini görerek toparlayalım. İlişkisel modelin "ne iste, nasıl'ı sisteme bırak" ilkesinin asıl değeri, tam da burada ortaya çıkar. Kullanıcı yalnızca *ne* istediğini söylediği için, sistem *nasıl* sorusunu özgürce yanıtlayabilir: bir indeks ekleyince, kullanıcı sorgusunu hiç değiştirmeden, sistem o indeksi kullanmaya başlar; veri büyüyüp dağılım değişince, sistem planını ona göre uyarlar; daha iyi bir yürütme yolu mümkün hale gelince, kullanıcı bundan habersiz kazanç sağlar.

[2] G. Graefe, "Volcano—An Extensible and Parallel Query Evaluation System," *IEEE Trans. Knowledge and Data Engineering* 6(1), 1994.

Eğer sorgular, eski hiyerarşik ve ağ modellerinde olduğu gibi, "şu kayıttan şu bağlantıyı izle" diye adım adım yazılsaydı, bu esnekliğin hiçbiri mümkün olmazdı. Bildirimsel sorgu ile sorgu işleyicinin akıllı planlaması, aynı madalyonun iki yüzüdür: biri özgürlüğü tanır, diğeri o özgürlüğü performansa çevirir.

8.13 Bu bölümün bıraktığı yer

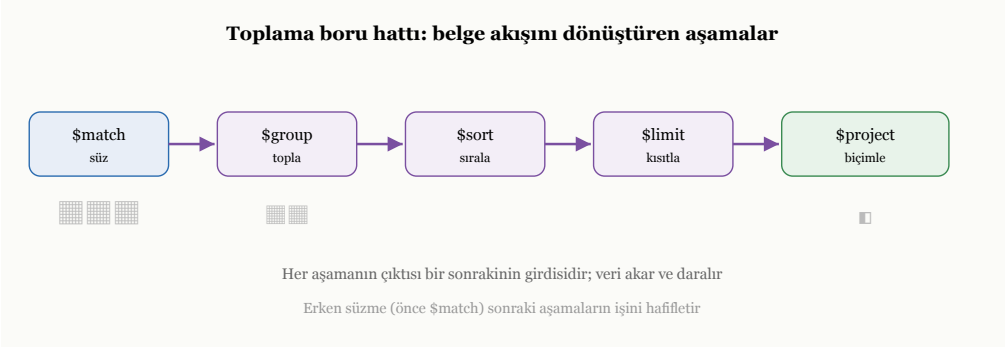
Bu bölümde, bir sorunun bir yanıta nasıl dönüştüğünü izledik: ham sorgunun ayrıştırılıp yapısal bir nesneye çevrilmesini; planlamanın hangi indeksi kullanacağına, koşulları hangi sırayla uygulayacağına maliyet tahminiyle karar vermesini; indeksin adayları daraltıp süzgecin onları inceltmesini; indeks yokken taramanın ve eşleşmeyeni çözmeden atlamanın inceliğini; sıralama, sınırlama ve erken sonlanmanın gücünü; ve tüm bunların, bildirimsel sorgunun tanıdığı özgürlükten doğduğunu gördük.

Şimdiye dek hep tek tek belgeleri bulup süzmekle ilgilendik: "şu koşullara uyan belgeleri getir." Ama veriye sorabileceğimiz daha zengin bir soru türü daha vardır. Tek tek belgeleri değil, **birçok belgenin toplu görüntüsünü** isteriz: "her bölgedeki ortalama yaş nedir", "her aydaki toplam satış kaçtır", "en çok satan on ürün hangileridir." Bu tür sorular, belgeleri yalnızca süzmekle kalmaz; onları **gruplar, özetler ve dönüştürür**. Bir sonraki bölümde, bu güçlü soru türünü — toplama (aggregation) ve onun ardındaki pipeline modelini — inceleyeceğiz.

Bölüm 9

Toplama: Pipeline Modeli, Gruplama ve Pencere Fonksiyonları

Şimdiye dek hep tek tek belgeleri bulup süzmekle ilgilendik: “şu koşullara uyan belgeleri getir.” Ama veriye sorabileceğimiz daha zengin bir soru türü vardır. Tek tek belgeleri değil, **birçok belgenin toplu görüntüsünü** isteriz: “her bölgedeki ortalama yaş nedir”, “her aydaki toplam satış kaçtır”, “en çok satan on ürün hangileridir.” Bu sorular, belgeleri yalnızca süzmekle kalmaz; onları **gruplar, özetler ve dönüştürür**. Bu bölüm, bu güçlü soru türünü — toplama (aggregation) ve onun ardındaki pipeline modelini — inceliyor. Toplama, bir veritabanını bir kayıt deposundan bir analiz aracına dönüştüren yetenektir.



Şekil 9.1: Toplama boru hattı: belge akışını dönüştüren ardışık aşamalar.

9.1 Süzmenin ötesi: veriyi özetlemek

Önceki bölümdeki sorgular, hep belgeleri **seçmekle** ilgiliydi; sonuç, her zaman var olan belgelerin bir alt kümesiydi. Toplama farklıdır: sonucu, var olan belgelerden **türetilmiş**, çoğu zaman hiçbir tek belgede bulunmayan yeni bir şeydir. “Her bölgedeki ortalama yaş” sorusunun yanıtı, hiçbir kullanıcı belgesinde yazmaz; o, birçok belgenin yaş değerlerini bir araya getirip hesaplanarak üretilir. Toplama, veriden bilgi **damıtmaktır**.

Bu tür soruları yanıtlamak, önceki bölümdeki araçlarla mümkün değildir. Süzgeçler bir belgeyi alır ya da atar; ama “bu bin belgenin ortalamasını al” diyemez. Gruplama, ortalama alma, en büyüğü bulma gibi işlemler, belgeleri **birleştiren** yeni bir mekanizma gerektirir. İşte toplama, bu mekanizmayı sunar.

9.2 Pipeline modeli: montaj hattı

Toplamayı düzenlemenin en yaygın ve en zarif yolu, **pipeline** modelidir. Pipeline'ı bir montaj hattı gibi düşünebilirsiniz. Bir uçtan belgeler akarak girer; hat boyunca bir dizi istasyondan geçer; her istasyon belgelere belirli bir işlem uygular ve sonucu bir sonraki istasyona aktarır; diğer uçtan ise nihai sonuç çıkar. Her istasyona pipeline'ın bir **aşaması** (stage) denir. Bir aşamanın çıktısı, doğrudan bir sonraki aşamanın girdisidir.

Bu modelin gücü, **bileşilebilirliğinde** yatar. Karmaşık bir analizi, sıfırdan tek bir dev işlem olarak yazmak yerine, her biri tek bir basit işi yapan küçük aşamaları arka arkaya dizerek kurarsınız. Bir aşama süzer, bir sonraki gruplar, bir sonraki sıralar, bir sonraki ilk onu alır. Tıpkı yapı taşlarıyla bir şey inşa etmek gibi: az sayıda basit, iyi tanımlanmış parçayı birleştirerek, sınırsız çeşitlilikte karmaşık analizler kurabilirsiniz. Bu, pipeline modelini hem anlaşılır hem de esnek kılar.

9.3 Pipeline'ın aşamaları

Bir pipeline'da en sık karşılaşılan aşamaları tanıyalım. Bunları, belirli bir ürünün sözdizimiyle değil, yaptıkları işle anlatacağız; çünkü fikirler, adlardan daha kalıcıdır.

Süzme aşaması, önceki bölümdeki sorgunun ta kendisidir: koşula uymayan belgeleri akıştan atar. Pipeline'ın genellikle ilk aşaması olarak kullanılır ve nedeni önemlidir: akışı erkenden daraltmak, sonraki tüm aşamaların işini azaltır. Üstelik bu erken süzme, önceki bölümde gördüğümüz indekslerden de yararlanabilir; yani pipeline'ın başındaki bir süzme, tüm koleksiyonu taramak yerine indeksle az sayıda belgeye inebilir.

Gruplama aşaması, toplamanın kalbidir ve birazdan ona ayrı bir başlık açacağız. Kısaca, belgeleri bir anahtara göre gruplara ayırır ve her grup için özet değerler hesaplar.

Sıralama, atlama ve sınırlama aşamaları, önceki bölümde tanıdığımız işlevlerin pipeline içindeki karşılıklarıdır: akışı belirli bir düzene sokar ya da belirli bir kısmına indirir. "En çok satan on ürün" sorusu, bir gruplamadan sonra gelen bir sıralama ve bir sınırlamayla yanıtlanır.

Yeniden şekillendirme aşamaları, her belgenin biçimini değiştirir: yeni hesaplanmış alanlar ekler, gereksiz alanları atar, alanları yeniden düzenler. Bunlar, akıştaki belgeleri nihai sonucun istediği biçime sokar.

Açma aşaması (unwind), ilginç ve güçlü bir dönüşümdür. İçinde bir liste barındıran bir belgeyi alır ve onu, listenin her öğesi için bir tane olmak üzere, **birçok belgeye** açar. Bir siparişin kalemler listesini düşünün; açma aşaması, o tek siparişi, her kalem için ayrı bir belgeye dönüştürür. Böylece liste öğelerini tek tek gruplayıp analiz edebilirsiniz — örneğin "her üründen toplam kaç adet satıldı" sorusunu, kalemleri açıp sonra ürüne göre gruplayarak yanıtlarsınız.

Birleştirme aşaması (lookup), başka bir koleksiyondan ilişkili belgeleri getirir. Üçüncü ve dördüncü bölümlerde, belge modelinin ilişkili veriyi ya gömdüğünü ya da ona referansla işaret ettiğini söylemiştik. Referansla bağlanmış veriyi bir araya getirmek gerektiğinde, birleştirme aşaması devreye girer; bu, belge dünyasının, ilişkisel modelin birleştirme gücünden ölçülü bir ödünç almasıdır.

Çok-yönlü analiz aşaması (facet), tek bir geçişte birden çok özeti birden üretir. Bir alışveriş sitesinin sonuç sayfasını düşünün: aynı süzülmüş ürün kümesinden, hem kategoriye göre sayıları, hem fiyat aralıklarının dağılımını, hem de en yüksek puanlı birkaç ürünü aynı anda istersiniz. Bu aşama, aynı girdi üzerinde birkaç alt-pipeline'ı paralel çalıştırıp her birinin sonucunu ayrı bir alanda toplar; böylece veriyi tek bir geçişte birden çok açıdan özetlersiniz.

9.4 Sıranın önemi

Pipeline'da aşamaların **sırası**, hem sonucu hem de performansı belirler. En temel ilke, süzmeyi olabildiğince erkene almaktır. Eğer önce gruplar, sonra süzerseniz, gereksiz belgeleri de gruplamış olursunuz; oysa

önce süzüp sonra gruplarsanız, yalnızca gereken belgeleri gruplarsınız. Akışı erken daraltmak, sonraki her aşamanın taşıması gereken yükü azaltır. Pipeline'ın açık, aşamalı yapısı, sistemin bu sırayı görüp — kullanıcı yazmasa bile — kimi durumlarda kendiliğinden eniyilemesine de olanak tanır; önceki bölümdeki bildirimsel özgürlüğün toplamadaki yankısıdır bu.

9.5 Gruplamanın derinliği

Gruplama, toplamanın kalbi olduğu için biraz daha yakından bakmayı hak eder. Gruplama iki şey ister: belgeleri hangi anahtara göre gruplayacağınızı (örneğin bölge) ve her grup için ne hesaplayacağınızı.

Her grup için hesaplanan değere bir **biriktirici** (accumulator) aracılığıyla ulaşılır. En yaygın biriktiriciler sezgiseldir: grup içindeki belgeleri **sayma**, bir alanın değerlerini **toplama**, **ortalamasını** alma, en **büyüğünü** ya da en **küçüğünü** bulma, ya da bir alanın tüm değerlerini bir **listede toplama**. Gruplama kavramsal olarak şöyle işler: sistem belgeleri tek tek gezer, her belgenin grup anahtarına bakar, o anahtara ait "kovaya" belgenin katkısını ekler. Tüm belgeler gezildiğinde, her kovada o grubun özeti hazırdır.

Burada önemli bir performans gerçeği vardır: gruplama, doğası gereği, **tüm ilgili belgeleri görmek** zorundadır. Bir grubun ortalamasını, o gruba ait son belgeyi görmeden hesaplayamazsınız. Bu yüzden gruplama, önceki bölümdeki erken sonlanma kestirmesinden yararlanamaz; bir koleksiyonu baştan sona taramayı gerektirebilir. İşte bu yüzden toplama sorguları, tekil aramalara kıyasla çoğunlukla daha pahalıdır.

Yine de zarif bir kestirme vardır. Eğer yalnızca **saymak** istiyorsanız ve gruplama anahtarının bir indeksi varsa — yedinci bölümdeki kapsayan indeksleri hatırlayın — belgelerin hiçbirine dokunmaya gerek kalmaz. İndeks, her anahtarın altında kaç belge olduğunu zaten bilir; saymak için yalnızca bu indeks bilgisini okumak yeterlidir. Üçüncü kısımda OxiDB'nin, indeksli bir alana göre yalnızca sayan gruplamaları, hiç belge okumadan, doğrudan indeksten yanıtladığını göreceğiz.

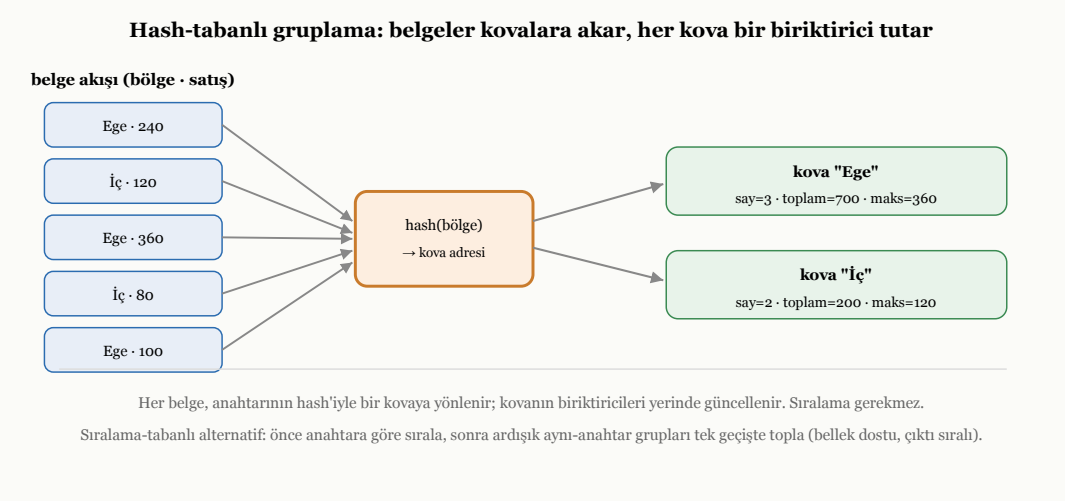
9.6 Gruplamanın iki gerçekleştirme yolu

Yukarıda gruplamayı "her belgeyi kendi kovasına ekle" diye anlattık; ama bu "kova" mecazının altında iki gerçek strateji yatar ve aralarındaki seçim, büyük veride performansı belirler.

Birincisi **hash-tabanlı gruplamadır** (hash-based grouping). Sistem, bellekte, grup anahtarından kovaya eşleyen bir **hash tablosu** tutar. Her belge geldiğinde, anahtarının hash'i hesaplanır, ilgili kova bulunur ve o kovanın biriktiricileri yerinde güncellenir — say bir artar, toplam değere eklenir, en büyük gerekirse yenilenir. Belgeleri önceden sıralamaya gerek yoktur; tek bir geçiş yeter. Bu yöntem hızlıdır, ama bir koşulu vardır: tüm grupların kovaları belleğe sığmalıdır. Ayrık grup sayısı çok büyükse — milyonlarca farklı anahtar — hash tablosu belleğe sığmaz ve sistemin kovaları diske taşıması gerekir ki bu maliyeti ciddi biçimde artırır.

İkincisi **sıralama-tabanlı gruplamadır** (sort-based grouping). Sistem önce tüm belgeleri grup anahtarına göre **sıralar**; sıralandıktan sonra, aynı anahtara sahip belgeler arka arkaya dizilmiş olur. Artık tek bir geçişle akarsınız: anahtar değiştiği an, biten grubu çıktıya verir, biriktiricileri sıfırlar, yeni gruba başlarsınız. Bu yöntemin bedeli sıralamadır; ama iki büyük avantajı vardır. Birincisi, herhangi bir anda yalnızca **tek bir grubun** durumunu bellekte tutarsınız — milyonlarca grup olsa bile bellek sabit kalır. İkincisi, eğer veri zaten o anahtara göre sıralı geliyorsa — örneğin gruplama anahtarının sıralı bir indeksi varsa — sıralama bedavaya gelir ve sıralama-tabanlı gruplama, tek bir akış geçişine indirgenir. Bir önceki bölümdeki sırala-birleştir birleştirmesiyle aynı sezgi burada da iş başındadır: sıralı düzen, gruplamayı ucuzlatır.

Eniyileyici, ikisi arasında, beklenen ayrık grup sayısına ve girdinin zaten sıralı olup olmadığına bakarak seçim yapar: az sayıda grup ve sıralanmamış girdi varsa hash; çok sayıda grup ya da zaten sıralı girdi varsa sıralama.



Şekil 9.2: Hash-tabanlı gruplama.

9.7 Birleştirme aşaması, perde arkasında bir hash birleştirmesidir

Yukarıda birleştirme aşamasının başka bir koleksiyondan ilişkili belgeleri getirdiğini söyledik. Bunun nasıl yapıldığı, bir önceki bölümdeki birleştirme algoritmalarına doğrudan bağlanır. En saf gerçekleştirme, dış akıştaki her belge için ilgili koleksiyonu baştan sona taramaktır — yani bir iç içe döngü birleştirmesi; bu, dış akış uzunsa felaket olur. Verimli motorlar bunun yerine, ilgili koleksiyondan eşleştirme anahtarı üzerinde bir **hash tablosu** kurar ve dış akışı bir kez gezip her belgenin eşleşenini bu tablodan anlık olarak çeker — yani bir hash birleştirmesi. Daha da iyisi, eşleştirme anahtarının hedef koleksiyonda bir **indeksi** varsa, indeksli iç içe döngüyle her dış belge için eşleşen doğrudan indeksten bulunur ve hiçbir tam tarama gerekmez. Belge dünyasında birleştirmenin "pahalı" diye anılmasının nedeni, çoğu zaman bu indeksin eksik olması ve aşamanın saf, taramalı biçime düşmesidir.

9.8 Pencere fonksiyonları: satırı yok etmeden hesaplamak

Toplamanın bir de bambaşka bir yüzü vardır ve onu gruplamayla karşılaştırarak anlamak en kolayıdır. Gruplama, birçok belgeyi tek bir özete **çökertir**: bin satıştan, her bölge için tek bir ortalama kalır. Ama bazı analizlerde, satırları kaybetmek istemeyiz; her belgeyi olduğu gibi tutmak, ama her birine, **komşularına bakarak** hesaplanmış yeni bir değer eklemek isteriz. İşte bunu yapan araca **pencere fonksiyonu** (window function) denir.

Aradaki farkı bir örnekle netleştirelim. Günlük satışlarınız olsun. Gruplama size "bu ayın toplam satışı" gibi tek bir özet verir — günler kaybolur. Pencere fonksiyonu ise her günü olduğu gibi bırakır, ama her güne yanına bir sütun ekler: "yılbaşından bu güne kadarki kümülatif toplam", ya da "son yedi günün hareketli ortalaması", ya da "bu günün satış sıralamasındaki yeri", ya da "bir önceki günün satışı." Her satır yerinde durur; yalnızca, komşu satırlardan türetilmiş bir bilgiyle zenginleşir.

Pencere fonksiyonu üç şeyi belirleyerek çalışır. Önce belgeleri **bölümlere** ayırır — örneğin her bölgeyi ayrı bir bölüm olarak. Sonra her bölümü bir alana göre **sıralar** — örneğin tarihe göre. En sonra, her belge için, o belgenin çevresindeki bir **pencereye** (örneğin "baştan bu satıra kadar" ya da "son yedi satır") bakarak istenen değeri hesaplar. Kümülatif toplam, baştan o satıra kadar olan pencereyi toplar; hareketli ortalama, son birkaç satırın penceresini ortalar; sıralama ise satırın bölüm içindeki yerini söyler. Üçüncü kısımda

OxiDB'nin pencere fonksiyonlarını tam da böyle — bölümle, sırala, pencere üzerinde hesapla — gerçekleştirdiğini göreceğiz.

Bu üç adımın — bölümle, sırala, çerçeveyi hesapla — en ince yeri sonuncusudur ve biraz daha açmayı hak eder. "Çerçeve" (frame), her satır için tam olarak hangi komşu satırların hesaba katılacağını söyler ve iki farklı biçimde tanımlanabilir.

Şekil 9.3: Pencere çerçevesi.

Birincisi **satır çerçevesidir** (row frame): çerçeve, geçerli satıra göre **konumla** belirlenir — "baştan bu satıra kadar olan tüm satırlar" (kümülatif toplam böyle hesaplanır) ya da "bu satır ve ondan önceki altı satır" (yedi günlük hareketli ortalama böyle olur). Burada önemli olan, satırların sıradaki *yeridir*; değerleri ne olursa olsun, sayılarına göre çerçeveye girer ya da girmezler.

İkincisi **aralık çerçevesidir** (range frame): çerçeve, satırların **değerine** göre belirlenir — "sıralama anahtarı, geçerli satırınkine belirli bir aralık içinde yakın olan tüm satırlar." Burada kaç satır olduğu değil, hangi değerlere sahip oldukları önemlidir; aynı sıralama değerine sahip satırlar (örneğin aynı güne ait birden çok kayıt) hep birlikte çerçeveye girer ya da girmez. Aralık çerçeveleri özellikle zaman temelli analizlerde — "son 30 günün penceresi" gibi, satır sayısı değil takvim aralığı tanımlandığında — gereklidir. Bu ayrım, gerçekleştirimi de etkiler: satır çerçevesi basit konum aritmetiğiyle, aralık çerçevesi ise değer karşılaştırmasıyla yürütülür ve sıralama anahtarına erişim gerektirir. Üçüncü kısımda OxiDB'nin pencere fonksiyonlarını şimdilik satır temelli çerçevelerle gerçekleştirdiğini, aralık ve zaman temelli çerçeveleri ise henüz açık bir eksik olarak dürüstçe ele alacağız.

Gruplama ile pencere fonksiyonu, analitik soruların iki büyük ailesidir. Gruplama "her grup için tek bir özet ver" derken, pencere fonksiyonu "her satırı koru, ama ona komşularından türeyen bir değer ekle" der. Finansal raporlardan zaman serisi analizlerine, sıralama tablolarından dönemsel değişim hesaplarına kadar pek çok gerçek analiz, bu ikisinin birinden ya da birleşiminden doğar.

9.9 Toplamanın performans doğası

Toplama, tekil aramalardan farklı bir performans karakteri taşır ve bunu bilmek önemlidir. Gruplama ve birçok dönüşüm, doğası gereği çok sayıda belgeye, kimi zaman tüm koleksiyona dokunur. Bu maliyeti yönetmek için iyi bir toplama motoru birkaç şeye dikkat eder. Süzmeyi mümkün olduğunca öne çekerek akışı erkenden daraltır. Aşamaları, tüm veriyi belleğe yığmak yerine, belgeleri tek tek akıtarak işleyebildiği ölçüde **akış halinde** çalıştırır; böylece dev bir koleksiyonu, onu bütünüyle belleğe almadan özetleyebilir. Ve mümkün olan her yerde — saymadaki indeks kestirmesi gibi — indekslerden yararlanır. Yine de toplamanın özünde, "çoğu zaman çok şeye dokunmak" vardır; bu yüzden toplama performansını anlamak, tekil sorgu performansını anlamaktan farklı bir bakış gerektirir.

9.10 Bu bölümün bıraktığı yer

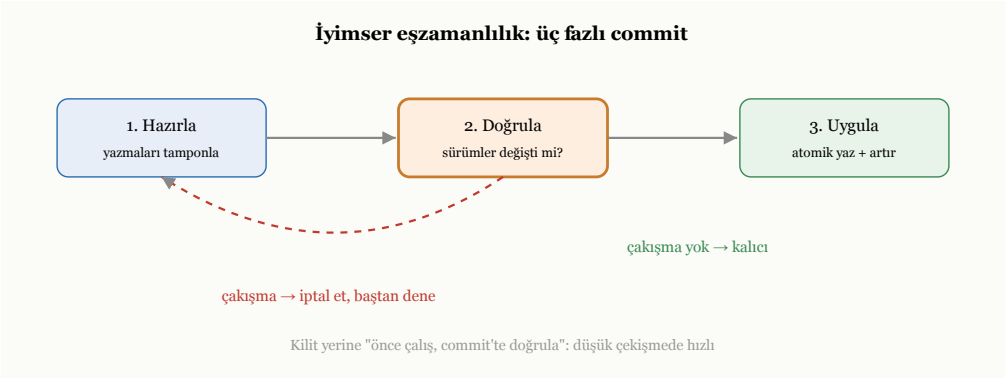
Bu bölümde, veriye sorabileceğimiz en zengin soru türünü — toplamayı — tanıdık. Toplamanın, belgeleri seçmekle kalmayıp onlardan yeni bilgi damıttığını; pipeline modelinin, karmaşık analizleri basit aşamaları arka arkaya dizerek kurmaya olanak tanıdığını; süzme, gruplama, açma, birleştirme ve çok-yönlü analiz gibi aşamaların ne işe yaradığını; gruplamanın bir anahtara göre biriktiricilerle çalıştığını ve doğası gereği çok şeye dokunduğunu; pencere fonksiyonlarının ise satırları korurken komşulardan değer türettiğini gördük.

Buraya kadar, Kısım II boyunca hep **okumayla** — veriyi saklamak, bulmak, süzmek, özetlemek — ilgilendik. Ama bir veritabanı yalnızca okunmaz; sürekli **yazılır**, hem de çoğu zaman birçok kullanıcı tarafından aynı anda. İşte o an, birinci bölümde değindiğimiz en sinsi sorunlar geri gelir: iki kişi aynı kaydı aynı anda değiştirirse ne olur; bir işlem yarıda kalırsa tutarlılık nasıl korunur; sistem, bir avuç kullanıcının kaosa sürüklemesine izin vermeden, düzeni nasıl sürdürür? Bir sonraki bölümde, veritabanının belki de en zarif kavramına — işlemlere (transactions) ve onların verdiği "ya hep ya hiç" güvencesine — eğiliyoruz.

Bölüm 10

İşlemler: ACID, Yalıtım, Kilitleme, MVCC ve OCC

Kısım II boyunca, dokuzuncu bölümün sonuna kadar, hep okumayla ilgilendik: veriyi saklamak, bulmak, süzmek, özetlemek. Ama bir veritabanı yalnızca okunmaz; sürekli yazılır, hem de çoğu zaman birçok kullanıcı tarafından aynı anda. İşte o an, birinci bölümde değindiğimiz en sinsi sorunlar geri gelir: iki kişi aynı kaydı aynı anda değiştirirse ne olur; bir işlem yarıda kalırsa tutarlılık nasıl korunur; sistem, bir avuç kullanıcının veriyi kaosa sürüklemesine izin vermeden düzeni nasıl sürdürür? Bu bölüm, veritabanının belki de en zarif kavramına — işlemlere — ve onların verdiği güvencelere eğiliyor.



Şekil 10.1: İyimser eşzamanlılıkta üç fazlı commit.

10.1 İşlem nedir: bölünmez bir bütün

Birinci bölümdeki para transferi örneğini hatırlayalım: bir hesaptan para düşmek ve diğerine eklemek. Bu iki adım, ayrı ayrı düşünüldüğünde tehlikelidir; çünkü arada bir çökme ya da bir karışıklık, parayı buharlaştırabilir. Asıl istediğimiz, bu iki adımın **tek bir bölünmez bütün** olarak ele alınmasıdır: ya ikisi de olur ya da hiçbiri. İşte bir **işlem** (transaction), tam da budur — birçok işlemi tek bir mantıksal birim olarak gruplayan, "ya hep ya hiç" güvencesi veren bir kavram.

İşlem, kullanıcıya basit bir söz verir: işlemin içindeki adımlar, dışarıdan bakıldığında, tek bir an'da, bölünmeden gerçekleşmiş gibi görünür. Yarıda kalmış, yarısı olmuş bir durum asla görünmez. Bu söz, sandığınızdan çok daha zor tutulur; çünkü hem çökmeyle hem de aynı anda çalışan diğer işlemlerle baş etmek gerekir. İşlem

kavramının erdemleri ve sınırları, veritabanı kuramında klasik bir incelemenin konusudur.[1] Bu güvencelerin tümü, geleneksel olarak dört harfle özetlenir: ACID.

10.2 Dört güvence: ACID

ACID, bir işlemin verdiği dört sözün İngilizce baş harflerinden oluşur. Bu dördünü tanımak, işlemlerin neyi vaat ettiğini — ve bu bölümün geri kalanında hangi sorunu çözeceğimizi — netleştirir.

Atomiklik (atomicity), "ya hep ya hiç" sözüdür: işlemin tüm adımları gerçekleşir ya da hiçbiri gerçekleşmez. Bu güvencenin temeli, altıncı bölümde gördüğümüz yazma-öncesi günlük ve kurtarmadır; işlem yarıda çökse bile, kurtarma ya işlemi tümüyle tamamlar ya da tümüyle geri alır.

Tutarlılık (consistency), işlemin veritabanını bir geçerli durumdan başka bir geçerli duruma taşımasıdır; veriye dair kuralların — örneğin "bir hesap bakiyesi eksiye düşemez" — işlem sonunda hâlâ geçerli olmasıdır. Dürüst olmak gerekir ki bu "C", büyük ölçüde uygulamanın sorumluluğundadır: veritabanı, işlemin atomik ve yalıtılmış olmasını sağlar; ama hangi durumların "geçerli" olduğunu tanımlamak ve gözetmek, çoğunlukla uygulamaya düşer.

Yalıtım (isolation), bu bölümün asıl zor konusudur ve birazdan ona uzun uzun eğileceğiz. Kısaca: aynı anda çalışan işlemler birbirinin ayağına basmamalı; sonuç, sanki işlemler sırayla, teker teker çalışmış gibi olmalıdır — gerçekte hepsi iç içe, aynı anda çalışsa bile.

Dayanıklılık (durability), altıncı bölümün konusuydu: bir işlem "tamamlandı" dendiğinde, artık hiçbir çökme onu geri alamaz.

Bu dört güvenceden ikisini — atomiklik ve dayanıklılık — daha önce, WAL bölümünde temellendirmiştik. Tutarlılık ise büyük ölçüde uygulamanın işidir. Geriye, gerçekten zor ve bu bölümün kalbinde yatan olan kalır: yalıtım.

10.3 Yalıtımın çözdüğü sorun: eşzamanlılık anormallikleri

Yalıtımın neden zor olduğunu anlamak için, onun engellemeye çalıştığı bozulmaları görmek gerekir. Birden çok işlem aynı veriye aynı anda dokunduğunda ortaya çıkabilecek, "anormallik" denen birkaç klasik bozulma vardır.

Kayıp güncelleme: İki işlem aynı kaydı okur, ikisi de üzerinde çalışır, sonra sırayla kaydeder. İkincinin kaydı, birincininkini hiç haberi olmadan ezer; birincinin değişikliği kaybolur. Birinci bölümde bu örneği vermiştik; yalıtımın en temel görevi, bunu önlemektir.

Kirli okuma: Bir işlem, başka bir işlemin henüz "tamamlandı" dememiş, geçici değişikliğini okur. Eğer o diğer işlem sonradan geri alınırsa, ilk işlem hiç var olmamış bir veriye dayanarak karar vermiş olur.

Tekrarlanamayan okuma: Bir işlem aynı kaydı iki kez okur ve arada başka bir işlem o kaydı değiştirdiği için, iki okumada farklı değerler görür. İşlemin gözünde dünya, kendi ortasında değişmiştir.

Hayalet kayıt: Bir işlem belirli bir koşula uyan kayıtları sayar; arada başka bir işlem o koşula uyan yeni bir kayıt ekler; ilk işlem aynı sayımı tekrarladığında farklı bir sonuç görür — sanki bir hayalet belirmiştir.

Yazma eğriltme: Daha sinsi bir anormalliktir; çünkü iki işlem **farklı** kayıtlara yazsa bile, birlikte ortak bir kuralı bozabilirler. Klasik örnek: en az bir doktorun nöbette kalması gereken bir hastane. İki nöbetçi doktor vardır; ikisi de aynı anda izne çıkmak ister. Her işlem, kendi anlık görüntüsünde "öteki doktor hâlâ nöbette" diye görür, kuralın sağlandığını sanır ve kendi doktorunu izne çıkarır. İkisi de ayrı kayıtlara yazdığı için doğrudan çakışmazlar; ama ikisi birden tamamlandığında, kimse nöbette kalmaz — kural, ne birinin ne ötekinin yazdığına bakılarak değil, ancak ikisi birlikte düşünüldüğünde ihlal edilmiş olur.

[1] Jim Gray, "The Transaction Concept: Virtues and Limitations," *Proc. VLDB*, 1981.

Bu anormalliklerin hepsinin ortak kökü, işlemlerin birbirinin yarım kalmış ya da eşzamanlı çalışmasını "görebilmesidir". Mükemmel yalıtım, her işleme sanki veritabanında **yalnız kendisi varmış** gibi bir dünya sunmayı amaçlar. Bu anormalliklerin uzun yıllar gevşek, ürüne-bağımlı tariflerle anlatıldığını; bunların gerçekten kesin, gerçekleştirimden bağımsız bir biçimselleştirmesinin — işlemlerin okuma-yazma bağımlılıklarından kurulan bir çizge üzerinde — ancak sonraki araştırmalarla geldiğini belirtmek gerekir.[2] Bu biçimsel bakış, hangi yalıtım düzeyinin tam olarak hangi anormalliği engellediğini ürün broşürlerinin gevşek diline başvurmadan söyleyebilmemizi sağlar.

10.4 Yalıtım düzeyleri: bir tayf

Mükemmel yalıtım — her işlemin sanki tek başınaymış gibi çalışması — en güçlü güvencedir ve "serileştirilebilirlik" (serializability) diye anılır: sonuç, sanki işlemler bir sıraya dizilip teker teker çalışmış gibi olur. Ama bu güçlü güvence pahalıdır; çünkü işlemlerin birbirini görmesini tümüyle engellemek, eşzamanlılığı ve dolayısıyla performansı kısıtlar.

Bu yüzden veritabanları, yalıtımı bir **tayf** üzerinde sunar. En gevşek uçta, işlemler birbirinin yarım işini görebilir — hızlıdır ama yukarıdaki anormalliklerin çoğuna açıktır. Daha sıkı düzeyler, sırayla kirli okumayı, sonra tekrarlanamayan okumayı, en sonunda hayalet kayıtları da engeller; her sıkı düzey daha güvenlidir ama daha az eşzamanlılığa, yani daha düşük performansa mal olur. Bir veritabanı tasarımcısının ya da kullanıcısının seçimi, çoğu zaman bu tayf üzerinde "ne kadar güvence, ne kadar hız" sorusunu yanıtlamaktır. Mutlak doğru bir nokta yoktur; uygulamanın ne kadar güçlü bir güvenceye gerçekten ihtiyaç duyduğuna bağlıdır. Bu standart yalıtım düzeylerinin tam olarak neyi garanti edip neyi etmediği, ünlü bir eleştiride titizlikle çözümlenmiştir.[3]

10.5 Yalıtımı sağlamanın üç felsefesi

Peki yalıtım, pratikte nasıl sağlanır? Üç temel felsefe vardır ve her biri, eşzamanlılığa farklı bir tavırla yaklaşır.

10.5.1 Kilitleme: kötümser yaklaşım

İlk felsefe **kilitlemedir** ve dünyaya kötümser bakar: "çatışma olacağını varsay, baştan önle." Bir işlem bir veriye dokunmadan önce, onun üzerinde bir **kilit** alır; o kilit elindeyken, başka hiçbir işlem o veriye dokunamaz, sırada bekler. Tipik olarak iki tür kilit vardır: okuma kilidi (birçok okuyucu aynı anda tutabilir, ama bir yazıcıyı bekletir) ve yazma kilidi (yalnızca bir işleme verilir, herkesi bekletir). Bunu tek şeritli bir köprüye benzetebilirsiniz: köprüde bir anda yalnızca bir araç geçebilir, gerisi bekler.

Ne var ki kilit almak ve bırakmak tek başına serileştirilebilirliği garanti etmez; kilitlerin **ne zaman** alınıp bırakıldığı da önemlidir. Bunu güvence altına alan klasik disipline **iki fazlı kilitleme** (two-phase locking, 2PL) denir. Adı, her işlemin yaşamının iki keskin faza ayrılmasından gelir. Önce bir **büyüme fazı** (growing phase) gelir: işlem yalnızca kilit *alır*, hiç bırakmaz; ihtiyaç duydukça kilit kümesi büyür. Bir kez ilk kilidini bıraktığı an, geri dönüşü olmayan biçimde **küçülme fazına** (shrinking phase) geçer: artık yalnızca kilit *bırakabilir*, yeni hiçbir kilit alamaz. Bu basit kuralın — "bıraktıktan sonra bir daha alma" — şaşırtıcı bir sonucu vardır: işlemlerin serileştirilebilir bir sırada çalışmasını matematiksel olarak garanti eder.

Pratikte çoğu sistem bunun daha güçlü bir biçimini, **katı iki fazlı kilitlemeyi** (strict 2PL) kullanır: işlem, *yazma* kilitlerini büyüme fazında bırakmaz, tamamlanana (ya da geri alınana) kadar **hepsini sonuna dek**

[2] A. Adya, B. Liskov ve P. O'Neil, "Generalized Isolation Level Definitions," *Proc. IEEE ICDE*, 2000.

[3] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil ve P. O'Neil, "A Critique of ANSI SQL Isolation Levels," *Proc. ACM SIGMOD*, 1995.

tutar. Bunun nedeni kirli okumayı ve karmaşık geri-alma sorunlarını da önlemektir: bir işlem henüz tamamlanmadan kilidini bıraksaydı, başkası onun geçici değişikliğini okuyabilir ve o işlem sonradan geri alınınca ortalık karışırdı. Katı 2PL, "kilitleri commit'e kadar tut" diyerek bu kapıyı tümüyle kapatır.

Kilitleme, güvenliği kesin biçimde sağlar; ama eşzamanlılığı kısıtlar — herkes sırada beklediği için. Üstelik kendine özgü bir tehlike doğurur: **kilitlenme** (deadlock). İki işlem düşünün; birincisi A verisinin kilidini almış, B'yi bekliyor; ikincisi B'nin kilidini almış, A'yı bekliyor. İkisi de ötekinin bırakmasını bekler ve hiçbiri bırakmaz; sonsuza dek donup kalırlar. Veritabanları bu tehlikeyle iki yolla baş eder. Birincisi **saptamadır** (deadlock detection): sistem, "kim kimi bekliyor" ilişkisini bir **bekleme çizgesi** (wait-for graph) olarak tutar; bu çizgede bir **döngü** belirdiği an, bir kilitlenme oluşmuş demektir. Sistem döngüyü kıracak bir "kurban" işlem seçer (genellikle en az iş yapmış olanı), onu iptal eder, kilitlerini bıraktırır ve sonra yeniden denemesine izin verir. Bazı sistemler çizgeyi sürekli izlemek yerine daha ucuz bir **zaman aşımı** (timeout) yaklaşımı kullanır: bir kilit için fazla bekleyen işlemi kilitlenmiş varsayıp iptal eder — basit ama bazen suçsuz işlemleri de iptal eder.

İkinci ve daha zarif yol, kilitlenmenin **hiç oluşmamasını** sağlayacak bir disiplin uygulamaktır (deadlock prevention). En temiz örnek, kilitleri her zaman aynı, belirli bir **toplam sıraya** göre almaktır; herkes kilitleri aynı sırayla alırsa, "A'yı tutup B'yi, B'yi tutup A'yı bekleme" döngüsü mantıken oluşamaz. Üçüncü kısımda OxiDB'nin tam da böyle, dokunacağı koleksiyonların kilitlerini sıralı bir düzende alarak kilitlenmeyi baştan imkânsız kıldığını — yani saptama yerine önlemeyi seçtiğini — göreceğiz.

10.5.2 MVCC: çok sürümlü eşzamanlılık

İkinci felsefe, kilitlemenin "okuyucular ve yazıcılar birbirini bekler" sorununa zarif bir çözüm getirir ve adı **çok sürümlü eşzamanlılık denetimidir** (MVCC) — fikrin kuramsal temelleri, eşzamanlılık denetimi araştırmasının kurucu metinlerinden birine uzanır.[4] Fikri şudur: bir veriyi değiştirirken eski sürümün üzerine yazma; onun **yeni bir sürümünü** oluştur, eskisini bir süre koru.

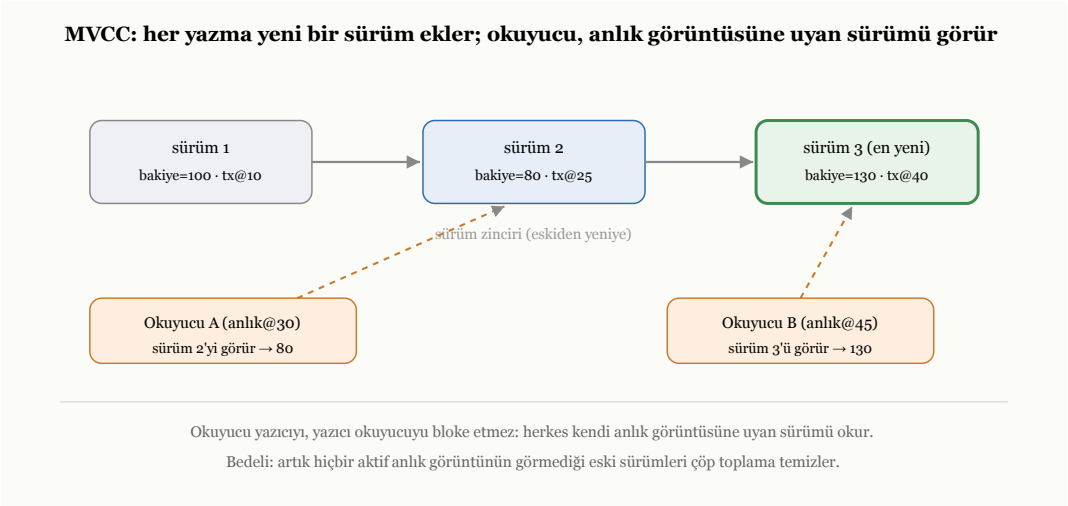
Bunun sonucu çarpıcıdır. Bir işlem okumaya başladığında, ona o an'ın bir **anlık görüntüsü** verilir — sanki o an bir fotoğraf çekilmiş gibi. İşlem, çalıştığı süre boyunca hep o tutarlı fotoğrafı görür; başkalarının yaptığı yeni değişiklikler onu etkilemez, çünkü o değişiklikler yeni sürümler yaratır, işlemin gördüğü eski sürüm olduğu gibi durur. Böylece okuyucular yazıcıları, yazıcılar da okuyucuları beklemez: **okuyucu yazıcıyı bloke etmez, yazıcı okuyucuyu bloke etmez.** Bu, kilitlemenin en büyük darboğazını ortadan kaldırır.

Bu mekanizmayı biraz daha somutlaştıralım. Her belge, artık tek bir değer değil, zamanda arkaya doğru uzanan bir **sürüm zinciridir** (version chain): her sürüm, onu yazan işlemin bir zaman damgasını ya da numarasını taşır ve bir önceki sürüme işaret eder. Bir işlem başladığında ona bir **anlık görüntü** (snapshot) — gerçekte, "şu numaraya kadar tamamlanmış işlemleri görebilirsin" diyen bir görünürlük ölçütü — atanır. Bir belgeyi okurken, sistem o belgenin sürüm zincirini en yeniden eskiye tarar ve işlemin anlık görüntüsüne **uyan** ilk sürümü döndürür: kendi anlık görüntüsünden sonra tamamlanmış sürümleri atlar, ondan önce tamamlanmış en yeni sürümü seçer. Bu kurala **anlık görüntü yalıtımı** (snapshot isolation) denir ve MVCC'nin sunduğu yalıtım düzeyinin adıdır.

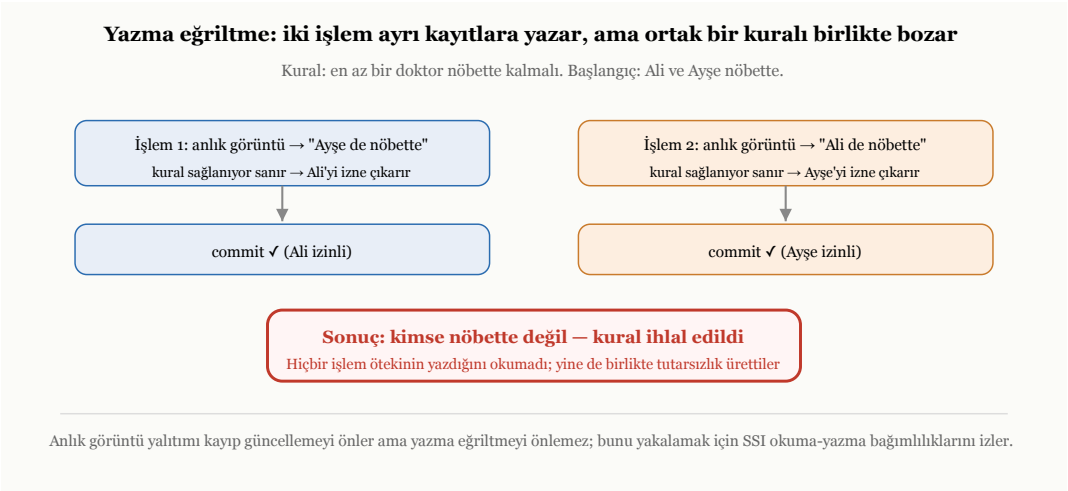
MVCC'nin bedeli, eski sürümleri saklamaktır; ve artık hiçbir aktif anlık görüntünün göremeyeceği eski sürümleri ara sıra temizlemek gerekir. Bu temizliğe **çöp toplama** (garbage collection) denir: sistem, en eski hâlâ açık anlık görüntünün gerisinde kalan, yani artık hiçbir işlemin asla göremeyeceği sürümleri saptayıp geri kazanır. Bu yüzden çok uzun süre açık kalan bir işlem, gizli bir maliyet doğurur: arkasında, temizlenemeyen bir sürüm yığını birikir ve hem disk hem de okuma maliyeti şişer. Burada beşinci bölümle güzel bir bağ kurulur: append-only depolama motorları, veriyi zaten asla üzerine yazmadan, hep yeni sürümler ekleyerek tuttuğu için, MVCC'ye doğal bir zemin sunar — eski sürümler orada zaten vardır.

Anlık görüntü yalıtımının önemli bir inceliği vardır: kayıp güncellemeyi, kirli okumayı ve tekrarlanamayan okumayı temiz biçimde önler; ama tek başına **serileştirilebilir değildir**. Az önce gördüğümüz yazma eğriltme anormalliği, tam da anlık görüntü yalıtımının yakalayamadığı boşluktur: iki işlem ayrı kayıtlara yazdığı için sürüm çakışması olmaz, doğrulamadan geçerler, ama ortak kuralı birlikte bozarlar.

[4] P. Bernstein ve N. Goodman, "Multiversion Concurrency Control—Theory and Algorithms," *ACM TODS* 8(4), 1983.



Şekil 10.2: MVCC sürüm zinciri.



Şekil 10.3: Yazma eğriltme.

Bu boşluğu kapatmak için geliştirilen yöntem, **serileştirilebilir anlık görüntü yalıtımıdır** (serializable snapshot isolation, SSI).[5] SSI, anlık görüntü yalıtımının üzerine ince bir izleme katmanı ekler: işlemler arasındaki okuma-yazma bağımlılıklarını gözler ve serileştirilebilirliği bozabilecek bir **tehlikeli yapı** — kabaca, eşzamanlı iki işlem arasında belirli bir bağımlılık örüntüsü — belirdiğinde, işlemlerden birini iptal eder. Böylece anlık görüntü yalıtımının okuyucu-yazıcı çakışmasızlığını korur, ama yazma eğriltme gibi son anormallikleri de eler. Bedeli, bu bağımlılıkları izlemenin getirdiği ek defter tutma yüküdür.

10.5.3 OCC: iyimser yaklaşım

Üçüncü felsefe, kilitlemenin tam tersi bir ruh hâliyle yaklaşır ve adı **iyimser eşzamanlılık denetimidir** (OCC) — fikri ilk kez resmî biçimde ortaya koyan çalışmaya kadar uzanır.[6] Varsayımı şudur: çatışmalar aslında nadirdir; çoğu zaman iki işlem aynı veriye aynı anda dokunmaz. Madem öyle, neden baştan kilitleyip herkesi bekletelim?

OCC şöyle çalışır. İşlem, hiçbir kilit almadan, serbestçe çalışır; yapacağı değişiklikleri hemen uygulamak yerine bir kenarda **biriktirir**. İş bittiğinde, "tamamla" demeden hemen önce, bir **doğrulama** adımı gelir: işlem, dokunduğu verilerin, kendisi çalışırken başka biri tarafından değiştirilip değiştirilmediğini kontrol eder. Bunu genellikle her veriye iliştirilmiş bir **sürüm numarası** ile yapar: işlem veriyi okuduğunda sürüm numarasını hatırlar; tamamlarken o numaranın hâlâ aynı olup olmadığına bakar. Hiçbir şey değişmemişse — ki iyimser varsayıma göre çoğu zaman böyledir — değişiklikler güvenle uygulanır. Ama bir çakışma saptanırsa — yani başka biri o veriyi bu arada değiştirmişse — işlem **iptal edilir** ve baştan denenir.

Bunu, bir mağazada alışverişe benzetebilirsiniz: ürünleri sepete koyarken kimseye sormazsınız (kilit yok); kasaya geldiğinizde, almak istediğiniz şeyin hâlâ uygun olup olmadığı kontrol edilir (doğrulama); bir sorun çıkarsa, o turu baştan yaparsınız (iptal ve yeniden deneme). OCC, çatışmaların nadir olduğu durumlarda muhteşemdir; çünkü hiç kimse boşuna beklemez. Ama çatışmaların sık olduğu, herkesin aynı veriye saldırdığı durumlarda israflıdır; çünkü çok sayıda işlem, sona kadar çalışıp sonra iptal edilip yeniden denenir. Üçüncü kısımda OxiDB'nin tam olarak bu iyimser yaklaşımı kullandığını — değişiklikleri tamamlanana dek biriktirdiğini, tamamlarken sürüm numaralarını doğruladığını ve çakışma bulursa iptal ettiğini — ayrıntısıyla göreceğiz.

10.6 Belge dünyasında işlemler: tek belge kolay, çoğu zor

Dördüncü bölümde, belge modelinde atomikliğin doğal sınırının tek bir belge olduğunu tohumlamıştık; şimdi o tohum meyve veriyor. Belge veritabanlarında, tek bir belgeyi değiştiren bir işlemi atomik kılmak görece kolaydır; çünkü o belge, tek bir bütün olarak yazılır. İşte dördüncü bölümde gömmenin bir avantajı olarak saydığımız şey buydu: ilişkili veriyi tek bir belgede topladığınızda, onları birlikte, tek bir atomik işlemde güncelleyebilirsiniz.

Zorluk, bir işlem **birden çok belgeye** dokunduğunda başlar. O zaman, birkaç ayrı belgenin değişikliğinin hep birlikte ya hep ya hiç gerçekleşmesini güvence altına almak gerekir ki bu, tek belgeye kıyasla daha fazla koordinasyon ister. Veri birden çok makineye dağılmışsa — ki on ikinci bölümün konusu bu olacak — zorluk daha da artar; çünkü artık farklı makinelerdeki değişikliklerin birlikte tamamlanması ya da birlikte geri alınması gerekir. Bunu sağlamak için, işlemin önce tüm taraflara "hazır mısın" diye sorduğu, hepsi "evet" derse "tamamla" dediği iki aşamalı bir mutabakat gibi düzenekler kullanılır; ama bu düzenekler hem yavaştır hem de kendi başına çökme senaryolarıyla doludur.

Buradan, dördüncü bölümdeki tasarım kararının önemi bir kez daha belirir: tutarlı kalması gereken birimi tek bir belgenin içine sığdırabiliyorsanız, işlemler basit ve hızlı kalır; bu birim birçok belgeye yayılıyorsa, daha güçlü ve daha pahalı işlem güvencelerine ihtiyaç duyarsınız.

---

[5] M. Cahill, U. Röhm ve A. Fekete, "Serializable Isolation for Snapshot Databases," *Proc. ACM SIGMOD*, 2008.

[6] H. T. Kung ve J. T. Robinson, "On Optimistic Methods for Concurrency Control," *ACM TODS* 6(2), 1981.

10.7 Bedava güvence yoktur

Bu bölümün her köşesinden aynı ders yükselir: güvence bedavaya gelmez. Daha güçlü yalıtım, daha az eşzamanlılık ve daha düşük performans demektir. Bir işlemin kapsamı genişledikçe — tek belgeden çok belgeye, tek makineden çok makineye — onu tutarlı tutmanın maliyeti artar. Kilitleme güvenliktir ama bekletir; MVCC beklemeyi azaltır ama sürüm saklamayı ve temizlemeyi getirir; OCC çatışma azken uçar ama çatışma çokken israf eder. Doğru seçim, her zaman olduğu gibi, iş yükünüze bağlıdır: çatışmalar ne sıklıkta oluyor, işlemler ne kadar geniş, hangi anormallikleri gerçekten önlemeniz gerekiyor?

10.8 Bu bölümün bıraktığı yer

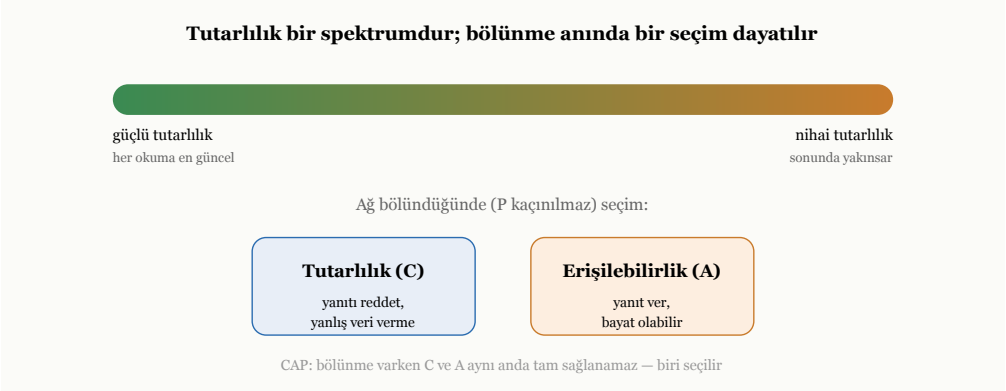
Bu bölümde, veritabanının eşzamanlı yazmalar ve yarım kalmalar karşısında düzeni nasıl koruduğunu gördük. İşlemin bölünmez bir bütün olduğunu; ACID'in dört güvencesini; yalıtımın engellediği anormallikleri ve yalıtım düzeyleri tayfını; ve yalıtımı sağlamanın üç felsefesini — kötümser kilitlemeyi (iki fazlı kilitleme, kilitlenme saptama ve önleme), çok sürümlü MVCC'yi (sürüm zinciri, anlık görüntü yalıtımı, çöp toplama ve onun yazma eğriltmeye karşı SSI ile güçlendirilmesi) ve iyimser OCC'yi — tanıdık. Tek belgeli işlemlerin kolay, çok belgeli ve dağıtık işlemlerin zor olduğunu ve her güvencenin bir bedeli olduğunu gördük.

Yalıtım, eşzamanlılığın yalnızca bir yüzüdür. Veri tek bir makinede dururken bu kavramlar yeterince zorludur; ama veri birden çok makineye yayıldığında, eşzamanlılık ve tutarlılık tümüyle yeni bir boyut kazanır. Farklı makineler aynı anlık görüntüyü görmeyebilir; bir makinedeki yazma, diğerine henüz ulaşmamış olabilir; "tutarlı" olmanın ne anlama geldiği bile tartışmalı hale gelir. Bir sonraki bölümde, eşzamanlılık ve tutarlılık modellerine — özellikle veri dağıldıkça ortaya çıkan o zorlu seçimlere — daha geniş bir çerçeveden bakacağız.

Bölüm 11

Eşzamanlılık ve Tutarlılık

Önceki bölümde yalıtımı — birçok işlemin tek bir makinede birbirinin ayağına basmadan çalışmasını — inceledik. Ama orada hep örtük bir varsayım vardı: verinin tek bir kopyası, tek bir yerde duruyordu. Bu varsayım, gerçek dünyanın büyük sistemlerinde çoğu zaman geçerli değildir. Veri, dayanıklılık, erişilebilirlik ve hız için birden çok makinede, çoğu zaman birden çok kopya halinde tutulur. İşte o an, sandığımızdan çok daha derin bir soru belirir: birden çok kopya varken, "verinin güncel değeri" ne demektir ve "tutarlı olmak" tam olarak neyi ifade eder? Bu bölüm, bu soruyu ele alıyor ve bizi, verinin nasıl dağıtıldığını anlatacak bir sonraki bölüme hazırlıyor.



Şekil 11.1: Tutarlılık bir spektrumdur; bölünme anında CAP seçimi dayatılır.

11.1 Neden birden çok kopya

Önce, neden tek bir kopyayla yetinmediğimizi kısaca görelim; ayrıntısı bir sonraki bölümün konusu, ama tutarlılık sorununun neden doğduğunu anlamak için gereklidir. Veriyi çoğaltmanın birkaç nedeni vardır. Birincisi **dayanıklılıktır**: tek bir makine, diski bozulursa ya da yanarsa veriyi tümüyle yitirebilir; aynı veriyi birkaç makinede tutarsanız, birinin kaybı felaket olmaktan çıkar. İkincisi **erişilebilirliktir**: bir makine çökse bile, kopyayı tutan diğerleri hizmete devam edebilir. Üçüncüsü **ölçektir**: okuma yükünü birçok kopyaya dağıtarak tek bir makinenin kapasitesinin ötesine geçebilirsiniz. Dördüncüsü **gecikmedir**: kullanıcıya coğrafi olarak yakın bir kopya, uzaktaki bir makineden daha hızlı yanıt verir.

Bu nedenlerin hepsi ikna edicidir ve büyük sistemler bu yüzden kaçınılmaz olarak veriyi çoğaltır. Ama her kopya, bir bedelle gelir ve o bedel, tam da bu bölümün konusudur: kopyalar birbirinden ayrı düşebilir.

11.2 Yeni sorun: kopyalar anlaşmazlığa düşebilir

Tek bir kopya varken, "güncel değer" nettir: o tek kopyada ne yazıyorsa odur. Ama birden çok kopya olduğunda, bir an için durup düşünün: bir kopyaya bir yazma yapıldı, ama bu yazma henüz diğer kopyalara ulaşmadı. O anda iki kopyaya bakarsanız, **farklı değerler** görürsünüz. Hangisi "doğru"dur? İkisi de, bir bakıma; biri yeni yazmayı görmüş, diğeri henüz görmemiştir. "Verinin güncel değeri" kavramı, birden çok kopyayla birlikte, bulanıklaşır.

Bunu, aynı bilgiyi tutan birkaç deftere benzetebilirsiniz. Bir deftere yeni bir satır yazdınız; ama o satırı henüz diğer defterlere geçirmediniz. Şimdi biri gelip ikinci deftere bakarsa, eski bilgiyi görür. Defterler birbirine eşitlenene kadar, "doğru bilgi" hangi deftere baktığınıza göre değişir. Dağıtık bir veritabanında durum tam olarak budur: kopyalar, eşitlenene kadar, gerçeğin farklı anlık görüntülerini taşır.

İşte tutarlılık, bu anlaşmazlığın ne kadarına izin verileceğiyle ilgilidir. Tutarlılık, bir tayf üzerinde tercih edilen bir özelliktir; bir uçta katı, öteki uçta gevşek davranan sistemler vardır.

11.3 Tutarlılık tayfı: katıdan gevşeğe

Tayfın bir ucunda **güçlü tutarlılık** (strong consistency) vardır. Güçlü tutarlı bir sistem, sanki tek bir kopya varmış gibi davranır: her okuma, en son yazılmış değeri görür. Bir veriyi yazdıktan sonra, hangi kopyaya bakarsanız bakın, yeni değeri görürsünüz. Bu, üzerine akıl yürütmesi en kolay modeldir — çünkü tıpkı tek makineli dünyadaki gibi davranır. Ama pahalıdır: yeni bir değerin tüm kopyalarda geçerli sayılması için, kopyalar arasında **koordinasyon** ve **bekleme** gerekir. Bu koordinasyon, hem yazmaları yavaşlatır hem de — birazdan göreceğimiz gibi — makineler arasındaki iletişim koptuğunda erişilebilirliği tehlikeye atar.

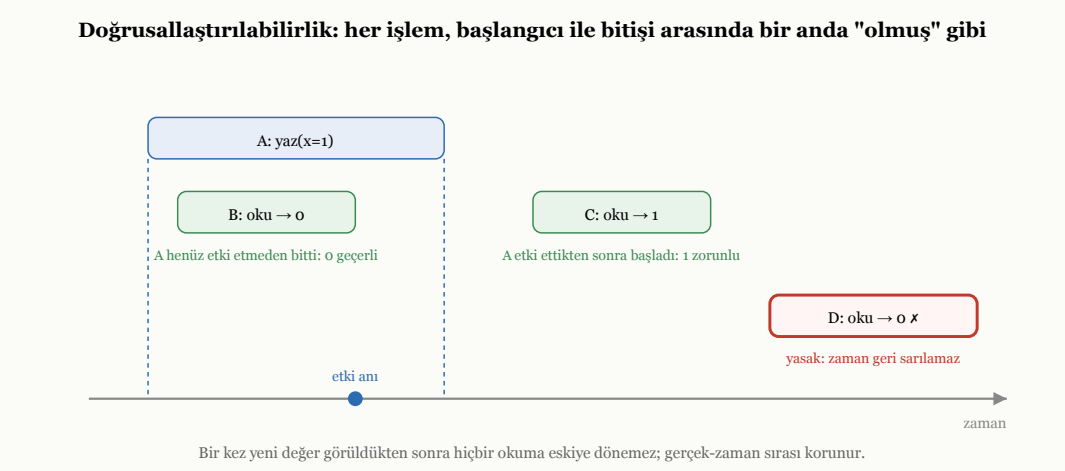
Tayfın öteki ucunda **nihai tutarlılık** (eventual consistency) vardır. Nihai tutarlı bir sistem, kopyaların geçici olarak anlaşmazlığa düşmesine izin verir; yalnızca şunu garanti eder: eğer yazmalar durursa, kopyalar eninde sonunda aynı değere **yakınsar**. Yani bir an için eski veri okuyabilirsiniz, ama sistem arkada sessizce eşitlenmeye devam eder ve sonunda herkes aynı gerçeği görür. Bu model ucuzdur, hızlıdır ve makineler arası iletişim koptuğunda bile hizmete devam edebilir; bedeli, okuyucuların ara sıra eski (bayat) veri görmesi ve uygulamanın buna katlanmak zorunda olmasıdır.

İki uç arasında, pratikte çok işe yarayan ara duraklar vardır. "Kendi yazdığını oku" (read-your-writes) güvencesi, en azından sizin yaptığınız bir değişikliği sonradan kendinizin göreceğini söyler — başkaları henüz görmese bile. "Tekdüze okuma" (monotonic reads), zamanı geriye sarmamayı, yani bir kez yeni bir değer gördükten sonra tekrar eskisine düşmemeyi garanti eder. "Nedensel tutarlılık" (causal consistency), neden-sonuç ilişkisi olan olayların herkese aynı sırada görünmesini sağlar — bir soruya verilen yanıt, soruyu görmeden görünmez. Bu ara modeller, güçlü tutarlılığın tüm maliyetini ödemeden, çıplak nihai tutarlılığın en can sıkıcı tuhaflıklarını giderir.

11.4 Güçlü tutarlılığı kesinleştirmek: doğrusallaştırılabilirlik

"Güçlü tutarlılık" dediğimiz şeyi sezgisel bıraktık; oysa onun kesin bir adı ve tanımı vardır: **doğrusallaştırılabilirlik** (linearizability). Bir sistemin doğrusallaştırılabilir olması şu demektir: her işlem, başladığı an ile bittiği an arasında **bir noktada**, anlık olarak gerçekleşmiş gibi davranır; ve bu anlık gerçekleşme noktaları, işlemlerin **gerçek-zaman** sırasına saygı gösterir. Yani A işlemi, B başlamadan önce bitmişse, A'nın etkisi B'nin etkisinden önce gelmek zorundadır. Sezgisel söylenişi şudur: sistem, sanki tek bir kopya varmış ve tüm işlemler tek bir sıraya diziliyormuş gibi davranır — üstelik bu sıra, dışarıdan gözlenen zaman akışıyla çelişmez. Bu kavramın eşzamanlı nesneler için kesin biçimselleştirmesi, eşzamanlılık kuramının kurucu metinlerinden birinde verilmiştir.[1]

[1] M. Herlihy ve J. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM TOPLAS* 12(3), 1990.



Şekil 11.2: Doğrusallaştırılabilirlik zaman çizgisi.

Doğrusallaştırılabilirliğin gerçek-zaman koşulu, onu daha gevşek bir akrabasından ayırır: **ardışık tutarlılık** (sequential consistency). Ardışık tutarlılık da işlemlerin tek bir ortak sıraya dizilmesini ister; ama bu sıranın gerçek-zaman akışına uymasını **şart koşmaz**. Yani A, B'den önce bitmiş olsa bile, ardışık tutarlı bir sistem onları sanki ters sırada olmuş gibi gösterebilir — yeter ki herkes *aynı* sırayı görsün ve her istemcinin kendi işlemleri kendi içinde sırada kalsın. Aradaki fark inceliklidir ama önemlidir: doğrusallaştırılabilirlik "global bir saatle uyumlu tek sıra" derken, ardışık tutarlılık yalnızca "tutarlı tek sıra" der. Bu yüzden doğrusallaştırılabilirlik daha güçlü, ama daha pahalıdır; çünkü gerçek-zaman sırasını korumak, kopyalar arasında daha sıkı koordinasyon ister.

Tüm bu modelleri bir merdiven gibi düşünebiliriz: en üstte doğrusallaştırılabilirlik, altında ardışık tutarlılık, onun altında nedensel tutarlılık, sonra oturum güvenceleri, en altta nihai tutarlılık. Yukarı çıktıkça güvence güçlenir ama gereken koordinasyon ve dolayısıyla maliyet artar; aşağı indikçe sistem hızlanır ve bölünmelere dayanıklılaşır, ama uygulamanın katlanması gereken tuhaflıklar çoğalır.

Şekil 11.3: Tutarlılık modelleri hiyerarşisi.

11.5 Neden mutlak güçlü tutarlılık her zaman mümkün değil: FLP

Bu noktada doğal bir soru belirir: madem güçlü tutarlılık bu kadar arzu edilir, neden onu her zaman, koşulsuz garanti edemiyoruz? Yanıtın derininde, dağıtık hesaplama kuramının en ünlü imkânsızlık sonucu yatar. Birden çok makinenin bir değer üzerinde **anlaşması** — buna mutabakat (consensus) denir — problemini düşünün. Şu olumsuz teorem kanıtlanmıştır: tümüyle eşzamansız bir ağda — yani mesajların ne zaman ulaşacağına dair hiçbir üst sınırın olmadığı, bir makinenin yavaş mı yoksa çökmüş mü olduğunu kesin ayırt edemediğiniz bir ortamda — tek bir makine bile çökebiliyorsa, her zaman sonlanacağı **garanti edilen** bir mutabakat algoritması **yoktur**.[2]

Bunun pratik anlamı şudur: "her zaman doğru karara varan ve her zaman da yanıt veren" bir sistem, eşzamansız bir dünyada matematiksel olarak imkânsızdır. Gerçek sistemler bu duvarı, varsayımları gevşeterek aşar: ya zaman aşımları gibi gevşek zamanlama varsayımları ekler (yavaşı çökmüşten ayırt etmeye çalışır), ya da kesin sonlanma yerine "neredeyse her zaman sonlanır" gibi olasılıksal güvencelerle yetinir. Bir sonraki bölümde göreceğimiz çoğunluk-tabanlı mutabakat protokolleri, işte bu gevşetmeleri kullanarak pratikte çalışır. FLP, bu protokollerin neden karmaşık olmak ve neden uç durumlarda "ilerleyememe" pahasına güvenliği korumak zorunda olduğunu açıklar — ve neden güçlü tutarlılığın hep bir bedeli olduğunu en derin düzeyde gösterir.

11.6 Bölünme anı ve kaçınılmaz seçim

Dağıtık sistemlerde tutarlılığı anlamanın kalbinde, sade ama derin bir gerçek yatar. Makineler birbiriyle bir ağ üzerinden konuşur ve ağlar bazen kopar. İki grup makine arasındaki iletişim kesildiğinde — buna **bölünme** (partition) denir — sistem ikiye ayrılır; her yarı, diğerinin ne yaptığını bilmez.

İşte tam o an, kaçınılmaz bir seçimle yüzleşirsiniz. Bir yazma isteği geldiğinde, sisteminizin iki seçeneği vardır. Ya isteği kabul edip hizmete devam edersiniz — ama o zaman, diğer yarıdan habersiz olduğunuz için, kopyaların anlaşmazlığa düşmesine, yani tutarsızlığa razı olursunuz. Ya da tutarlılığı korumak için isteği reddedersiniz — "diğer yarıyla konuşamadığım için, doğru olduğundan emin olamadığım bir yanıtı vermem" dersiniz — ama o zaman erişilebilirlikten ödün verirsiniz. Bölünme sırasında, **tutarlılık ile erişilebilirlik arasında** seçim yapmak zorundasınızdır; ikisine birden sahip olamazsınız. Bu içgörü, dağıtık sistemler kuramının en bilinen sonucudur ve genellikle "CAP" adıyla anılır.[3] Önemli olan, onu doğru anlamaktır: bu bir "her zaman üçten ikisini seç" sloganı değildir; özellikle ağ bölündüğünde, tutarlılık ile erişilebilirlik arasında yapılması gereken bir tercihtir — ilkenin kâşifi de yıllar sonra bu inceliğin altını özellikle çizmiştir.[4]

Daha az bilinen ama eşit derecede önemli bir incelik vardır: ağ hiç bölünmese bile, tutarlılık bir gecikme bedeliyle gelir.[5] Güçlü tutarlılık daha fazla koordinasyon, koordinasyon da daha fazla bekleme demektir. Yani seçim yalnızca felaket anında değil, sıradan zamanda da karşınızdadır: ne kadar güçlü tutarlılık isterseniz, her işlem o kadar yavaşlar. Tutarlılık, bedava gelmeyen bir lükstür.

11.7 Güçlü tutarlılık nasıl sağlanır: otorite ve çoğunluk

Peki birden çok kopyaya rağmen güçlü tutarlılık isteyen bir sistem, bunu nasıl başarır? İki temel fikir vardır ve ikisi de bir sonraki bölümde ayrıntılanacak; burada yalnızca özlerini görelim.

---

[2] M. Fischer, N. Lynch ve M. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *Journal of the ACM* 32(2), 1985.

[3] S. Gilbert ve N. Lynch, "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services," *ACM SIGACT News* 33(2), 2002.

[4] E. Brewer, "CAP Twelve Years Later: How the 'Rules' Have Changed," *Computer (IEEE)* 45(2), 2012.

[5] D. J. Abadi, "Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story," *Computer (IEEE)* 45(2), 2012.

Birinci fikir, tek bir **otorite** belirlemektir. Kopyalardan biri "lider" olarak seçilir ve tüm yazmalar önce ondan geçer. Lider, yazmaları belirli bir sıraya koyar ve diğer kopyalara bu sırayla iletir. Böylece, dağınık kopyalar olsa da, yazmaların "doğru sırası" konusunda tek bir karar mercii olur. Bu, anlaşmazlığı önlemenin en sade yoludur.

İkinci ve daha sağlam fikir, **çoğunluk mutabakatıdır**. Bir yazmanın "tamamlandı" sayılması için, tek bir makinenin değil, kopyaların **çoğunluğunun** onu kabul etmesi beklenir. Bir komitenin oy vermesini düşünün: bir kararın geçerli olması için çoğunluğun "evet" demesi gerekir. Çoğunluğun gücü şuradan gelir: herhangi iki çoğunluk, en az bir üyede mutlaka kesişir. Bu yüzden bir yazma çoğunluk tarafından kabul edildiyse ve bir okuma da çoğunluğa danışıyorsa, okuma o yazmayı mutlaka görür — çünkü kesiştikleri o ortak üye, yazmayı bilmektedir. Çoğunluk mutabakatı, hem güçlü tutarlılığı sağlar hem de azınlığın — örneğin çöken ya da ağdan kopan birkaç makinenin — sistemi yanlış bir karara sürüklemesini engeller. Üçüncü kısımda OxiDB'nin, kümeleme kipinde tam da böyle bir çoğunluk-tabanlı mutabakat protokolü kullandığını göreceğiz.

11.8 Ayarlanabilir tutarlılık

Madem tutarlılık bir tayf üzerinde bir tercihtir, neden bu tercihi tek seferde, tüm sistem için sabitleyelim? Birçok modern sistem, tutarlılığı **işlem başına ayarlanabilir** kılar. Aynı veritabanına, bir yazma için "yalnızca bir kopya kabul etsin yeter, hızlı olsun" diyebilir; başka, daha kritik bir yazma için "çoğunluk kabul etmeden tamamlanmış sayma" diyebilirsiniz. Aynı şekilde okumalarda da, "en yakın kopyadan oku, bayat olabilir ama hızlı" ya da "en güncel değeri gör, gerekirse bekle" arasında seçim yapabilirsiniz.

Bu ayarlanabilirlik, üçüncü bölümdeki ilkenin tutarlılıktaki yankısıdır: ihtiyacınız olandan fazlasını ödemeyin. Her yazma çoğunluk onayı beklemek zorunda değildir; her okuma en güncel değeri görmek zorunda değildir. Hangi işlemin ne kadar güçlü bir güvenceye gerçekten ihtiyaç duyduğunu belirler ve gerisinde hızı seçersiniz. Üçüncü kısımda OxiDB'nin dayanıklılık tarafında böyle bir ayar — her yazmada diske zorlayan katı kip ile arada bir zorlayan gevşek kip arasında bir seçim — sunduğunu, ama okuma tarafındaki ince ayar düğmelerinin henüz sınırlı olduğunu dürüstçe ele alacağız.

11.9 Olayları sıralamak: mantıksal saatler

Buraya kadar hep "önce", "sonra", "aynı anda" gibi sözcükler kullandık; ama dağıtık bir sistemde bu sözcükler sandığımızdan çok daha kaygandır. Farklı makinelerin duvar saatleri tam olarak senkron değildir; biri ötekinden birkaç milisaniye ileride ya da geride olabilir. O zaman, iki ayrı makinede olan iki olaydan hangisinin "önce" olduğunu, duvar saatlerine bakarak güvenle söyleyemeyiz. Bu sorunun çözümü, fiziksel zamanı bir kenara bırakıp olayları **nedensellik** üzerinden sıralamaktır ve temeli, dağıtık sistemler kuramının kurucu metnine dayanır.[6]

Buradaki anahtar kavram, **"önce-olur" ilişkisidir** (happened-before): bir olay, ya aynı makinede ondan önce geldiyse, ya da ona bir mesaj gönderdiyse, "önce olmuş" sayılır; ve bu ilişki geçişlidir. İki olay bu zincirle birbirine bağlanamıyorsa, onlar **eşzamanlıdır** (concurrent) — nedensel olarak ilişkisizdirler ve aralarında "doğru" bir sıra yoktur. Bu ilişkiyi sayılarla yakalamak için **mantıksal saatler** (logical clocks) kullanılır: her makine bir sayaç tutar, her olayda artırır, gönderdiği her mesaja sayacını iliştirir ve bir mesaj aldığında kendi sayacını gelen değerle uyumlu hale getirir. Böylece, fiziksel saatlere hiç güvenmeden, nedensel olarak bağlı olayların doğru sırada numaralanması sağlanır.

Mantıksal saatlerin tek bir sayaçlı temel biçiminin bir sınırı vardır: iki olayın *eşzamanlı* mı yoksa nedensel olarak bağlı mı olduğunu kesin ayırt edemez. Bunu yapabilmek için **vektör saatleri** (vector clocks) kullanılır: tek bir sayaç yerine, her makine *tüm* makinelerin sayaçlarından oluşan bir vektör tutar. İki olayın vektörlerini karşılaştırarak, birinin ötekinden kesin olarak önce mi geldiğini, yoksa gerçekten eşzamanlı mı

[6] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM* 21(7), 1978.

olduklarını ayırt edebilirsiniz. İşte nedensel tutarlılığı — neden-sonuç ilişkili olayların herkese aynı sırada görünmesini — pratikte gerçekleştiren mekanizma, çoğu zaman bu vektör saatleridir; çünkü "hangi yazma hangisinden nedensel olarak önce geldi" sorusunu, fiziksel saatlere hiç güvenmeden, kesin yanıtlamayı sağlar.

11.10 Tutarlılık ihtiyacı doğruluktan doğar

Bu bölümü bir ilkeyle toparlayalım. Üçüncü bölümde "veri modeli erişim örüntülerini izler" demiştik; tutarlılık için de benzer bir ilke geçerlidir: **tutarlılık ihtiyacı, uygulamanın doğruluk gereksiniminden doğar.** Her uygulama güçlü tutarlılığa muhtaç değildir. Bir sosyal medya beğeni sayısının birkaç saniye geç güncellenmesi kimseyi incitmez; orada nihai tutarlılık fazlasıyla yeterlidir ve getirdiği hız değerlidir. Ama bir banka bakiyesinin ya da bir stok adedinin bayat okunması ciddi sonuçlar doğurabilir; orada güçlü tutarlılık şarttır ve maliyeti haklıdır. Doğru tercih, mutlak değil, uygulamaya özgüdür: yanlış sonucun ne kadar zararlı olduğuna bakar, gereken en zayıf güvenceyi seçer ve gerisinde performansı kazanırsınız.

11.11 Bu bölümün bıraktığı yer

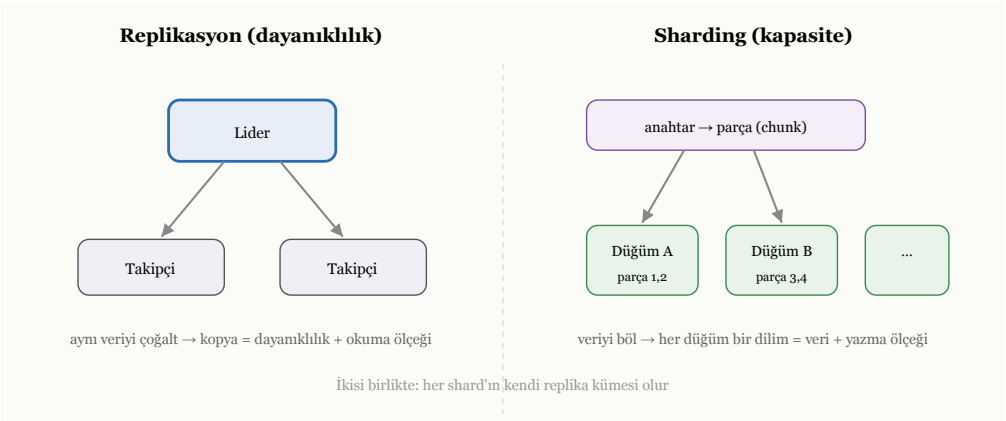
Bu bölümde, eşzamanlılığı tek makinenin ötesine taşıdık ve birden çok kopyayla birlikte gelen tutarlılık sorununu inceledik. Veriyi neden çoğalttığımızı; kopyaların neden anlaşmazlığa düşebildiğini; güçlü tutarlılıktan nihai tutarlılığa uzanan tayfı ve aradaki pratik durakları; güçlü tutarlılığın kesin adı olan doğrusallaştırılabilirliği ve onu ardışık tutarlılıktan ayıran gerçek-zaman koşulunu; mutabakatın eşzamansız bir dünyada neden koşulsuz garanti edilemediğini (FLP); ağ bölündüğünde tutarlılık ile erişilebilirlik arasındaki kaçınılmaz seçimi; güçlü tutarlılığın otorite ve çoğunluk mutabakatıyla nasıl sağlandığını; olayları fiziksel saate güvenmeden sıralamamızı sağlayan mantıksal ve vektör saatlerini; tutarlılığın işlem başına nasıl ayarlanabildiğini; ve tutarlılık ihtiyacının doğruluk gereksiniminden doğduğunu gördük.

Şimdiye dek hep tutarlılığın **anlamı** üzerinde durduk: kopyalar varken "doğru" ne demektir, hangi güvenceleri seçebiliriz. Henüz konuşmadığımız şey, **mekanizmadır**: veri pratikte birden çok makineye nasıl yayılır, kopyalar nasıl oluşturulup eşitlenir, bir makine çöktüğünde sistem nasıl ayakta kalır ve veri tek bir makineye sığmaz hale geldiğinde nasıl bölünür? Bir sonraki bölümde, ölçeklendirmenin iki büyük tekniğine — replikasyona ve sharding'e — ve bu bölümde tanıdığımız tutarlılık tercihlerini hayata geçiren düzeneklere eğiliyoruz.

Bölüm 12

Ölçeklendirme: Replikasyon, Konsensüs ve Sharding

Önceki bölümde, birden çok kopya varken tutarlılığın ne anlama geldiğini — yani işin **anlamını** — inceledik. Ama o kopyaların pratikte nasıl oluşturulduğunu, eşitlendiğini ve bir makine çöktüğünde sistemin nasıl ayakta kaldığını henüz konuşmadık. Bu bölüm, ölçeklendirmenin **mekanizmasını** ele alıyor: veriyi birden çok makineye yaymanın iki büyük tekniğini — replikasyonu ve sharding'i — ve bir önceki bölümde tanıdığımız tutarlılık tercihlerini hayata geçiren düzenekleri. Bu, bir veritabanını tek bir makinenin sınırlarının ötesine taşıyan yetenektir.



Şekil 12.1: Replikasyon ve sharding: dayanıklılık ile kapasitenin iki ayrı ekseni.

12.1 Neden tek makine yetmez

Bir veritabanı, uzun süre tek bir makinede mutlu yaşayabilir. Sorunlar, üç sınırdan birine dayandığında başlar. Birincisi **kapasitedir**: veri, tek bir makinenin diskine sığmayacak kadar büyüyebilir. İkincisi **iş hacmidir**: gelen istekler, tek bir makinenin işleyebileceğinden fazla olabilir. Üçüncüsü **erişilebilirliktir**: tek makine çökerse, tüm sistem birlikte çöker.

Bu sınırlara iki tür yanıt vardır. Birincisi **dikey ölçeklendirmedir**: makineyi büyütmek — daha çok bellek, daha hızlı disk, daha çok işlemci. Bu, bir yere kadar işe yarar; ama her zaman bir tavan vardır ve o tavana yaklaştıkça maliyet fırlar. İkincisi **yatay ölçeklendirmedir**: tek bir dev makine yerine, çok sayıda sıradan

makineyi birlikte çalıştırmak. Yatay ölçeklendirmenin tavanı çok daha yüksektir, ama bir bedeli vardır: artık birden çok makineyi koordine etmek zorundasınızdır ve önceki bölümün tüm tutarlılık zorlukları kapıdadır. Bu bölümdeki iki teknik — replikasyon ve sharding — yatay ölçeklendirmenin iki ayağıdır.

12.2 Replikasyon: aynı verinin kopyaları

Replikasyon, aynı veriyi birden çok makinede kopya halinde tutmaktır. Önceki bölümde *neden* çoğalttığımızı saymıştık — dayanıklılık, erişilebilirlik, okuma ölçeği, gecikme. Şimdi *nasıl* sorusuna geliyoruz.

En yaygın düzen, **lider-takipçi** modelidir. Kopyalardan biri **lider** olarak belirlenir ve tüm yazmalar ondan geçer. Lider, her değişikliği **takipçi** kopyalara iletir; takipçiler bu değişiklikleri kendi kopyalarına uygular. Okumalar ise hem liderden hem de takipçilerden karşılanabilir; böylece okuma yükü birçok makineye dağılır. Bunu, bir ana şube ile ona bağlı yan şubeleri olan bir kütüphaneye benzetebilirsiniz: yeni kitaplar ana şubeye gelir ve oradan yan şubelere dağıtılır; okuyucular ise en yakın şubeden hizmet alır.

Burada kritik bir tercih, liderin değişikliği takipçilere **ne zaman** kabul edilmiş saydığıdır. **Eşzamanlı** replikasyonda lider, bir yazmayı "tamamlandı" demeden önce takipçilerin onu aldığını bekler; bu güvenlidir — lider çökse bile veri takipçide vardır — ama yavaştır, çünkü her yazma takipçileri beklemek zorundadır. **Eşzamansız** replikasyonda lider beklemez; yazmayı hemen kabul eder ve takipçilere sonradan, kendi hızında iletir; bu hızlıdır, ama bir risk taşır: lider, bir değişikliği takipçilere iletmeden çökerse, o değişiklik kaybolabilir. Bu, doğrudan önceki bölümdeki güçlü-nihai tutarlılık tayfının replikasyondaki yüzüdür.

Pratikte çoğu sistem üçüncü bir yol izler: **yarı-eşzamanlı** replikasyon. Burada lider, tek bir yazmayı kabul etmeden önce takipçilerin **hepsini** değil, yalnızca **birini** (ya da yapılandırılabilir küçük bir alt kümesini) bekler. Böyle bir düzen, eşzamanlının güvenliğiyle eşzamansızın hızı arasında bir orta yol kurar: en az bir takipçide kopyası bulunduğu için yazma, liderin tek başına çökmesine dayanır; ama tüm takipçileri beklemediği için en yavaş takipçinin sistemi tıkamasına izin vermez. Eşzamanlı replikasyonun gizli tuzağı tam da budur: tek bir takipçi yavaşladığında ya da koptuğunda, onu bekleyen lider de durur; yani eşzamanlılık, dayanıklılığı artırırken erişilebilirliği düşürebilir. Yarı-eşzamanlı düzen, bu kuyruk gecikmesini (tail latency) en yavaş takipçiden kurtararak hafifletir.

12.3 Lider çökünce: failover ve iki büyük tehlike

Lider-takipçi modelinin asıl sınavı, **lider çöktüğünde** verilir. Sistem ayakta kalmak için, takipçilerden birini yeni lider olarak yükseltmelidir; bu sürece **failover** (devralma) denir. Kulağa basit gelir, ama iki sinsi tehlike içerir.

Birinci tehlike **kayıp yazmadır**: eğer replikasyon eşzamansızsa, yeni lider olacak takipçi, eski liderin son birkaç değişikliğini henüz almamış olabilir. O takipçi lider olunca, o değişiklikler kalıcı olarak yitirilir. İkinci ve daha tehlikeli olanı **ikiye bölünmüş beyindir** (split-brain): eski lider aslında ölmemiş, yalnızca ağdan kopmuşsa ve bu sırada bir takipçi de kendini yeni lider ilan etmişse, ortada **iki lider** belirir. İkisi de yazma kabul eder ve veri iki ayrı, çelişkili gerçeğe ayrılır; bu, onarılması son derece zor bir bozulmadır.

Bu iki tehlike, basit lider-takipçi modelinin yetmediği yeri gösterir. Asıl soru şudur: yeni lidere **kim** karar verir ve aynı anda iki liderin ortaya çıkmasını ne **engeller**? Bu soruların yanıtı, önceki bölümde tohumladığımız bir fikirde yatar: çoğunluk mutabakatı.

12.4 Üç topoloji: tek-lider, çok-lider, lidersiz

Şimdiye dek anlattığımız lider-takipçi düzeni, replikasyon **topolojilerinin** yalnızca biridir — en yaygın olanı: **tek-lider** (single-leader). Tek lider, tüm yazmaları bir noktadan geçirdiği için tutarlılık akıl yürütme-

sini basitleştirir; çünkü çakışan yazmalar diye bir sorun yoktur — her yazma aynı sıraya, aynı lidere uğrar. Bedeli, o tek liderin hem bir darboğaz hem de bir tekil çökme noktası olmasıdır.

İkinci topoloji **çok-liderdir** (multi-leader): birden çok kopya aynı anda yazma kabul eder ve değişikliklerini birbirine yayar. Bu, coğrafi olarak dağılmış sistemlerde caziptir — her bölge kendi yerel liderine yazar, gecikme düşer ve bir bölge ağ olarak koptuğunda diğerleri yazmaya devam eder. Ama ağır bir bedeli vardır: aynı veri iki ayrı liderde aynı anda farklı biçimde değiştirilebilir; buna **yazma çakışması** (write conflict) denir. Tek-liderde asla doğmayan bu sorun, çok-liderde merkezî bir kaygıdır ve çözülmesi gerekir — ya "son yazan kazanır" gibi bir kuralla (ki sessiz veri kaybı riski taşır), ya çakışan değerleri birleştiren özel veri yapılarıyla, ya da çakışmayı kullanıcıya geri sorarak. Çok-lider, erişilebilirlik ve yerel gecikme satın alır; karşılığında çakışma çözümünün tüm karmaşıklığını ödetir.

Üçüncü topoloji **lidersizdir** (leaderless): lider diye bir rol hiç yoktur. İstemci, bir yazmayı doğrudan birden çok kopyaya **paralel** gönderir; bir okumayı da birden çok kopyadan paralel ister ve gelen yanıtları karşılaştırır. Bu yaklaşımın atası, Amazon'un Dynamo sistemidir.[1] Lidersiz tasarımda asıl soru şudur: bir yazma kaç kopyaya ulaşmalı, bir okuma kaç kopyaya sormalı ki, sonuç güncel olsun? İşte burada, önceki bölümde tohumladığımız çoğunluk fikrinin bir akrabası devreye girer: **yetersayı** (quorum).

12.5 Yetersayı: W + R > N kuralı

Lidersiz bir sistemde, toplam **N** kopya olduğunu varsayalım. Her yazma, en az **W** kopyaya başarıyla işlenmeden "tamamlandı" sayılmaz; her okuma, en az **R** kopyadan yanıt toplar. Bu üç sayı arasındaki ilişki, sistemin ne kadar güncel veri döndüreceğini belirler. Sihirli koşul şudur:

W + R > N

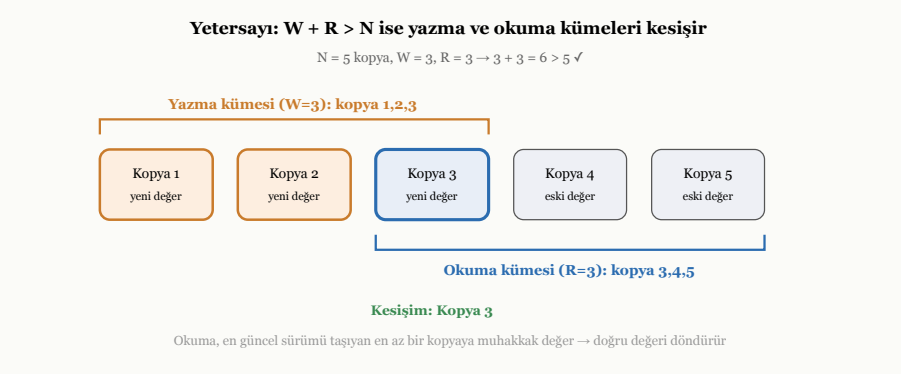
Bu eşitsizlik sağlandığında, yazma kümesi ile okuma kümesi **mutlaka en az bir kopyada kesişir.** Yani bir okuma, en güncel değeri taşıyan en az bir kopyaya muhakkak değer; çelişkili yanıtlar geldiğinde, en yeni sürümü (genelde bir sürüm damgasına bakarak) seçer ve doğru değeri döndürür. Bu, önceki bölümdeki çoğunluk-kesişimi fikrinin daha genel bir biçimidir: çoğunluk oylaması, aslında W ve R'nin ikisinin de çoğunluk (yarıdan fazlası) seçildiği özel bir yetersayı durumudur. Yetersayı çerçevesi, bu mantığı resmî olarak ilk kez ortaya koyan çalışmaya dayanır.[2]

Yetersayının zarafeti, **ayarlanabilir** olmasıdır. W'yi küçük, R'yi büyük seçerseniz, yazmalar hızlı ama okumalar maliyetli olur; tersini seçerseniz, yazma ağırdır ama okuma ucuzdur. Örneğin beş kopyalı bir sistemde W=3, R=3 hem yazmada hem okumada çoğunluğu zorlar (3+3 > 5) ve güçlü bir güncellik verir; ama hızlı okuma için W=4, R=2 seçilirse (4+2 > 5), okumalar yalnızca iki kopyaya sorar. Eğer W + R toplamı **N'yi aşmayacak** şekilde gevşetilirse — örneğin W=1, R=1 — sistem çok hızlanır ama artık nihai tutarlı hale gelir: bir okuma, henüz güncellenmemiş eski bir kopyaya düşebilir. Görüldüğü gibi yetersayı, önceki bölümdeki tutarlılık tayfını üç sayıyla **sürekli** biçimde ayarlamanın bir yoludur.

Yetersayının da gizli incelikleri vardır. Bir kopya geçici olarak çökmüşken yazma yine de W kopyaya ulaşabilsin diye, bazı sistemler yazmayı **geçici olarak yanlış bir kopyaya** emanet eder; çöken kopya geri döndüğünde bu emanet ona aktarılır. Buna *ipuçlu devir* (hinted handoff) denir. Ayrıca çakışan eşzamanlı yazmalar, lidersiz sistemlerde de — tıpkı çok-liderde olduğu gibi — bir çakışma çözümü gerektirir. Yetersayı, güncelliği **olasılıksal** değil **kümesel** bir güvenceyle sağlar; ama çakışma çözümünü, sürüm takibini ve onarım mekanizmalarını sırtlamak zorundadır.

[1] Giuseppe DeCandia vd., "Dynamo: Amazon's Highly Available Key-value Store," *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2007.

[2] David K. Gifford, "Weighted Voting for Replicated Data," *Proceedings of the 7th ACM Symposium on Operating Systems Principles (SOSP)*, 1979.



Şekil 12.2: Yetersayı kesişimi: W + R > N olduğunda yazma ve okuma kümeleri en az bir kopyada örtüşür.

12.6 Konsensüs: çoğunlukla anlaşmak

Konsensüs — önceki bölümde mutabakat diye andığımız kavramın aldığı ad — bir grup makinenin, bazıları çökse ya da ağdan kopsa bile, ortak bir karar üzerinde güvenle anlaşmasını sağlayan mekanizmadır. Modern dağıtık veritabanlarının çoğu, bu işi çoğunluk oylamasına dayanan bir protokolle yapar; en yaygın olanlardan birinin fikrini burada özetleyeceğiz.

Mantık şöyle işler. Makineler, yapılacak değişikliklerin **sıralı bir günlüğünü** üzerinde anlaşmaya çalışır. Bir değişikliğin günlüğe işlenmiş, yani "tamamlanmış" sayılması için, makinelerin **çoğunluğunun** onu kalıcı olarak kaydetmiş olması gerekir. Aynı şekilde, bir makinenin lider olabilmesi için de çoğunluğun oyunu alması gerekir. Çoğunluğun büyüsü, önceki bölümde değindiğimiz şu gerçekten gelir: herhangi iki çoğunluk, en az bir üyede mutlaka kesişir. Bu yüzden iki makine aynı anda lider olamaz — çünkü her ikisinin de çoğunluk oyu alması, en az bir makinenin ikisine birden oy vermesini gerektirir ki bu imkânsızdır. Böylece split-brain, tasarım gereği önlenmiş olur. Aynı kesişme özelliği, kayıp yazmayı da engeller: bir değişiklik çoğunluk tarafından kaydedildiyse, yeni lider seçilirken oy veren çoğunluk o değişikliği bilen en az bir makineyi içerir; dolayısıyla yeni lider o değişikliği asla kaybetmez.

Bu fikrin kuramsal temeli, dağıtık sistemler literatürünün dönüm noktalarından biridir. Bir grup makinenin, bazıları çökse bile tek bir değer üzerinde güvenle anlaşmasını sağlayan ilk eksiksiz protokol, **Paxos** adıyla ortaya konmuştur.[3] Paxos'un mantığı, hayalî bir adada toplanan ve sürekli salondan ayrılıp dönen bir meclisin, tutanağın tek bir tutarlı kopyası üzerinde nasıl anlaşabileceği benzetmesiyle anlatılır. Protokolün özü iki aşamalıdır. Önce bir makine, kendini geçici bir koordinatör (öneren) ilan etmek için çoğunluktan **söz** ister: "benden daha yeni bir öneriye söz vermediniz mi?" Çoğunluk bu sözü verirse, ikinci aşamada koordinatör asıl değeri önerir ve yine çoğunluğun kabulünü bekler. İki çoğunluğun her zaman kesiştiği gerçeği sayesinde, bir kez kabul edilmiş bir değer asla başka bir değerle ezilemez — çünkü yeni bir öneren, söz isterken o çoğunluğun en az bir üyesinden zaten kabul edilmiş değeri öğrenir ve onu korumak zorunda kalır. Paxos'un kötü şöhretli karmaşıklığı buradan doğar; sonraki yıllarda bu mantığı *anlaşılır* kılmayı açık bir tasarım hedefi yapan protokoller geliştirilmiştir ve OxiDB'nin kümeleme kipinde kullandığı protokol de bu daha anlaşılır soydandır — onu üçüncü kısımda ayrıntısıyla ele alacağız.

Konsensüsün gücü buradadır: tek bir protokolle hem güçlü tutarlılığı, hem otomatik failover'ı, hem de split-brain güvenliğini birlikte sağlar. Bedeli ise şudur: her yazma, çoğunluğa ulaşıp onların onayını beklemek zorundadır — yani bir gidiş-dönüş gecikmesi öder. Ayrıca, çoğunluğun anlamlı olması için yeterli sayıda makine gerekir; tipik olarak tek sayıda makine kullanılır ki "çoğunluk" net tanımlı olsun. Burada ince bir aritmetik vardır: **2f + 1** makineli bir küme, en fazla **f** makinenin çökmesine dayanır; çünkü kalan **f + 1** makine hâlâ çoğunluktur. Yani üç makine bir çökmeye, beş makine iki çökmeye dayanır. Çift sayıda makine kullanmak boşa kürek çekmektir: dört makine de yalnızca bir çökmeye dayanır (üçü gerekir), tıpkı üç makine gibi — ama bir makine daha beslemenin maliyetini öder. Bu yüzden konsensüs kümeleri neredeyse her zaman

[3] L. Lamport, "The Part-Time Parliament," *ACM Transactions on Computer Systems* 16(2), 1998.

tek sayıdadır. Bir başka incelik: çoğunluk **bölünmenin azınlık tarafında** kalan makineler, yazma kabul edemez — onlar çoğunluğu göremedikleri için duraklar. Bu, split-brain'i önlemenin diğer yüzüdür ve bedeli, azınlık tarafının geçici olarak hizmet veremeyişidir. Üçüncü kısımda OxiDB'nin, kümeleme kipinde tam da böyle bir çoğunluk-tabanlı konsensüs protokolü kullandığını ve lider seçimini, replikasyonu, failover'ı ve azınlığın lider seçememesini nasıl sağladığını ayrıntısıyla göreceğiz.

12.7 Sharding: veriyi bölmek

Replikasyon, aynı veriyi çoğaltarak erişilebilirlik ve okuma ölçeği sağlar; ama tek başına **kapasite** sorununu çözmez — her kopya yine tüm veriyi taşımak zorundadır. Veri, tek bir makinenin diskine sığmayacak kadar büyüdüğünde ya da yazma hacmi tek bir liderin kaldıramayacağı düzeye çıktığında, bambaşka bir tekniğe ihtiyaç vardır: **sharding** (parçalama). Sharding, veri kümesini makineler arasında **bölmektir**; her makine verinin yalnızca bir dilimini tutar. Böylece hem toplam kapasite hem de toplam yazma hacmi, makine sayısıyla birlikte büyür.

Sharding'in kalbindeki soru şudur: bir belgenin **hangi** makineye gideceğine nasıl karar verilir? Bu kararı veren alana **parça anahtarı** (shard key) denir — belgenin, hangi dilime ait olduğunu belirleyen bir alanı. Anahtarı dilimlere eşlemenin başlıca iki stratejisi vardır.

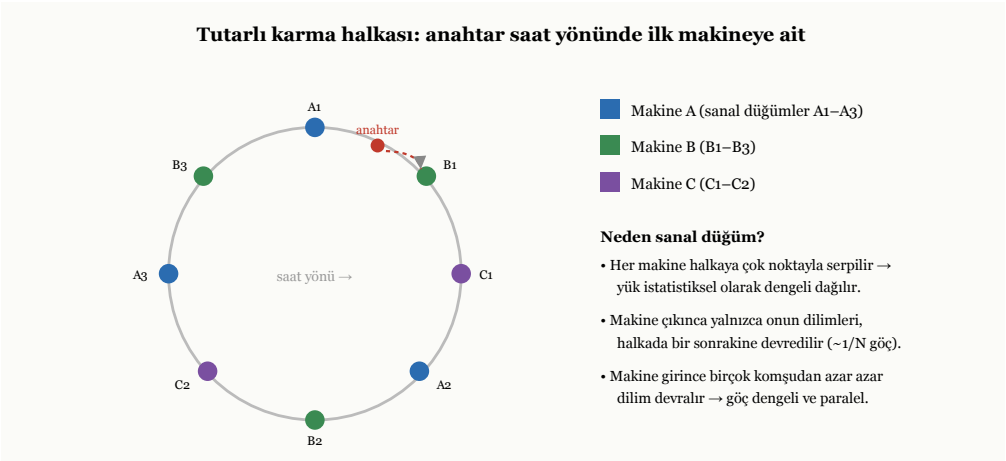
Birincisi **aralık bölümlemesidir**: anahtarın değer aralıklarına göre bölmek. Bir telefon rehberini A–H, I–P, R–Z diye üçe ayırmak gibi. Bu, aralık sorgularında iyidir — yakın değerler aynı makinededir — ama bir tehlike taşır: eğer yazmalar belirli bir aralıkta yoğunlaşırsa (örneğin son gelen kayıtlar hep en büyük anahtarı alıyorsa), o makine "sıcak nokta" haline gelip aşırı yüklenir, diğerleri boş durur. İkincisi **karma bölümlemesidir**: anahtarı bir karma fonksiyonundan geçirip sonuca göre dilim seçmek. Bu, kayıtları makinelere son derece dengeli dağıtır — sıcak nokta riski azalır — ama aralık yerelliğini kaybeder: yakın değerler farklı makinelere düşer, dolayısıyla aralık sorguları tüm makinelere yayılmak zorunda kalır.

Karma bölümlemenin saf hali bile, gözden kaçan bir kusur taşır. Diyelim ki dilim seçimini, anahtarın karmasını makine sayısına bölüp kalanını almakla yapıyoruz. Bu, makine sayısı **sabit** kaldığı sürece güzel çalışır; ama bir makine eklendiğinde ya da çıktığında, bölen değişir ve neredeyse **bütün** anahtarların dilimi değişir. Sonuç, kümenin tamamını yerinden oynatan dev bir veri göçüdür. Tam bu sorunu çözmek için icat edilmiş zarif bir teknik vardır: **tutarlı karma** (consistent hashing).[4]

Tutarlı karmanın fikri görsel olarak basittir. Karma değerlerini düz bir çizgi yerine bir **halka** üzerinde düşünün — saat kadranı gibi, en büyük değer en küçüğe bağlanan kapalı bir çember. Hem makineler hem de anahtarlar, karmaları aracılığıyla bu halka üzerinde bir noktaya düşer. Bir anahtarın hangi makineye ait olduğu şöyle bulunur: anahtarın noktasından saat yönünde yürünür ve rastlanan ilk makine o anahtarın sahibidir. Bu düzenin büyük kazancı şudur: bir makine halkadan çıktığında, yalnızca **ona ait olan** anahtarlar, halkada bir sonraki makineye devredilir; geri kalan her şey yerinde kalır. Yeni bir makine eklendiğinde de, yalnızca onun halkadaki komşusundan devraldığı dilim taşınır. Yani yeniden dengelemenin maliyeti, kümenin tamamıyla değil, eklenen ya da çıkan **tek makineyle** orantılı olur — ortalama olarak verinin yaklaşık 1/N'i yer değiştirir, hepsi değil.

Saf tutarlı karmanın bir zayıflığı kalır: makineler halka üzerinde rastgele dağıldığı için, biri şanssızca büyük bir yayı tek başına kaplayabilir ve orantısız yük üstlenebilir. Bunun çözümü, çoğu sistemin benimsediği zarif bir dolaylama katmanıdır: her fiziksel makineyi halkaya **bir** değil, **onlarca ya da yüzlerce nokta** olarak yerleştirmek. Bu noktalara **sanal düğüm** (virtual node) denir. Bir makine ne kadar çok sanal düğümle temsil edilirse, halka üzerindeki payı o kadar pürüzsüz ve istatistiksel olarak dengeli olur; ayrıca güçlü bir makineye daha çok sanal düğüm vererek yükü ağırlığına göre de dağıtabilirsiniz. Sanal düğümler, yeniden dengelemeyi daha da inceltir: yeni bir makine eklendiğinde, tek bir komşudan büyük bir blok almak yerine, halka boyunca serpilmiş birçok küçük dilimi birçok makineden azar azar devralır — böylece göç hem dengeli hem de paralel olur. Üçüncü kısımda OxiDB'nin sharding katmanının tam da böyle bir sanal-parça eşlemesi kullandığını göreceğiz.

[4] David Karger vd., "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web," *Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC)*, 1997.



Şekil 12.3: Tutarlı karma halkası: her makine için sanal düğümler yükü dengeli dağıtır; bir makine düşünce yalnızca komşu dilimleri devralır.

12.8 Yönlendirme ve parçalar-arası sorgu

Veri makinelere bölününce, gelen her istek **doğru** makineye gönderilmelidir. Bunu yapan bileşene **yönlendirici** (router) denir — çoğu zaman bir aracı (proxy) olarak çalışır. Yönlendirici, isteğe bakar, parça anahtarını çıkarır, hangi dilime ait olduğunu hesaplar ve isteği o dilimi tutan makineye iletir. Bunu, doğru masaya yönlendiren bir resepsiyona benzetebilirsiniz: anahtarınızı söylersiniz, sizi ait olduğunuz masaya gönderir.

Ama bir sorun vardır. Ya istek, parça anahtarını içermiyorsa? Ya da bir toplama sorgusu, tüm veriye birden bakmayı gerektiriyorsa? O zaman yönlendirici, isteği **tüm** makinelere göndermek, her birinden kısmi yanıtı toplamak ve bunları birleştirmek zorunda kalır. Bu örüntüye **dağıt-topla** (scatter-gather) denir. Bir sayım sorgusunu düşünün: yönlendirici, her dilime "sende kaç tane var" diye sorar, gelen sayıları toplar. Bir gruplama sorgusunda ise her dilim kendi kısmi gruplamasını yapar, yönlendirici bu kısmi sonuçları birleştirip nihai sonucu üretir. Üçüncü kısımda OxiDB'nin parçalar-arası toplama sorgularını tam olarak böyle, her dilimde kısmi hesap yapıp sonra birleştirerek — hem de tek-düğümle birebir aynı sonucu verecek biçimde — yürüttüğünü göreceğiz.

Burada, üçüncü bölümde attığımız bir tohum meyve verir. Kendi içinde bütün olan belgeler sharding'e doğal yatkındır: her belge bağımsız olduğu için, hangi dilime gideceğine kolayca karar verilir ve onu okumak için başka dilimlere bakmak gerekmez. Buna karşılık, yoğun biçimde birbirine bağlı veri — her yönden sorgulanan çok-çok ilişkiler — sharding'de zorlanır; çünkü ilişkili parçalar farklı makinelere düşebilir ve onları bir araya getirmek pahalı, parçalar-arası işlemler ise son derece çetin hale gelir.

12.9 İyi bir parça anahtarı ve yeniden dengelemenin yükü

Sharding'in başarısı, büyük ölçüde **parça anahtarının** iyi seçilmesine bağlıdır. İyi bir parça anahtarı üç özelliğe sahiptir. **Yüksek çeşitliliklidir**: çok sayıda farklı değeri vardır, böylece veri ince ince dağılabilir. **Erişimi dengelidir**: hiçbir değer ya da aralık, yükün orantısız bir kısmını üstlenmez; yani sıcak nokta oluşturmaz. Ve **yaygın sorgularla hizalıdır**: en sık yapılan sorgular, bu anahtarı içerir, böylece tüm dilimlere yayılmak yerine tek bir dilimi hedefler. Kötü seçilmiş bir parça anahtarı — örneğin çok az değeri olan ya da yükü tek bir makineye yığan bir alan — sharding'in tüm yararını yok edebilir.

Sharding'in göz ardı edilen bir maliyeti de **yeniden dengelemedir**. Veri büyüdükçe ya da makine eklendikçe-çıkarıldıkça, dilimlerin makineler arasında yeniden dağıtılması gerekir; bu, hareket halindeki veriyi tutarlı tutarken yapılması gereken, hassas bir operasyondur. Olgun sistemler, bu dengelemeyi

otomatik yöneten bir bileşen barındırır; ama bu otomatik dengeleme, kurması ve doğru işletmesi karmaşık bir yetenektir. Üçüncü kısımda OxiDB'nin sharding katmanını ele alırken, hangi parçaların manuel yapılandırıldığını ve hangi otomasyon düzeylerinin henüz tasarımda olmadığını dürüstçe değerlendireceğiz.

12.10 İkisini birleştirmek

Replikasyon ile sharding, birbirinin alternatifi değildir; farklı sorunları çözerler ve büyük sistemlerde **birlikte** kullanılırlar. Tipik bir büyük ölçek topolojisi şöyledir: veri, kapasite ve yazma hacmi için birçok dilime **bölünür** (sharding); ve sonra her dilim, erişilebilirlik ve dayanıklılık için kendi içinde **çoğaltılır** (replikasyon). Böylece sistem hem tek makineye sığmayan veriyi taşır, hem de herhangi bir makinenin çökmesine dayanır. Bu iki tekniğin birleşimi, modern büyük ölçekli veri sistemlerinin standart iskeletidir.

12.11 Yine aynı ders: dağıtım bedava değildir

Bu bölüm de, kitap boyunca tekrarlanan o dersi yeniden söyler. Dağıtım, kapasite ve erişilebilirlik satın alır; ama bunu koordinasyon, karmaşıklık ve sorgu kısıtlarıyla öder. Replikasyon, tutarlılık ile gecikme arasında bir tercih dayatır. Konsensüs, güçlü güvenceler verir ama her yazmaya bir çoğunluk gidiş- dönüşü ekler. Sharding, kapasiteyi büyütür ama parçalar-arası sorguları ve işlemleri zorlaştırır ve iyi bir parça anahtarı seçme yükü getirir. Tek bir makinede çalışan basit bir veritabanı, çoğu zaman, dağıtık bir sistemden çok daha az karmaşıktır ve çok daha kolay akıl yürütülür. Bu yüzden bilge bir tasarımcı, dağıtımı bir varsayılan değil, gerçekten ihtiyaç doğduğunda başvurulan bir araç olarak görür.

12.12 Bu bölümün bıraktığı yer

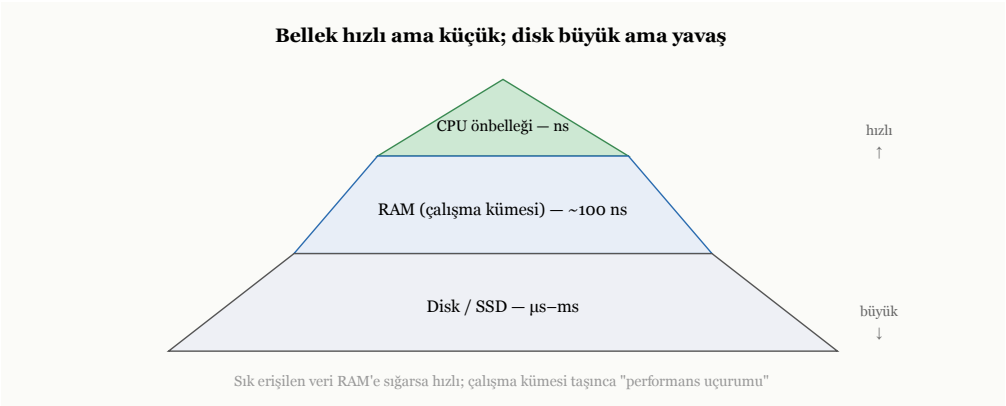
Bu bölümde, bir veritabanını tek makinenin ötesine taşıyan iki tekniği tanıdık. Replikasyonun aynı veriyi çoğaltarak erişilebilirlik ve okuma ölçeği sağladığını; lider-takipçi modelini, eşzamanlı ve eşzamansız replikasyonu, failover'ın tehlikelerini ve konsensüsün — çoğunluk mutabakatının — bu tehlikeleri nasıl çözdüğünü gördük. Sharding'in veriyi bölerek kapasite ve yazma ölçeği kazandırdığını; parça anahtarını, bölümleme stratejilerini, yönlendirmeyi, dağıt-topla örüntüsünü ve iyi bir anahtar seçmenin önemini öğrendik. Ve ikisinin büyük sistemlerde nasıl birleştirildiğini gördük.

Şimdiye dek hep birden çok makine arasındaki koordinasyondan söz ettik. Ama her bir makinenin, kendi içinde de çözmesi gereken bir kaynak yönetimi sorunu vardır. Beşinci bölümde değinmiştik: bellek hızlı ama küçük ve uçucu, disk yavaş ama büyük ve kalıcıdır. Tek bir düğüm, hangi veriyi bellekte tutacağına, hangisini diske bırakacağına nasıl karar verir; sınırlı belleğini en çok işe yarayacak veriyle nasıl doldurur? Bir sonraki bölümde, bu sessiz ama performansı belirleyen dengeye — bellek, önbellek ve disk arasındaki ödünleşime — eğiliyoruz.

Bölüm 13

Bellek, Önbellek ve Disk Ödünleşimi

Önceki iki bölümde, bir veritabanını birçok makineye yaymanın koordinasyon sorunlarıyla uğraştık. Şimdi bakışımızı yeniden tek bir makineye, tek bir düğümün içine çeviriyoruz; çünkü her düğüm, kendi içinde de çözmesi gereken bir kaynak yönetimi sorunu taşır. Beşinci bölümde tohumlamıştık: bellek hızlı ama küçük ve uçucu, disk yavaş ama büyük ve kalıcıdır. Bir düğümün performansının büyük bölümü, tek bir karara dayanır: hangi verinin bellekte tutulacağı, hangisinin diske bırakılacağı. Bu bölüm, o sessiz ama belirleyici dengeyi — bellek, önbellek ve disk arasındaki ödünleşimi — ele alıyor.



Şekil 13.1: Bellek hiyerarşisi ve çalışma kümesi: hız ile boyutun ödünleşimi.

13.1 Temel asimetri ve oyunun özü

Önce, kitap boyunca birkaç kez değindiğimiz asimetriyi netleştirelim, çünkü bu bölümdeki her şey ondan doğar. Bellek çok hızlıdır ama pahalıdır, sınırlıdır ve elektrik gidince içeriğini kaybeder. Disk çok yavaştır ama ucuzdur, büyüktür ve kalıcıdır. Bu ikisini, bir masa ile bir depoya benzetebilirsiniz: masanız (bellek) küçüktür ama üzerindeki her şeye anında uzanırsınız; depo (disk) kocamandır ama oradan bir şey getirmek zaman alır. Gerçekte bu hiyerarşi iki katmandan da ibaret değildir: en tepede, işlemcinin hemen yanındaki çok küçük ama nanosaniye ölçeğinde hızlı önbellekler (CPU önbelleği) bulunur; altında ana bellek, en altta da disk gelir. Her kademede, bir öncekine göre daha büyük ama daha yavaş bir depolama vardır. Bizim için asıl belirleyici olan, bu hiyerarşinin bellek ile disk arasındaki en keskin basamağıdır.

Oyunun özü şudur: çalışmak için ihtiyaç duyduğunuz şeyleri masanın üstünde tutmak, gerisini depoda bırakmak. Bir veritabanının tek bir düğümdeki hızı, asıl olarak bu kararı ne kadar iyi verdiğine bağlıdır. Doğru veri

bellekteyse, sistem uçar; yanlış veri bellekteyse ve aradığınız sürekli diskten getirilmek zorundaysa, sistem sürünür. Bütün mesele, sınırlı belleği en çok işe yarayacak veriyle doldurmaktır.

Bu kademelerin ne kadar farklı hızlarda olduğunu somutlaştırmak, sezgiyi keskinleştirir. İşlemcinin yanındaki birinci düzey önbellekten bir değer okumak, nanosaniyenin küçük bir kesri sürer; ana bellekten bir okuma, bunun yaklaşık iki yüz katıdır ama yine de yüz nanosaniye mertebesindedir. Buradan sonra uçurum başlar: katı hal diskinden (SSD) rastgele bir blok okumak mikrosaniyeler — yani ana bellekten yüzlerce kat yavaş — alır; dönen bir sabit diskten (HDD) rastgele bir okuma ise milisaniyeler, yani ana bellekten **on binlerce** kat yavaştır, çünkü disk kafasının doğru iza kayması fiziksel zaman ister. Bu büyüklük farklarını ölçeklendirmek için sık kullanılan bir benzetme şudur: bellek erişimini bir saniye sayarsak, bir SSD okuması birkaç dakika, bir HDD okuması ise saatler mertebesine karşılık gelir. Bir veritabanı tasarımının neredeyse her kararı, işte bu uçurumun yanlış tarafına düşmemek üzerine kuruludur.

13.2 Çalışma kümesi ve uçurum etkisi

Bu kararı anlamanın anahtarı, **çalışma kümesi** (working set) kavramıdır. Herhangi bir anda, verinin yalnızca bir kısmı "sıcaktır" — yani etkin biçimde kullanılmaktadır. Bir e-ticaret sitesinde, o anki kampanyadaki ürünler, son siparişler, aktif kullanıcılar sıcaktır; yıllar önceki kayıtlar ise soğuk, nadiren dokunulan veridir. Çalışma kümesi, işte bu sıcak kısımdır.

Bu sezgisel "sıcak veri" kavramının arkasında, biçimsel bir kuram vardır. Çalışma kümesi, aslında belirli bir zaman penceresi içinde bir sürecin dokunduğu farklı veri birimlerinin (sayfaların) kümesi olarak tanımlanır ve bu tanım, bellek yönetimi literatürünün temel taşlarından birinde ortaya konmuştur.[1] Bu kuramın can alıcı gözlemi, çoğu iş yükünün **yerellik** (locality) sergilediğidir: programlar gelişigüzel her yere değil, belirli bir süre boyunca verinin dar bir bölgesine yoğunlaşır. Bu yüzden çalışma kümesi genelde tüm veriden çok daha küçüktür ve belleğe sığma şansı yüksektir. Pencere ne kadar geniş seçilirse çalışma kümesi o kadar büyür; doğru pencere, "şu an gerçekten gerekli olanı" yakalayacak kadar geniş, ama soğuk geçmişi de içine çekecek kadar geniş olmayanıdır. Bir sistemi boyutlandırmak, aslında bu çalışma kümesini ölçüp ona yetecek belleği sağlamak demektir.

Buradan çarpıcı bir gerçek doğar: performans, kademeli bir eğim değil, bir **uçurumdur**. Çalışma kümeniz belleğe sığdığı sürece, sistem hızlıdır — ihtiyaç duyduğunuz her şey zaten masanın üstündedir. Ama çalışma kümesi belleği biraz olsun aşmaya başladığında, sistem aniden çöker; çünkü artık sürekli olarak depoya koşmak, bir şeyi getirip masaya koymak için başka bir şeyi kaldırmak, sonra onu da yeniden istemek zorunda kalırsınız. Bu sürekli oraya buraya taşıma durumuna **debelenme** (thrashing) denir ve performansı dramatik biçimde düşürür. Bu yüzden bir veritabanını boyutlandırırken sorulacak en kritik soru, "tüm verim belleğe sığar mı" değil, "**çalışma kümem** belleğe sığar mı" sorusudur.

13.3 Önbellek: belleği akıllıca kullanmak

Bir düğüm, sıcak veriyi bellekte tutmak için bir **önbellek** (cache) işletir. Mantığı basittir. Bir veri istendiğinde, sistem önce önbelleğe bakar. Veri orada varsa — buna "isabet" (hit) denir — hızlıca, diske hiç gitmeden döndürülür. Yoksa — buna "ıska" (miss) denir — diskten getirilir, kullanıcıya verilir ve gelecekte yine istenebileceği umuduyla önbelleğe yerleştirilir. Önbellek, böylece uygulama ile yavaş disk arasında bir tampon görevi görür; sık istenen veriyi belleğe çekerek disk erişimini en aza indirir.

Önbellek tek bir katmanda olmak zorunda değildir. Sistem, ham disk bloklarını önbellekleyebilir — böylece aynı bloğu tekrar tekrar diskten okumaktan kurtulur. Ya da çözülmüş, kullanıma hazır belgeleri önbellekleyebilir — böylece aynı belgeyi her seferinde yeniden ayrıştırmaktan kurtulur. Ayrıca indeks yapıları da bellekte tutulur, çünkü onlar neredeyse her sorguda kullanılır. Farklı katmanları önbelleklemek, farklı türden tekrar eden maliyetleri ortadan kaldırır. Üçüncü kısımda OxiDB'nin hem çözülmüş belgeler hem de ham baytlar için ayrı önbellekler tuttuğunu göreceğiz.

[1] Peter J. Denning, "The Working Set Model for Program Behavior," *Communications of the ACM*, 11(5), 1968.

13.4 Tahliye: yer açmak için ne atılır

Önbellek sonludur; eninde sonunda dolar. Dolduğunda, yeni bir şeyi içeri almak için eskilerden birini **atmak** (tahliye etmek) gerekir. Hangi verinin atılacağına karar veren kurala **tahliye politikası** denir ve bu, önbelleğin ne kadar işe yarayacağını doğrudan belirler.

En yaygın politika, "en uzun süredir dokunulmayanı at" ilkesidir — kısaca LRU (least recently used). Mantığı bir bahse dayanır: yakın zamanda kullandığınız bir şeyi, yine yakında kullanma olasılığınız yüksektir; çok uzun süredir dokunmadığınız bir şeyi ise muhtemelen bir süre daha kullanmayacaksınız. Bu yüzden LRU, en uzun süredir boşta duranı kurban seçer. Bir kütüphanecinin, sık istenen kitapları el arabasında yakınında tutup, aylardır kimsenin sormadıklarını rafa kaldırmasına benzer.

LRU'nun saf hali, gerçekte küçük bir maliyet taşır: her erişimde, dokunulan öğeyi "en yeni" konuma taşımak için bir kayıt güncellenmelidir. Bunu ucuzlatan zarif bir yaklaşım, **CLOCK** (saat) algoritmasıdır. CLOCK, öğeleri bir halka üzerinde dizer ve her birine tek bitlik bir "kullanıldı" işareti verir. Bir öğeye dokunulduğunda yalnızca bu bit 1 yapılır — sıralamayı yeniden düzenlemek gerekmez. Yer açmak gerektiğinde, bir ibre halka üzerinde döner: bitini 1 bulduğu öğeyi atmaz, ona bir şans daha tanıyıp bitini sıfırlar ve ilerler; bitini 0 bulduğu ilk öğeyi kurban seçer. Böylece CLOCK, LRU'nun davranışını yaklaşık olarak taklit eder ama erişim başına neredeyse hiç maliyet taşımaz; bu yüzden işletim sistemlerinin sayfa değiştirme mekanizmalarında yaygın biçimde kullanılır.

LRU'nun ölçtüğü tek şey **yakınlıktır** — bir öğeye en son ne zaman dokunulduğu. Buna karşı duran felsefe **LFU**'dur (least frequently used, en az sıklıkla kullanılan): her öğenin **kaç kez** istendiğini sayar ve en seyrek isteneni atar. LFU, gerçekten popüler olan ama her an dokunulmayan veriyi korumakta iyidir; ama iki zayıflığı vardır. Birincisi, bir zamanlar çok istenip artık soğumuş bir öğe, geçmiş sayacı yüksek olduğu için önbellekte gereksiz yere tutunabilir. İkincisi, sıklığı saymanın LRU'dan daha pahalı bir defter tutması vardır. İşte buradan, ikisini birleştirme arayışı doğar.

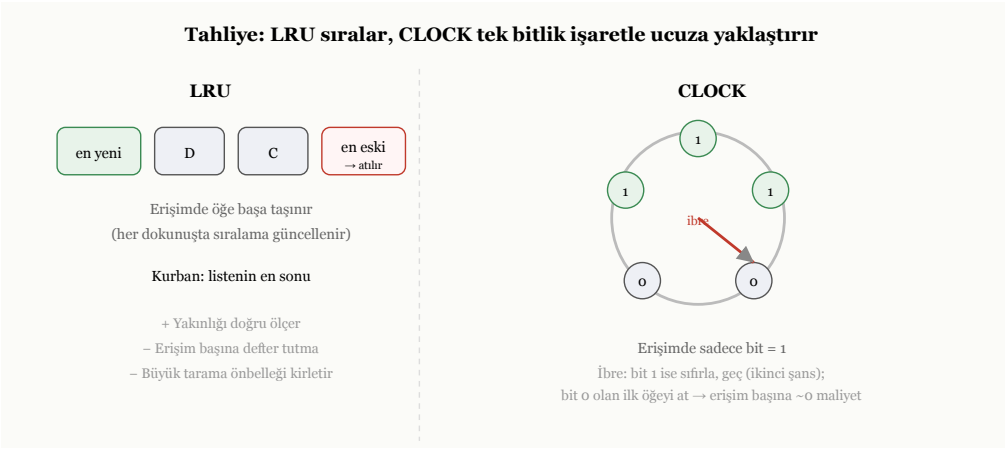
Bu arayışın olgun bir ürünü, **ARC** (adaptive replacement cache, uyarlamalı değiştirme önbelleği) adlı politikadır.[2] ARC'ın inceliği, yakınlık ile sıklığı **uyarlamalı** olarak dengelemesidir. Önbelleği zihinsel olarak iki listeye böler: yalnızca bir kez görülen ("yeni gelen") öğeler ve birden çok kez görülen ("kanıtlanmış sıcak") öğeler. Dahası, yakın geçmişte attığı öğelerin kimliklerini — verisini değil, yalnızca hangi öğe olduğunu — bir "hayalet liste"de tutar. Az önce attığı bir öğe hemen geri istenirse, bunu bir hata olarak algılar ve ilgili listeye ayırdığı payı büyütür. Böylece ARC, iş yükü sıklık-ağırlıklıysa o yöne, yakınlık-ağırlıklıysa diğer yöne kendiliğinden kayar; hiçbir elle ayar gerektirmez. Bunun bir yan faydası, ARC'ın **tarama-dirençli** olmasıdır — çünkü büyük bir taramanın ürettiği "yalnızca bir kez görülen" öğeler, kanıtlanmış sıcak öğeleri tutan listeye hiç dokunamaz.

Bu, doğrudan LRU'nun en bilinen zayıflığına götürür. Büyük bir tarama düşünün — örneğin tüm koleksiyonu bir kez baştan sona okuyan bir toplama sorgusu. Bu tarama, gerçekte sıcak olmayan bir sürü veriyi önbelleğe doldurur ve bu sırada asıl sıcak veriyi dışarı atar. Tarama bittiğinde önbellek, bir daha kullanılmayacak soğuk veriyle dolu, asıl ihtiyaç duyulan sıcak veriden ise yoksun kalmıştır. Buna **önbellek kirlenmesi** denir. ARC gibi politikaların tarama-direnci, tam da bu kirlenmeyi önlemeyi hedefler: bir kez görülen tarama verisini, kanıtlanmış sıcak veriyi koruyan bölmeden ayrı tutar. Temel ders şudur: iyi bir tahliye politikası, yalnızca "ne zaman kullanıldı" değil, "gerçekten sıcak mı" sorusunu da gözetmeye çalışır.

13.5 Önbelleğin kendi maliyeti ve sınırlama zorunluluğu

Önbellek bedava bir kazanç değildir; kendi maliyetleri vardır. Önbelleğe ayrılan bellek, başka hiçbir işe ayrılamayan bellektir. Önbelleği yönetmenin — neyin ne zaman kullanıldığını takip etmenin — kendi küçük yükü vardır. Ve en tehlikelisi: eğer bir önbellek sınırsız büyümeye bırakılırsa, tüm belleği yiyip bitirebilir ve sistemi belleksiz bırakıp çökertebilir.

[2] Nimrod Megiddo ve Dharmendra S. Modha, "ARC: A Self-Tuning, Low Overhead Replacement Cache," *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)*, 2003.



Şekil 13.2: Önbellek tahliyesi: LRU sıralamayı yeniden düzenler; CLOCK ise tek bitlik "kullanıldı" işaretiyle ucuza yaklaştırır.

Bu yüzden ciddi sistemler, önbelleklerini bir **bellek bütçesiyle sınırlar**: önbellek, belirli bir boyutu aşamaz; o boyuta ulaştığında, yeni bir şey almak için mutlaka eskisini atar. Böylece bellek kullanımı öngörülebilir kalır ve denetimden çıkmaz. Üçüncü kısımda OxiDB'nin, bellek kullanımının kontrolden çıkmasını önlemek için önbelleklerini tam da böyle sabit bir bütçeyle sınırladığını ve bunun bellek ayak izini nasıl dizginlediğini göreceğiz.

13.6 İki felsefe: belleğe öncelikli ve diske öncelikli

Beşinci bölümde, depolama motorlarının iki felsefesini görmüştük; bellek-disk ilişkisinde de benzer bir ikilik vardır ve doğrudan o bölümle bağlanır.

Birinci felsefe **belleğe önceliklidir**: tüm veriyi bellekte tutmak, diski yalnızca kalıcılık için — yani anlık görüntüler ve günlük için — kullanmak. Bu yaklaşımda her okuma, zaten bellekte olan veriye dokunduğu için olağanüstü hızlıdır. Bedeli, belleğin **veriyle birlikte büyümesidir**: veri ikiye katlanınca, gereken bellek de ikiye katlanır. Bu, küçük ve orta ölçekli veri için harikadır, ama veri büyüdükçe bellek hem pahalı hem de bir tavana dayanan bir kısıt haline gelir. Üçüncü kısımda OxiDB'nin varsayılan kipinin tam da bu belleğe öncelikli yaklaşım olduğunu göreceğiz.

İkinci felsefe **diske önceliklidir**: bellekte yalnızca küçük bir şey — örneğin bir kimlik-konum dizini ve sıcak bir alt küme — tutmak, verinin asıl gövdesini diskte bırakmak ve gerektiğinde oradan getirmek. Bu yaklaşımda bellek kullanımı, veriyle birlikte büyümez; öngörülebilir ve düşük kalır. Bedeli, soğuk veriye erişimin diske gitmeyi, yani bir gecikmeyi göze almasıdır. Bu, belleğin değerli ya da verinin çok büyük olduğu durumlar için biçilmiş kaftandır. Üçüncü kısımda OxiDB'nin "disk-öncelikli" kipinin tam olarak bunu yaptığını — taze açılışta tüm veri yerine yalnızca küçük bir dizinin bellekte durduğunu — ve bunun bellek ayak izini nasıl dramatik biçimde küçülttüğünü ayrıntısıyla göreceğiz.

13.7 Tampon havuzu: veriyi sayfa sayfa yönetmek

Önbelleğin klasik, açıkça yönetilen biçimine veritabanı dünyasında **tampon havuzu** (buffer pool) denir. Mantığı, diski sabit boyutlu bloklara — **sayfalara** (page) — bölmek ve bellekte bu sayfalardan belirli bir sayıda tutabilen bir havuz ayırmaktır. Bir sayfa istendiğinde, havuzda varsa doğrudan oradan verilir; yoksa diskten okunup havuzdaki bir çerçeveye yerleştirilir, gerekiyorsa bir başka sayfa tahliye edilerek. Tampon

havuzunun sayfa temelli olması bir tesadüf değildir: disk zaten blok blok okunup yazıldığı için, önbelleği de aynı birimle yönetmek, getirme ve geri yazma işlemlerini doğal olarak hizalar.

Tampon havuzunun, bizi altıncı bölüme bağlayan kritik bir görevi vardır: **kirli** (dirty) sayfaları yönetmek. Bellekte değiştirilmiş ama henüz diske yazılmamış bir sayfa kirlidir. Tampon havuzu, bu sayfayı hemen diske yazmaz — çünkü aynı sayfa kısa süre içinde tekrar değişebilir; bunun yerine yazmaları biriktirip toptan, verimli biçimde diske akıtır. Ama bir kirli sayfayı tahliye etmeden önce mutlaka diske yazması gerekir, yoksa değişiklik kaybolur. Burada önce-yaz günlüğüyle (WAL) ince bir kural devreye girer: bir veri sayfası diske yazılmadan önce, onu tarif eden günlük kaydının diske inmiş olması gerekir — böylece çökme sonrası kurtarma her zaman tutarlı bir noktadan başlayabilir. Yani tampon havuzunun tahliye kararı, yalnızca bir performans meselesi değil, aynı zamanda dayanıklılığın bir parçasıdır.

13.8 İşletim sistemine güvenmek: belleğe yansıtma

Bellek yönetimini elle yapmanın zarif bir alternatifi vardır ve beşinci bölümde ona değinmiştik: bir dosyayı **belleğe yansıtmak** (mmap). Bu teknikte, dosyaya erişmek bellekteki bir adrese erişmek kadar basit hale gelir ve verinin diskten belleğe ne zaman getirileceğine, bellek darda kaldığında neyin geri atılacağına işletim sistemi karar verir.

Bu sihrin altındaki düzenek, **sayfa hatasıdır** (page fault). Dosya belleğe yansıtıldığında, aslında hiçbir veri hemen okunmaz; yalnızca bir adres aralığı "söz verilmiş" olur. Programınız bu aralıktan henüz bellekte olmayan bir adrese ilk kez dokunduğunda, işlemci bunu fark eder ve denetimi işletim sistemine devreder — işte bu kesintiye sayfa hatası denir. İşletim sistemi sessizce devreye girer, ilgili sayfayı diskten okuyup belleğe yerleştirir ve programınız hiçbir şey olmamış gibi kaldığı yerden devam eder. İlk dokunuş, gizliden gizliye bir disk okuması ödetir; sonraki dokunuşlar ise, sayfa artık bellekte olduğu için, sıradan bellek hızındadır. Bu mekanizma, "lazım olduğunda getir" (demand paging) ilkesinin ta kendisidir ve veritabanının açıkça hiçbir okuma çağrısı yapmadan diski tembelce, erişim düzenine göre belleğe çekmesini sağlar.

Bunun güzelliği, önbellek yönetiminin büyük bölümünü işletim sistemine **devretmesidir**. İşletim sistemi zaten dosya sayfalarını bellekte tutan, sıcakları saklayıp soğukları atan olgun bir mekanizmaya sahiptir; mmap, bu mekanizmadan doğrudan yararlanır. Sıcak dosya sayfaları bellekte kalır, soğuklar otomatik olarak atılır ve önemlisi, bunlar disk üzerinde bir kopyaya sahip oldukları için bellek baskı altındayken **geri alınabilir** — yani gerektiğinde serbest bırakılabilir.

Ama bu kolaylığın bir inceliği vardır ve onu görmek önemlidir. Belleğe yansıtılan sayfalar, dokunulduklarında bellekte yer kaplar ve sistemin "bu süreç ne kadar bellek kullanıyor" ölçümüne dahil olur. Bu yüzden, diske öncelikli, mmap'e dayanan bir sistemin bellek kullanımı yanıltıcı görünebilir: taze açılışta çok düşükken, tüm veriye dokunan büyük bir taramadan sonra yükselebilir — çünkü işletim sistemi o sayfaları belleğe çekmiştir. Ama bu yükselen bellek, geri alınabilir bir bellektir; baskı altında işletim sistemi onu sessizce serbest bırakır. "Veritabanım ne kadar bellek kullanıyor" sorusunun, sandığınızdan çok daha incelikli olmasının bir nedeni budur: kalıcı (anonim) bellek ile geri alınabilir (dosya destekli) belleği ayırmak gerekir. Üçüncü kısımda OxiDB'nin disk-öncelikli kipinde tam da bu olguyu — taze açılışta çok düşük, tam iş yükünden sonra yükselen ama geri alınabilir bellek — ölçümlerle göreceğiz.

13.9 Sıkıştırma: yer, işlemci ve sıfır-kopya üçgeni

Belleği ve diski yönetmenin bir aracı daha vardır: **sıkıştırma**. Veriyi sıkıştırarak saklamak, hem diskte hem bellekte daha az yer kaplamasını sağlar; böylece aynı belleğe daha çok veri sığar, çalışma kümesinin sığma olasılığı artar. Ama sıkıştırma bedava değildir: sıkıştırılmış veriyi okumak için, her erişimde onu **açmak** gerekir ve bu, işlemci zamanı harcar.

Burada üç yönlü bir ödünleşim belirir: yer, işlemci ve bir üçüncü incelik. Veri sıkıştırılmamışsa ve şifrelenmemişse, onu okumak için hiçbir dönüşüm gerekmez; bellekteki ham baytlara doğrudan, hiç kopyalamadan

dokunulabilir — buna **sıfır-kopya** erişim denir ve son derece hızlıdır. Veri sıkıştırılmışsa, bu sıfır-kopya olanağı kaybolur; her okuma bir açma ve bir kopyalama gerektirir. Dolayısıyla sıkıştırma, yer kazandırırken okuma hızından ödün verebilir. Hangisinin kazandığı, veriye bağlıdır: gerçekten sıkışan, çok okunmayan veri için sıkıştırma kazandırır; az sıkışan ama çok ve hızlı okunan veri için sıkıştırmamak daha iyidir. Üçüncü kısımda OxiDB'nin disk-öncelikli kipinde, sıkıştırılmış ve sıkıştırılmamış depolama arasındaki bu tam ödünleşimi ölçtüğünü ve belirli yüklerde sıkıştırmayı kapatmanın taramaları nasıl hızlandırdığını göreceğiz.

13.10 Bu bölümün bıraktığı yer

Bu bölümde, tek bir düğümün performansını belirleyen sessiz dengeyi — bellek, önbellek ve disk arasındaki ödünleşimi — inceledik. Bellek ile diskin temel asimetrisini ve kademeler arasındaki büyüklük farklarını; çalışma kümesi kavramını, onun biçimsel tanımını ve belleğe sığıp sığmamasının yarattığı uçurum etkisini; önbelleğin nasıl çalıştığını ve tahliye politikalarının (LRU, CLOCK, LFU, ARC ve önbellek kirlenmesinin) inceliklerini; tampon havuzunun sayfa temelli yönetimini ve kirli sayfa kuralını; önbelleğin kendi maliyetini ve sınırlama zorunluluğunu; belleğe öncelikli ve diske öncelikli iki felsefeyi; belleğe yansıtmayı, sayfa hatası düzeneğini ve bellek ölçümünün inceliğini; ve sıkıştırmanın yer-işlemci-sıfırkopya üçgenini gördük.

Böylece Kısım II'nin neredeyse tamamını — bir belge veritabanının içeride nasıl çalıştığını — tamamlamış olduk: veriyi sakladık, dayanıklı kıldık, indeksledik, sorguladık, özetledik, tutarlı tuttuk, ölçeklendirdik ve belleğini yönettik. Geriye, OxiDB'ye geçmeden önce ele alınması gereken son bir genel konu kaldı ve o, belki de en çok ihmal edilenidir: tüm bu veriyi **korumak**. Kim okuyabilir, kim yazabilir; veri yetkisiz ellere geçerse ne olur; kimin ne yaptığı nasıl kaydedilir? Bir sonraki bölümde, bir veritabanının güvenlik boyutuna — kimlik doğrulamaya, yetkilendirmeye, şifrelemeye ve denetime — eğilerek Kısım II'yi kapatıyoruz.

Bölüm 14

Güvenlik: Kimlik Doğrulama, Yetkilendirme, Şifreleme ve Denetim

Kısım II boyunca, bir belge veritabanının içeride nasıl çalıştığını katman katman açtık: veriyi sakladık, dayanıklı kıldık, indeksledik, sorguladık, özetledik, tutarlı tuttuk, ölçeklendirdik ve belleğini yönettik. Ama tüm bu emek, gözden kaçırılması kolay ama hayati bir şeyi varsayar: veriye yalnızca yetkili kişilerin, yalnızca yetkili biçimde eriştiğini. Veri değerlidir ve çoğu zaman hassastır; bu da onu bir hedef yapar. Bu bölüm, bir veritabanının güvenlik boyutunu — verinin kim tarafından, nasıl erişilebileceğini ve çalınsa bile korunmasını sağlayan mekanizmaları — ele alarak Kısım II'yi kapatıyor.

Şekil 14.1: Güvenliğin dört katmanı: kimlik, yetki, şifreleme, denetim.

14.1 Tehdit ve katmanlı savunma ilkesi

Güvenliği düşünmenin ilk adımı, neye karşı korunduğumuzu görmektir. Tehdit, hem dışarıdan gelir — sisteme yetkisiz girmeye çalışan saldırganlar — hem de içeriden: yetkisini kötüye kullanan ya da dikkatsizce davranan kullanıcılar. Üstelik veritabanı, bir kuruluşun en değerli verisini tek bir yerde topladığı için, özellikle çekici bir hedeftir.

Bu yüzden güvenlikte tek bir savunma hattına güvenmek tehlikelidir. Asıl ilke, **katmanlı savunmadır** (defense in depth): birbirini tamamlayan birden çok koruma katmanı kurmak, böylece bir katman aşılsa bile diğerlerinin devrede kalması. Bu bölümde göreceğimiz dört mekanizma — kimlik doğrulama, yetkilendirme,

şifreleme ve denetim — bu katmanların başlıcalarıdır. Bunları, korumalı bir binaya benzetebiliriz: kapıda kimlik kontrolü (kimlik doğrulama), içeride hangi odalara girebileceğinizi belirleyen kartlı geçiş (yetkilendirme), değerli eşyaların kilitli kasada durması (şifreleme) ve her hareketi kaydeden güvenlik kameraları (denetim). Hiçbiri tek başına yeterli değildir; güçlü güvenlik, hepsinin birlikte çalışmasından doğar.

14.2 Kimlik doğrulama: sen kimsin

İlk katman **kimlik doğrulamadır** (authentication) ve tek bir soruyu yanıtlar: "Sen kimsin?" Sisteme erişmek isteyen birinin, iddia ettiği kişi olduğunu kanıtlaması gerekir. Bunu kanıtlamanın en yaygın yolu paroladır; ama parolaların güvenli işlenmesi, sandığınızdan daha incelikli bir iştir.

En temel kural şudur: **parolalar asla düz metin olarak saklanmaz.** Eğer bir veritabanı kullanıcıların parolalarını olduğu gibi tutarsa, o veritabanı bir gün sızdırıldığında — ki sızıntılar olur — tüm parolalar bir çırpıda ele geçer. Bunun yerine, parolanın tek yönlü bir dönüşümü, yani bir **özeti** (hash) saklanır. Bu dönüşüm tek yönlüdür: paroladan özet hesaplanabilir, ama özetten parola geri elde edilemez. Kullanıcı giriş yaptığında, girdiği parolanın özeti hesaplanır ve saklanan özetle karşılaştırılır; ikisi uyuşuyorsa parola doğrudur. Böylece veritabanı sızsa bile, ortaya yalnızca geri çevrilemez özetler çıkar.

Bu temel fikir, iki ek önlemle güçlendirilir. Birincisi **tuzlama** (salting): her parolaya, ona özgü rastgele bir değer eklenip öyle özetlenir. Tuzlama, saldırganların önceden hesaplanmış dev özet tablolarıyla — sözde **gökkuşağı tabloları** (rainbow tables) — parolaları toplu halde çözmesini engeller; çünkü her parolanın tuzu farklı olduğu için, önceden hesaplanmış hiçbir tablo işe yaramaz ve saldırgan, ne kadar uğraşırsa uğraşsın her parolayı **ayrı ayrı** kırmaya zorlanır. Tuz gizli olmak zorunda değildir; özetin yanında açıkça saklanabilir, çünkü değeri gizlilikten değil, benzersizlikten gelir. Tuzun bir akrabası **biberdir** (pepper): tüm parolalara eklenen, ama veritabanından **ayrı** bir yerde — örneğin uygulamanın yapılandırmasında — tutulan gizli bir değer. Biber, veritabanı tek başına sızdığında saldırganın elindekini işe yaramaz kılar, çünkü biberi bilmeden hiçbir tahmini doğrulayamaz.

İkincisi, özet fonksiyonunu **bilinçli olarak yavaş** seçmektir. Sıradan bir kullanıcı için tek bir parolayı saniyenin küçük bir kesrinde doğrulamak yeterlidir; ama bir saldırgan, milyarlarca olası parolayı deneyerek kaba kuvvetle kırmaya çalışır. Özet hesaplamasını yavaş yapmak, meşru girişi neredeyse hiç etkilemezken, kaba kuvvet saldırısını pratikte imkânsız hale getirir. Bu yavaşlığın denetlenebilir miktarına **iş faktörü** (work factor) denir: bir ayarla, özetin kaç tur yineleneceğini büyütüp küçültebilirsiniz. Donanım her yıl hızlandığı için iş faktörü, zamanla yukarı çekilmesi gereken bir kadrandır — bugün güvenli bir maliyet, on yıl sonra zayıf kalacaktır.

Bu amaçla tasarlanmış birkaç özel fonksiyon kuşağı vardır ve aralarındaki fark öğreticidir. İlk kuşak, yalnızca **işlemci zamanını** pahalı kılar — yani saf yineleme. Ama saldırganlar, parola tahminlerini binlerce çekirdekli grafik işlemcilerde (GPU) ya da özel donanımda koşturarak bu maliyeti devasa ölçüde paralelleştirebilir. Buna karşı geliştirilen ikinci kuşak, fonksiyonu **bellek açısından da pahalı** yapar: hesaplamak için büyük miktarda bellek gerektirir, böylece her bir paralel kopya çok bellek isteyeceği için, ucuz paralel donanım avantajını yitirir.[1] Bu fikri en olgun haline taşıyan, açık bir yarışmayla seçilmiş güncel standart, hem işlemci hem bellek hem de paralellik derecesini ayrı ayrı ayarlanabilir kılan bir fonksiyondur.[2] Üçüncü kısımda OxiDB'nin parolaları, tam da böyle bellek-zoru bir fonksiyonla tuzlayıp özetlediğini göreceğiz.

Bir incelik daha vardır. Parolanın kendisini ağ üzerinden göndermek tehlikelidir; çünkü hattı dinleyen biri onu yakalayabilir. Bu yüzden olgun sistemler, parolayı hiç göndermeden, kullanıcının onu bildiğini kanıtlamasını sağlayan **meydan-okuma yanıt** (challenge-response) yöntemleri kullanır: sunucu bir soru sorar, kullanıcı parolasını kullanarak ona doğru yanıtı üretir, ama parolanın kendisi hattan hiç geçmez.

Bu fikrin yaygın ve özenle tasarlanmış bir somutlaşması, **SCRAM** adlı standartlaştırılmış protokoldür.[3] SC-

[1] Colin Percival, "Stronger Key Derivation via Sequential Memory-Hard Functions," *BSDCan*, 2009 (scrypt).

[2] Alex Biryukov, Daniel Dinu ve Dmitry Khovratovich, "Argon2: New Generation of Memory-Hard Functions for Password Hashing and Other Applications," *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2016.

[3] Abhijit Menon-Sen vd., "Salted Challenge Response Authentication Mechanism (SCRAM) SASL and GSS-API Mechanisms," *IETF RFC 5802*, 2010.

RAM'ın iç işleyişine yakından bakmak öğreticidir, çünkü tek bir mekanizmada bu bölümdeki birçok fikri birleştirir. El sıkışma dört iletiyle yürür. İstemci işe, bir kullanıcı adı ve kendi ürettiği rastgele bir değerle — **istemci çentiği** (client nonce) — başlar. Sunucu yanıtında, kullanıcıya özgü **tuzu** ve daha önce sözünü ettiğimiz **iş faktörünü** (yineleme sayısını), bir de kendi rastgele değerini ekleyerek geri gönderir. Şimdi her iki taraf da, parolayı tuzla ve iş faktörüyle yavaşça özetleyip aynı gizli anahtarı türetir; ama bu anahtarı doğrudan göndermek yerine, iki çentiği de içeren ortak bir metni o anahtarla işleyerek bir **kanıt** üretir ve yalnızca kanıtı yollar. Sunucu da kendi sakladığı bilgilerle aynı kanıtı hesaplar ve karşılaştırır; uyuşuyorsa istemci parolayı biliyordur. Son iletide sunucu, kendisinin de doğru anahtarı bildiğini gösteren bir sunucu imzası yollayarak **karşılıklı** kimlik doğrulamayı tamamlar — yani yalnızca sunucu istemciyi değil, istemci de sunucuyu doğrular.

Şekil 14.2: SCRAM el sıkışması: dört ileti boyunca parola hattan hiç geçmez; her iki taraf birbirini doğrular.

SCRAM'ın bu tasarımının üç güzel özelliği vardır. Birincisi, parola hattan hiç geçmez. İkincisi, her iki tarafın da ürettiği rastgele çentikler sayesinde, bir saldırganın eski bir oturumu kaydedip sonradan yeniden oynatması (replay) işe yaramaz — her el sıkışma benzersizdir. Üçüncüsü, sunucunun sakladığı veri çalınsa bile, saldırgan onunla doğrudan giriş yapamaz; çünkü asıl kanıtı üretmek için yine de parolayı bilmesi gerekir. Üçüncü kısımda OxiDB'nin tam da bu SCRAM protokolünü kullandığını ve parolaları yavaş, tuzlanmış bir özetle sakladığını göreceğiz. Parolaların ötesinde, sertifikalar ya da kuruluşun merkezi kimlik sistemine bağlanma gibi daha gelişmiş yöntemler de vardır; üçüncü kısımda OxiDB'nin bunların hangilerini desteklediğini, hangilerinin henüz eksik olduğunu dürüstçe ele alacağız.

14.3 Yetkilendirme: ne yapabilirsin

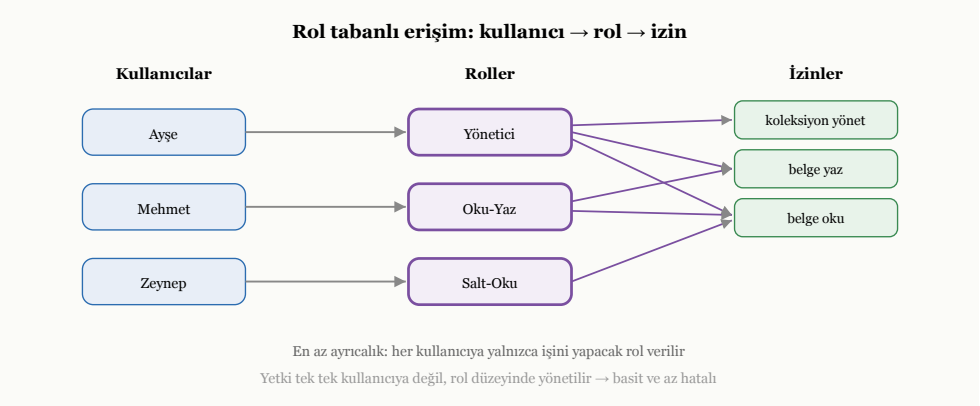
Kimliği doğrulanmış olmak, her şeyi yapabilmek demek değildir. İkinci katman **yetkilendirmedir** (authorization) ve farklı bir soruyu yanıtlar: "Sen kimsin" değil, "Sen ne yapabilirsin?" Kimliği bilinen bir kullanıcının, hangi veriye erişebileceğine ve hangi işlemleri yapabileceğine karar verir.

Yetkilendirmenin temel ilkesi, **en az ayrıcalıktır** (least privilege): her kullanıcıya, işini yapması için gereken en az yetkiyi vermek, fazlasını vermemek. Bir raporu yalnızca okuması gereken birine yazma yetkisi vermek, gereksiz bir risktir; o hesap ele geçirilirse, verebileceği zarar yetkisiyle sınırlıdır. En az ayrıcalık, olası bir ihlalin etki alanını daraltır.

Yetkileri tek tek her kullanıcıya atamak, çok sayıda kullanıcıda yönetilemez hale gelir. Bu yüzden yaygın yaklaşım, **rol tabanlı erişim denetimidir** (RBAC). Yetkiler, anlamlı rollerde gruplanır — örneğin yalnızca okuyabilen bir rol, okuyup yazabilen bir rol, her şeyi yapabilen bir yönetici rolü — ve kullanıcılara bu roller atanır. Bir kullanıcının ne yapabileceği, sahip olduğu rolden gelir; yetkileri tek tek değil, rol düzeyinde yönetirsiniz. Bu, hem daha basit hem de daha az hata yapılan bir modeldir.

Rol tabanlı modelin de bir sınırı vardır. Roller, **statik** gruplardır: yetkiyi, kullanıcının hangi role ait olduğuna göre verir. Ama bazı kararlar, yalnızca "kim olduğuna" değil, **bağlamın özniteliklerine** bağlıdır — günün saati, isteğin geldiği ağ, belgenin bir alanının değeri ya da kullanıcının bir özelliği gibi. Bunları saf rollerle ifade etmek, "gündüz-okuyabilen-muhasebeci" türünden bir rol patlamasına yol açar. İşte bu noktada **öznitelik tabanlı erişim denetimi** (ABAC) devreye girer: yetkiyi sabit rollerle değil, kullanıcının, kaynağın ve ortamın özniteliklerini değerlendiren kurallarla verir. ABAC çok daha esnektir, ama bedeli karmaşıklıktır — kuralların doğru yazıldığını ve birbiriyle çelişmediğini güvence altına almak güçtür. Pratikte çoğu sistem, kaba taneli yetki için RBAC'ı, ince taneli ve bağlama duyarlı kararlar için ABAC benzeri kuralları **birlikte** kullanır.

Bu, yetkilendirmenin **ayrıntı düzeyi** boyutuna bağlanır. Erişim, tüm veritabanı düzeyinde verilebilir; ya da daha ince taneli olarak belirli koleksiyonlar düzeyinde; ya da en ince haliyle, **tek tek belgeler** düzeyinde — örneğin "bir kullanıcı yalnızca kendi belgelerini görebilir" gibi bir kural. Bu son tür kural aslında özniteliklere — belgenin sahip alanı ile isteği yapanın kimliğinin karşılaştırılmasına — dayandığı için, ABAC'ın belge düzeyindeki bir yüzüdür. Ayrıntı düzeyi inceldikçe, koruma güçlenir ama yönetim karmaşıklaşır. Üçüncü kısımda OxiDB'nin hem rol tabanlı bir erişim denetimi sunduğunu hem de belge düzeyinde, "bu belgeye yalnızca sahibi erişebilir" türünden kurallar tanımlamaya olanak verdiğini göreceğiz.



Şekil 14.3: Rol tabanlı erişim: yetkiler rollerde gruplanır, kullanıcılara roller atanır — yetki tek tek değil, rol düzeyinde yönetilir.

14.4 Şifreleme: çalınsa bile okunamaz

Kimlik doğrulama ve yetkilendirme, sisteme **meşru** yoldan erişimi denetler. Ama ya biri bu denetimleri tümüyle atlarsa — diski fiziksel olarak çalarsa ya da ağ trafiğini dinlerse? İşte üçüncü katman, **şifreleme**, tam da bu duruma karşı korur: veriyi öyle bir biçime sokar ki, anahtarı olmayan biri onu ele geçirse bile okuyamaz.

Şifrelemenin iki ayrı cephesi vardır. Birincisi **aktarım sırasında şifrelemedir**: veri ağ üzerinden giderken şifrelenir, böylece hattı dinleyen biri yalnızca anlamsız baytlar görür. Bu, özellikle veritabanına uzaktan bağlanan sistemlerde vazgeçilmezdir. İkincisi **dururken şifrelemedir** (at rest): veri diske şifrelenmiş olarak yazılır, böylece diski ya da dosyaları çalan biri onları okuyamaz. Beşinci ve altıncı bölümlerde verinin diske nasıl yazıldığını anlatmıştık; dururken şifreleme, o yazma katmanına eklenen bir dönüşümdür: veri diske inmeden şifrelenir, okunurken çözülür.

Şifrelemenin nasıl yapıldığı da inceliklidir. Çoğu sistem, bugün açık bir yarışmayla seçilmiş ve dünya çapında standartlaşmış bir blok şifresine dayanır.[4] Ama tek başına bir şifre, gizliliği sağlar; veriyi **gizler** ama onun **değiştirilmediğini** garanti etmez. Bir saldırgan, anahtarı bilmese bile şifreli baytları bozup veriyi

[4]National Institute of Standards and Technology, "Advanced Encryption Standard (AES)," *FIPS PUB 197*, 2001.

sessizce çarpıtabilir. Bunu önlemek için modern sistemler, şifrelemeyi ve bütünlük doğrulamasını tek bir adımda birleştiren **kimliği doğrulanmış şifreleme** (authenticated encryption, AEAD) kiplerini kullanır. Böyle bir kip, şifreli verinin yanına bir **doğrulama etiketi** ekler; çözme sırasında bu etiket tutmazsa — yani veri kurcalandıysa — çözme işlemi başarısız sayılır ve bozuk veri asla kabul edilmez. Yani AEAD, hem "bunu yalnızca anahtar sahibi okuyabilir" hem de "bu, yazıldığı gibi, bozulmadan duruyor" güvencesini birlikte verir. Bu kiplerin kritik bir kuralı vardır: aynı anahtarla iki farklı şifrelemede asla aynı tek-kullanımlık değer (nonce) yinelenmemelidir; aksi halde güvence çöker. Üçüncü kısımda OxiDB'nin, depolama katmanında tam da böyle bir kimliği doğrulanmış şifreleme kullandığını göreceğiz.

Şifrelemenin kalbinde, dürüstçe konuşulması gereken bir gerçek yatar: şifreleme, yalnızca **anahtarı** kadar güvenlidir. Veriyi şifrelemek, onu koruyan anahtarı güvende tutmak sorununu doğurur; anahtar ele geçerse, şifreleme hiçbir işe yaramaz. Bu yüzden anahtar yönetimi, şifrelemenin ayrılmaz ve çoğu zaman en zor parçasıdır. Olgun sistemlerde anahtarlar, veritabanının yanında düz olarak durmaz; ayrı ve sıkı korunan bir **anahtar yönetim sistemine** (key management system, KMS) emanet edilir. Yaygın bir desen, iki katmanlı anahtarlamadır: asıl veriyi şifreleyen veri anahtarının kendisi, KMS'te tutulan bir ana anahtarla şifrelenip saklanır. Böylece veri anahtarını döndürmek (eskisini emekliye ayırıp yenisine geçmek) ya da bir sızıntı şüphesinde iptal etmek kolaylaşır; ana anahtar ise hiçbir zaman korunaklı sınırının dışına çıkmaz. Ayrıca bir sınırı da görmek gerekir: dururken şifreleme, çalınan diske karşı korur, ama çalışan sisteme zaten meşru biçimde girmiş bir saldırgana karşı korumaz — çünkü o saldırgan, sistemin verdiği çözülmüş veriyi görür. Daha ileri biçimler — veriyi veritabanının kendisinden bile gizleyen, yalnızca istemcide çözülen şifreleme — bu boşluğu kapatmaya çalışır. Üçüncü kısımda OxiDB'nin hem aktarım sırasında hem de dururken şifrelemeyi desteklediğini, ama bu daha ileri biçimlerin henüz kapsam dışı olduğunu göreceğiz.

14.5 Denetim: kim, ne zaman, ne yaptı

İlk üç katman, kötü şeylerin **olmasını engellemeye** çalışır. Dördüncü katman, **denetim** (audit), farklı bir amaca hizmet eder: olan biteni **kaydetmek**. Denetim, "kim, ne zaman, ne yaptı" sorusunu yanıtlayan bir kayıt tutar. Bir şeyi önlemez; ama hesap verebilirliği sağlar, bir ihlal olduğunda neyin nasıl olduğunu sonradan çözmeyi mümkün kılar ve birçok düzenlemenin zorunlu kıldığı izlenebilirliği karşılar. Binadaki güvenlik kameralarına benzer: bir hırsızlığı o an durdurmaz, ama sonradan kimin ne yaptığını gösterir.

Denetimin kendine özgü bir ödünleşimi vardır: ne kadar çok şey kaydederseniz, o kadar eksiksiz bir iz tutarsınız, ama o kadar çok yer harcar ve sistemi o kadar yavaşlatırsınız. Bu yüzden ne kaydedileceğine — her okuma mı, yalnızca yazmalar mı, yalnızca yetki ihlalleri mi — dikkatle karar vermek gerekir. Ayrıca denetim kayıtlarının kendisi de yönetilmelidir: sınırsız büyümemeleri için döndürülmeli (eski kayıtlar arşivlenip yenilerine yer açılmalı).

Denetimin en kritik ama en çok ihmal edilen yanı, kayıtların **bütünlüğüdür**. Bir denetim günlüğünün değeri, ona güvenilebilmesinden gelir; oysa sisteme sızmış bir saldırganın ilk işlerinden biri, çoğu zaman izini silmek için günlükteki kendi satırlarını silmek ya da değiştirmektir. Bu yüzden ciddi sistemler, günlüğü yalnızca-ekleme (append-only) tutmakla kalmaz, onu **kurcalama-belirten** (tamper-evident) hale getirmeye çalışır. Yaygın bir teknik, her kayda bir öncekinin özetini katmaktır — böylece kayıtlar bir **özet zinciriyle** birbirine bağlanır; ortadaki tek bir satırı bile değiştirmek, ondan sonraki tüm özetleri tutarsız kılar ve oynama anında belli olur. Daha da güçlü bir koruma, günlüğü gerçek zamanlı olarak ayrı, salt-yazılır bir hedefe akıtmaktır; böylece kayıt, saldırganın eriştiği makineden bağımsız bir yerde de durur. Bütünlük güvencesi olmadan denetim, en çok ihtiyaç duyulduğu anda — bir ihlalin ardından — sessizce işe yaramaz hale gelebilir. Üçüncü kısımda OxiDB'nin, isteğe bağlı olarak açılan ve boyut ya da zamana göre döndürülebilen bir denetim günlüğü tuttuğunu göreceğiz.

14.6 Güvenlik bir bütündür ve ödünleşimlerle doludur

Bu dört katmanı ayrı ayrı anlattık, ama asıl ders, onların bir **bütün** oluşturmasıdır ve her biri ötekiler olmadan eksik kalır. Yetkilendirme olmadan kimlik doğrulama anlamsızdır: kim olduğunuzu bilip de ne yapabi-

leceğinizi sınırlamıyorsanız, herkes her şeyi yapabilir. Anahtar yönetimi olmadan şifreleme bir tiyatrodur: anahtar herkesin görebileceği bir yerdeyse, şifrelemenin hiçbir değeri yoktur. Gözden geçirilmeyen denetim yalnızca gürültüdür: hiç kimse bakmıyorsa, kayıt tutmanın bir anlamı kalmaz. Güvenlik, ancak tüm katmanlar birlikte ve tutarlı biçimde çalıştığında gerçek olur.

Ve güvenlik de, kitabın geri kalanı gibi, ödünleşimlerle doludur: daha güçlü güvenlik, çoğu zaman daha az kolaylık ve daha düşük performans demektir. Yavaş parola özetleri, meydan-okuma protokolleri, şifreleme, denetim kaydı — hepsi bir maliyet taşır. Belki de en sık unutulan gerçek, güvenlik ihlallerinin çoğunun karmaşık saldırılardan değil, basit **yanlış yapılandırmalardan** kaynaklandığıdır: açık bırakılmış bir yetki, varsayılan bir parola, kapatılması unutulmuş bir kapı. Bu yüzden iyi bir veritabanının güvenli **varsayılanlarla** gelmesi — güvenliği elde etmek için ekstra çaba değil, onu zayıflatmak için bilinçli çaba gerektirmesi — en az mekanizmaların kendisi kadar önemlidir.

14.7 Kısım II'nin sonu ve Kısım III'e geçiş

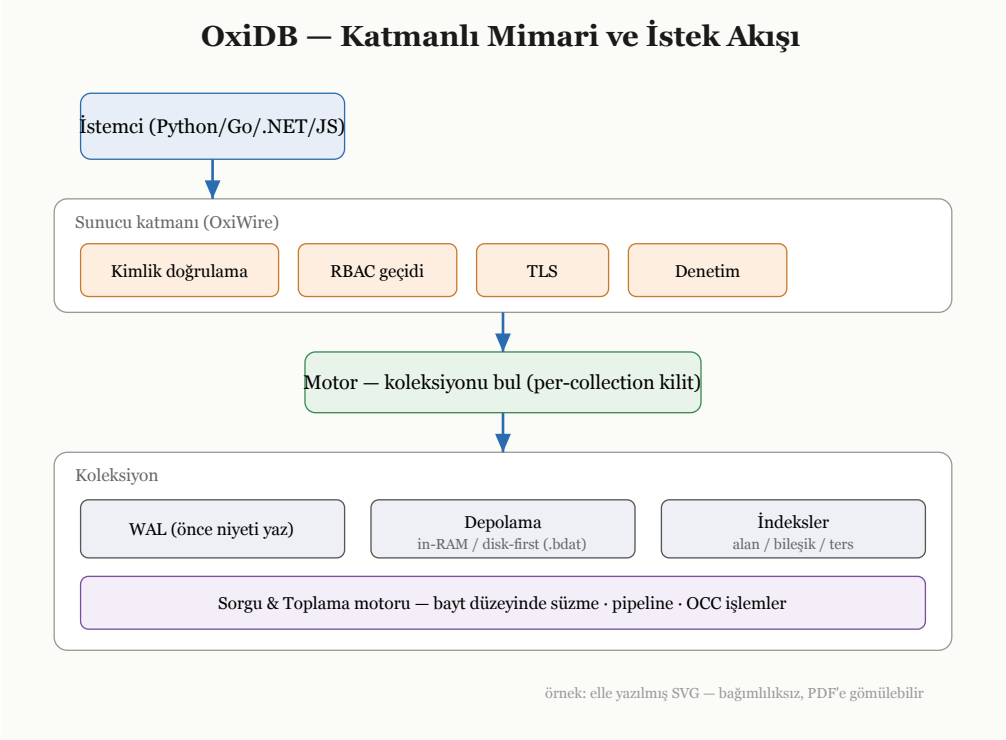
Bu bölümle birlikte Kısım II'yi tamamlamış olduk. Artık bir belge veritabanının içeride nasıl çalıştığına dair eksiksiz bir zihinsel haritamız var. Verinin diske nasıl yazıldığını ve çökmeden nasıl kurtarıldığını biliyoruz. Aradığımızı taramadan nasıl bulduğumuzu, sorularımızın nasıl yanıtlandığını ve veriyi nasıl özetlediğimizi biliyoruz. Eşzamanlı yazmalar karşısında tutarlılığın nasıl korunduğunu, sistemin tek makinenin ötesine nasıl ölçeklendiğini, belleğin nasıl yönetildiğini ve verinin nasıl korunduğunu biliyoruz. Bu harita, herhangi bir belge veritabanına — yalnızca OxiDB'ye değil — uygulanır; çünkü hepsi, bu aynı temel sorunları, bu aynı ilkelerle çözer.

Şimdiye dek hep ilkeler düzeyinde, "bir belge veritabanı bu sorunu *nasıl çözer*" sorusuyla kaldık. Üçüncü kısımda ise somuta iniyoruz: tüm bu ilkelerin, gerçek bir motorda — OxiDB'de — nasıl hayata geçtiğini adım adım göreceğiz. Her kavramı, onun OxiDB'deki karşılığına, yapılan mühendislik tercihlerine ve o tercihlerin ardındaki gerekçelere bağlayacağız. Kısım II size bu kitabın sözlüğünü ve dilbilgisini öğretti; Kısım III, o dille yazılmış gerçek bir metni birlikte okuyacağımız yerdir. Bir sonraki bölümde, OxiDB'ye genel bir bakışla ve mimarisinin kuş bakışı görünümüyle başlıyoruz.

Bölüm 15

OxiDB'ye Genel Bakış ve Mimari

Kitabın ilk iki kısmı boyunca, hep ilkeler düzeyinde kaldık: bir belge veritabanının çözmesi gereken sorunları ve bu sorunları çözmenin genel yollarını inceledik. Artık somuta iniyoruz. Üçüncü kısım, bu ilkelerin gerçek bir motorda — OxiDB'de — nasıl hayata geçtiğini adım adım gösteriyor. Buradan itibaren her kavramı, onun OxiDB'deki karşılığına, yapılan mühendislik tercihine ve o tercihin ardındaki gerekçeye bağlayacağız. Bu bölüm, ayrıntılara dalmadan önce bir kuş bakışı sunuyor: OxiDB nedir, hangi parçalardan oluşur ve bu parçalar birbirine nasıl bağlanır?



Şekil 15.1: OxiDB'nin katmanlı mimarisi ve bir isteğin akışı.

15.1 OxiDB nedir

OxiDB, Rust dilinde yazılmış, hızlı ve gömülebilir bir veritabanı motorudur. Kalbinde belge modeli yatar: verisini, ikinci kısımda tanıdığımız anlamda belgeler — iç içe geçebilen, alanlardan oluşan, kendini tanımlayan nesneler — halinde tutar ve bunları JSON tabanlı bir sorgu diliyle sorgular. Kitabın bu kısmının büyük bölümü, işte bu belge motorunu enine boyuna inceler.

Ne var ki OxiDB, tek bir modele hapsolmuş değildir. Belge motorunun yanında, aynı sunucunun içinde, **birbirinden bağımsız iki motor daha** yaşar: satırları, tabloları ve join'leri olan bir **ilişkisel SQL motoru** ve yüksek hacimli ölçüm akışları için sütunsal, sıkıştırmalı bir **zaman serisi motoru**. Bunlara, büyük ikili nesneler için S3 uyumlu bir nesne depolama katmanı ile sıcak ve geçici durum için bellek-içi bir anahtar-değer katmanı eşlik eder. Bu motorlar ortak bir depolamayı paylaşmaz; her biri kendi dosyalarına, kendi isim uzayına ve kendi sorgu diline sahiptir. Bir koleksiyon adı ile bir SQL tablosu adı asla çakışmaz.[1]

Bu, ikinci bölümde gördüğümüz "her model bir ödünleşimdir" ilkesinin reddi değil, ona verilen olgun bir yanıttır: OxiDB, tek bir modeli her veri şekline zorla giydirmek yerine, her veri şekline kendi doğasına uygun bir motorla karşılık verir — ve bunu, uygulamanın beş ayrı sistemi ayrı ayrı işletmesine gerek kalmadan, tek bir sunucu, tek bir bağlantı ve tek bir işletim disiplini içinde yapar. Hangi verinin hangi motora ait olduğu sorusu ise ciddi bir tasarım sorusudur; kitabın kapanış bölümü tümüyle buna ayrılmıştır.

OxiDB'yi diğer birçok veritabanından ayıran ilk şey, **tek bir kimliğe sıkışıp kalmamasıdır**. Birinci bölümde, bir veritabanının uygulamayla iki uçtan konuşabileceğini söylemiştik: gömülü bir kütüphane olarak ya da ağ üzerinden bir sunucu olarak. OxiDB, aynı çekirdek motoru üzerine kurulu olarak, her iki uçta da çalışabilir. Bir uygulamanın içine gömülü bir kütüphane olarak yerleşebilir; ayrı bir süreç olarak çalışıp ağ üzerinden birçok istemciye hizmet verebilir; hatta bir web tarayıcısının içinde, küçültülmüş bir biçimde çalışabilir. Bunların hepsinde, altta yatan mekanizmalar — depolama, indeks, işlem, kurtarma — özünde aynıdır. Bu yüzden ikinci kısımda öğrendiğimiz her şey, OxiDB hangi kılıkta çalışırsa çalışsın geçerlidir.

15.2 Katmanların kuş bakışı görünümü

OxiDB'nin mimarisini anlamanın en kolay yolu, onu ikinci kısımda kurduğumuz zihinsel haritayla eşleştirmektir; çünkü OxiDB'nin katmanları, tam da o haritanın parçalarına karşılık gelir. En tepeden en dibe doğru inelim.

En üstte **motor** durur: tüm koleksiyonlara sahip olan, gelen istekleri doğru koleksiyona yönlendiren merkezi bileşen. Motor, her koleksiyonu ayrı bir kilitle yönettiği için, farklı koleksiyonlara erişen istekler birbirini beklemeden, eş zamanlı ilerleyebilir. Koleksiyonlar, ilk yazma geldiğinde kendiliğinden oluşur; önceden tanımlanmaları gerekmez — dördüncü bölümdeki şema esnekliğinin bir yansımasıdır bu.

Bu eşzamanlılık modelinin somut yapısına bakmaya değer, çünkü kitap boyunca döneceğimiz "eş zamanlı okuma" yeteneğinin kaynağı budur. Motor, koleksiyon adından koleksiyona giden bir eşlemeyi, bir **okuma-yazma kilidi** (read-write lock) ardında tutar: koleksiyon **listesini** değiştiren ender bir işlem — örneğin yeni bir koleksiyonun ilk kez oluşması — bu kilidi yazma kipinde, kısa bir an için tutar; oysa var olan bir koleksiyona erişen sıradan isteklerin ezici çoğunluğu, kilidi yalnızca okuma kipinde tutup koleksiyonu işaret eden paylaşımlı bir tutamak (handle) alır ve hemen bırakır. Böylece dış kilit, neredeyse hiçbir zaman bir darboğaza dönüşmez. Asıl eşzamanlılık ise koleksiyonun **içinde** yaşar: belge baytları, kova düzeyinde (bucket-level) kilitlenen, kilitsiz okumaya yakın davranan eş zamanlı bir hash eşlemesinde tutulur. Bu yapı sayesinde aynı koleksiyonun farklı belgelerine dokunan birçok iş parçacığı, tek bir büyük kilidi sırayla beklemek yerine paralel ilerler.[2] Bu yapı, kitap boyunca andığımız "koleksiyon içinde eş zamanlı erişim" yeteneğinin mimari temelidir.

[1] Her motor varsayılan olarak kapalıdır ve yalnızca bir ortam değişkeniyle açılır; kapalı bir motor ne bellek ne de disk maliyeti doğurur. Motorların ayrıntıları 28. ve sonraki bölümlerdedir.

[2] scc (Scalable Concurrent Containers) — kova düzeyinde kilitleme ve atomik göstericilerle eş zamanlı okuma/yazmayı tek bir dış kilit darboğazı olmadan destekleyen Rust veri yapıları kütüphanesi.

Her **koleksiyon**, kendi içinde, ikinci kısımda tanıdığımız bileşenleri barındırır: belge baytlarını tutan bir depolama katmanı; dayanıklılığı sağlayan bir yazma-öncesi günlük; sık erişilen veriyi bellekte tutan önbellekler; aramayı hızlandıran alan ve bileşik indeksler; ve eşzamanlılık denetimi için belge başına sürüm sayaçları. Yani her koleksiyon, küçük ölçekte, ikinci kısmın tüm hikâyesini içinde taşır.

Bu koleksiyonun içine indiğimizde, beşinci ve altıncı bölümlerin somut karşılıklarını buluruz. **Depolama katmanı**, ikinci kısımda anlattığımız iki felsefeyi birden sunar: bugün varsayılan olan diske öncelikli kip ile tüm veriyi bellekte tutan opsiyonel belleğe öncelikli kip. **Yazma-öncesi günlük**, altıncı bölümdeki dayanıklılık ve kurtarma mekanizmalarını — sağlama toplamları, denetim noktaları, çökmeden geri dönme — hayata geçirir.

Bir üst katmana çıktığımızda, yedinci, sekizinci ve dokuzuncu bölümlerin karşılıklarını görürüz. **İndeksler**, yalnızca temel alan indeksleriyle sınırlı değildir; OxiDB bileşik indeksleri, süreyle dolan (TTL) indeksleri, benzersizlik indekslerini, tam metin için ters indeksleri ve benzerlik araması için vektör indekslerini de destekler. **Sorgu ve toplama motoru**, sekizinci ve dokuzuncu bölümlerde anlattığımız operatörleri ve pipeline aşamalarını — gruplamayı, çok-yönlü analizi, pencere fonksiyonlarını — yürütür. **İşlem yöneticisi**, onuncu bölümdeki iyimser eşzamanlılık denetimini uygular.

En dışta, koleksiyonların ve motorun çevresinde, on ikinci ve on dördüncü bölümlerin karşılıkları yer alır. **Sunucu katmanı**, ağ üzerinden gelen istekleri karşılar; kendi ikili iletişim protokolünü, kimlik doğrulamayı, rol tabanlı erişim denetimini, aktarım şifrelemesini ve denetim günlüğünü barındırır. **Kümeleme katmanı**, on ikinci bölümdeki replikasyonu ve konsensüsü; ayrı bir yönlendirici bileşen ise sharding'i ve parçalar-arası toplamayı sağlar.

15.3 Çekirdeğin ötesi: ek yüzeyler

OxiDB, klasik bir belge veritabanının ötesine geçen birkaç ek yüzey de sunar ve bunları kuş bakışı haritaya eklemek, mimarinin genişliğini görmek açısından yararlıdır. Bu yüzeylerin ortak özelliği, hepsinin aynı çekirdek motorun üzerine oturmasıdır.

OxiDB, doğrudan bir HTTP arayüzü ve gerçek zamanlı bir abonelik kanalı sunarak, bir uygulamanın veritabanına web teknolojileriyle, ek bir katman kurmadan bağlanmasına olanak tanır; buna eşlik eden bir kimlik ve belge düzeyinde kural sistemiyle, küçük uygulamaların sunucu yazmadan doğrudan veritabanıyla çalışmasını mümkün kılar. Ayrı bir bellek-içi anahtar-değer katmanı, yaygın bir önbellek protokolüyle uyumlu çalışır; bir mesajlaşma protokolü desteği, veritabanını nesnelerin interneti gibi senaryolara taşır. Büyük ikili nesneler için bir nesne deposu, zamanın belirli bir noktasına geri dönmeyi sağlayan bir kurtarma yeteneği ve dururken şifreleme, bu ek yüzeyleri tamamlar. Bu kitabın odağı çekirdek belge motorudur; bu ek yüzeylere üçüncü kısmın ilerleyen bölümlerinde, çekirdeği anladıktan sonra değineceğiz.

15.4 Çalışma alanının yapısı ve istemciler

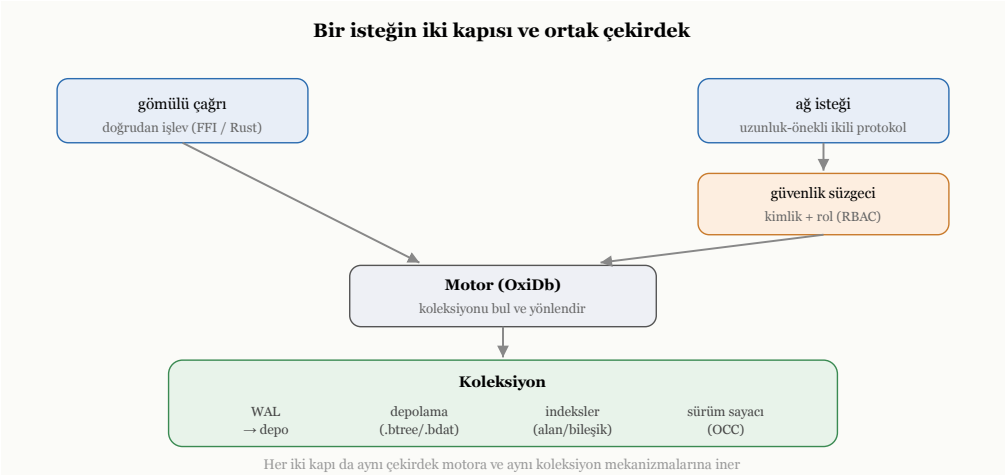
OxiDB'nin kod tabanı, birkaç ayrı parçaya bölünmüştür ve bu bölünme, mimarisinin mantığını yansıtır. Çekirdekte, belge motorunun kendisi yer alır — depolama, indeks, sorgu, işlem; yani ikinci kısmın tüm mekanizmaları. Bunun üzerine, çekirdeği ağ üzerinden sunan sunucu bileşeni; ve diğer programlama dillerinden çağrılabilmesini sağlayan bir köprü katmanı eklenir. Bu köprü sayesinde, OxiDB yalnızca Rust'tan değil, başka birçok dilden de kullanılabilir.

Bu köprünün üzerine kurulu **istemci kütüphaneleri**, OxiDB'yi çeşitli programlama dillerinden — örneğin Python, Go, .NET ve JavaScript'ten — kullanılabilir kılar. Bu istemciler, ya gömülü kipte çekirdeğe doğrudan bağlanır, ya da sunucu kipinde ağ üzerinden konuşur. İkinci kısımda anlattığımız tüm kavramlar — koleksiyonlar, sorgular, indeksler, işlemler — bu istemcilerin hepsinde aynı biçimde karşımıza çıkar; çünkü hepsi, aynı çekirdek motorun farklı kapılarıdır.

Bu bölünmeyi somut adlarıyla anmak yararlıdır, çünkü kod tabanını gezerken bu adlarla karşılaşırsınız. Çekirdek motor, kendi içinde bağımsız bir kütüphane (`oxidb` paketi) olarak yaşar; ikinci kısımda tek tek anlattığımız tüm mekanizmalar — depolama, yazma-öncesi günlük, indeksler, sorgu ve toplama, işlem yöneticisi, sıkıştırma, şifreleme — bu çekirdeğin parçalarıdır ve hiçbiri ağ ya da kimlik doğrulama hakkında bir şey bilmez. Bunun üzerine oturan **sunucu bileşeni** (`oxidb-server`), ağ protokolünü, kimlik doğrulamayı, rol tabanlı erişim denetimini, aktarım şifrelemesini ve denetim günlüğünü ekler; çekirdeği bir kütüphane olarak çağırır. Üçüncü parça, **C uyumlu köprü katmanıdır** (`oxidb-client-ffi`): çekirdeği, hemen her dilin konuşabildiği sade bir C arayüzü ardına koyar; Python, Go, .NET ve diğer istemciler bu köprü üzerinden gömülü kipte çekirdeğe doğrudan bağlanabilir. Bu üçlü ayrım rastgele değildir; "çekirdek hiçbir taşıma katmanını tanımaz" ilkesi sayesinde aynı motor, gömülü bir kütüphane, ağ sunucusu ya da tarayıcıdaki bir wasm modülü olarak, kodun tekrarlanmasına gerek kalmadan çalışabilir.

15.5 Bir isteğin yaşam döngüsü

Tüm bu parçaların nasıl birlikte çalıştığını görmek için, bir isteğin sistemden nasıl geçtiğini kabaca izleyelim; bu izlek, üçüncü kısmın geri kalanının yol haritası gibidir.



Şekil 15.2: Bir isteğin iki kapısı ve ortak çekirdek.

Bir istek iki yoldan gelebilir. Gömülü kipte, doğrudan bir işlev çağrısı olarak; sunucu kipinde ise ağ üzerinden, OxiDB'nin kendi ikili protokolüyle çerçevelenmiş bir mesaj olarak. Sunucu kipinde istek, önce bir güvenlik süzgecinden geçer: kimlik doğrulanır ve rolün bu işleme izni olup olmadığı denetlenir. Bu süzgeçten geçen istek, motora ulaşır; motor, isteğin hedeflediği koleksiyonu bulur. Eğer istek bir okumaysa, sorgu motoru devreye girer: koşulları ayrıştırır, hangi indeksin işe yarayacağına karar verir, adayları daraltır ve süzer. Eğer istek bir yazmaysa, önce yazma-öncesi günlüğe kaydedilir ve dayanıklı kılınır, sonra depolama katmanına ve indekslere uygulanır. Sonuç, geldiği yoldan geri döner. Kümeleme kipinde, bir yazma ek olarak konsensüs katmanından, sharding'de ise yönlendiriciden geçer.

İki kapı arasındaki fark, yalnızca isteğin nasıl çerçevelendiğindedir; çekirdeğe indikten sonra izlenen yol aynıdır. Gömülü kipte istek, bir işlev çağrısı olarak gelir ve hiçbir ağ ya da güvenlik katmanına uğramadan doğrudan motora ulaşır; güvenlik denetimi, gömülü kipi kullanan uygulamanın kendi sorumluluğundadır. Sunucu kipinde ise istek, ağ üzerinden, OxiDB'nin uzunluk-önekli ikili protokolüyle çerçevelenmiş bir mesaj olarak gelir: her mesajın başında, gövdenin uzunluğunu bildiren bir önek vardır; böylece sunucu, bir mesajın nerede bitip diğerinin nerede başladığını, akışı çözmeye çalışmadan bilir. Bu mesaj önce güvenlik süzgecinden geçer — kimlik doğrulanır, rolün bu işleme izni denetlenir — ardından çekirdeğe, gömülü kiptekiyle aynı kapıdan girer. Bu yüzden sunucu katmanı, çekirdeğin üzerinde ince bir kabuktur: ona bir ağ yüzü ve bir güvenlik kapısı ekler, ama veritabanı işinin kendisini değiştirmez.

Bu kısa izlek, aslında bu kitabın bir özetidir: bir istek, ikinci kısımda tek tek anlattığımız tüm katmanlardan sırayla geçer. Üçüncü kısmın geri kalanı, bu izleğin her durağını yakın plana alır.

15.6 Üçüncü kısmın yol haritası

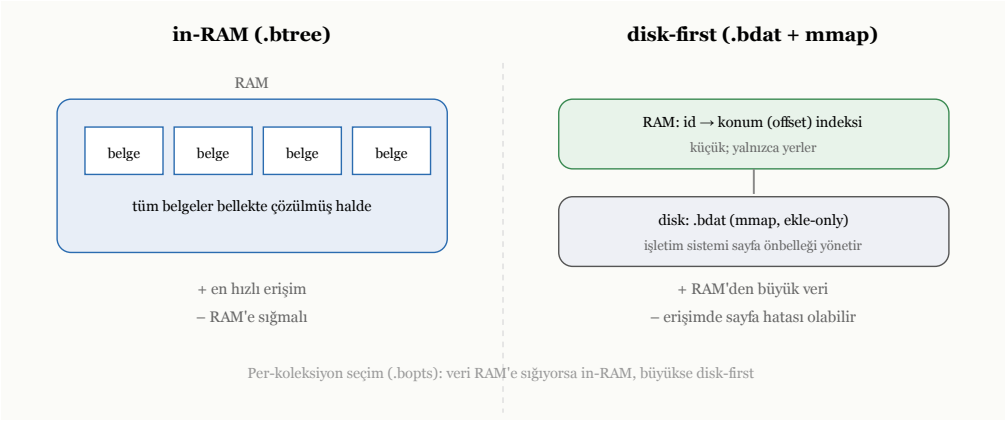
Üçüncü kısım, isteğin yaşam döngüsünü izleyerek, en dipten en üste doğru ilerleyecek. Önce **depolama katmanına** ineceğiz: OxiDB'nin belleğe öncelikli ve diske öncelikli kiplerini, verinin diskte nasıl yerleştiğini ve belleğe yansıtmanın bellek ayak izini nasıl belirlediğini göreceğiz. Ardından **dayanıklılık** ve kurtarma, sonra **indeksler**, **sorgu motoru** ve **toplama pipeline'ı** gelecek — her biri, ikinci kısımda kurduğumuz ilkeyi OxiDB'nin somut tercihine bağlayarak. **İşlemler**, **sıkıştırma**, ve ardından çekirdeğin ötesindeki ek yüzeyler — tam metin arama, nesne deposu, şifreleme, kurtarma — ele alınacak. Sonra **sunucu** katmanına ve protokolüne, oradan **ölçeklendirmeye** — Raft kümesi ve sharding'e — ve **istemcilere** geçeceğiz. Kısım, OxiDB'nin bellek optimizasyonunu ve gerçek bir karşılaştırmalı değerlendirmesini ele alarak kapanacak.

Bu yolculuğun her durağında, tanıdık bir kavramı yeniden göreceksiniz — ama bu kez soyut değil, somut. İkinci kısım size dili öğretti; üçüncü kısım, o dille yazılmış gerçek bir metni birlikte okuyacağımız yerdir. Bir sonraki bölümde, en dipten, depolama katmanından başlıyoruz: OxiDB belgeleri tam olarak nasıl saklar ve neden iki ayrı depolama kipi sunar?

Bölüm 16

OxiDB'nin Depolama Katmanı: In-RAM ve Disk-First, mmap, .bdat / .btree

Önceki bölümde OxiDB'nin mimarisine kuş bakışı baktık ve isteğin yaşam döngüsünü izleyerek en dipten, depolama katmanından başlayacağımızı söylemiştik. İşte buradayız. Beşinci ve on üçüncü bölümlerde, bir depolama motorunun belge baytlarını diske nasıl yerleştirdiğini ve bellek ile disk arasındaki ödünleşimi genel ilkeler düzeyinde tanımıştık. Bu bölüm, o ilkelerin OxiDB'de nasıl somutlaştığını gösteriyor: OxiDB belgeleri tam olarak nasıl saklar, neden iki ayrı depolama kipi sunar ve bu seçimin bellek ile hız üzerindeki sonuçları nelerdir?



Şekil 16.1: in-RAM ile disk-first depolama kiplerinin karşılaştırması.

16.1 Çekirdek: kimlikten baytlara

Beşinci bölümde, bir depolama motorunun belgenin içeriğiyle ilgilenmediğini, onun için bir belgenin "bir kimliğe bağlı bir bayt yığını" olduğunu söylemiştik. OxiDB'nin depolama çekirdeği tam olarak böyle çalışır: her belgeye bir kimlik verir ve o kimlikten belgenin baytlarına giden bir eşleme tutar. Üstüne kurulu tüm katmanlar — sorgu, indeks, işlem — bu çekirdeğe iki temel istekle gelir: "şu kimlikli belgeyi sakla" ve "şu kimlikli belgeyi geri ver."

Bu eşlemenin altında, eş zamanlı erişime izin veren bir yapı yatar. Birçok istek aynı anda farklı belgelere dokunabildiği için, depolama çekirdeği, parça parça kilitlenebilen — yani bir bölümüne dokunulurken başka bir bölümüne paralel erişilebilen — bir yapı kullanır. Bu, on beşinci bölümde değindiğimiz "koleksiyon içinde eş zamanlı erişim" yeteneğinin temelidir.

Belgenin kendisi, bu çekirdekte ham JSON metni olarak değil, dördüncü bölümde öngördüğümüz gibi **daha zengin, daha sıkı bir ikili biçimde** tutulur. Dışarıya, kullanıcıya JSON olarak görünen belge, içeride bu ikili biçime kodlanır; bu biçim hem yer açısından daha tutumludur hem de bir alana, tüm belgeyi baştan ayrıştırmadan doğrudan erişmeye olanak tanır. Bu inceliğin sorgu hızına katkısını, ileride sorgu motorunu ele alırken yeniden göreceğiz.

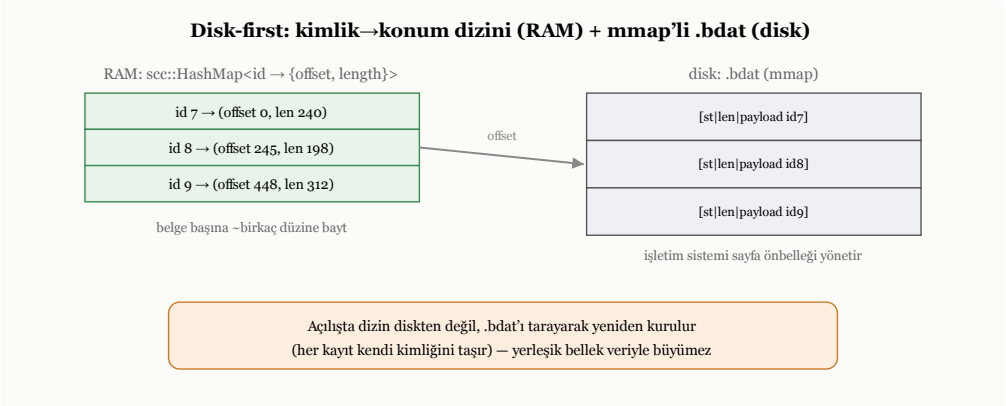
Bu çekirdeğe eşlik eden ikinci bir bellek yapısı daha vardır: bir **belge önbelleği** (document cache). Bir belgenin baytları okunduğunda, OxiDB onları her seferinde yeniden çözmek yerine, bir kez çözüp sonucu paylaşımlı, sayılı bir göstericinin (refcounted pointer) ardında tutar; aynı belgeye sonradan dokunan istekler, çözülmüş bu temsili kopyalamadan paylaşır. Böylece sık erişilen bir belge için ayrıştırma maliyeti yalnızca bir kez ödenir. Bu önbellek, iki depolama kipinin de üzerinde durur; ama özellikle disk-öncelikli kipte değerlidir, çünkü orada bir okuma diske gidebilir ve önbellek bu maliyeti tekrar tekrar ödemeyi önler.

16.2 İki kip, iki felsefe

OxiDB'nin depolama katmanının en belirleyici özelliği, on üçüncü bölümde tanıdığımız iki felsefeyi — belleğe öncelikli ve diske öncelikli — birden sunmasıdır. Bu iki kip, aynı çekirdek arayüzün arkasında, verinin nerede yaşadığına dair taban tabana farklı iki karar verir.

16.2.1 Disk-öncelikli kip: varsayılan

OxiDB'nin varsayılan kipi, on üçüncü bölümün diske öncelikli felsefesini hayata geçirir. Bu kipte, bellekte tüm belgeler değil, yalnızca küçük bir **kimlik-konum dizini** durur: her belgenin kimliğinden, o belgenin disk üzerindeki yerine — yani başlangıç konumu (offset) ve uzunluğa — giden, belge başına yalnızca birkaç düzine bayt tutan kompakt bir eşleme. Belgelerin asıl baytları ise, beşinci bölümdeki append-only felsefeyle yazılan bir veri dosyasında — `.bdat` dosyasında — durur ve gerektiğinde oradan okunur.



Şekil 16.2: Disk-first: kimlik-konum dizini ve mmap'li veri dosyası.

Bu kararın bellek üzerindeki etkisi çarpıcıdır. Yerleşik bellek artık veriyle birlikte büyümez; yalnızca o küçük dizin kadar yer kaplar. Milyon belgelik bir koleksiyonun bellekteki yükü, yüzlerce megabayttan birkaç düzine megabayta iner. Önemlisi, OxiDB bu kipte yeni bir depolama motoru icat etmez; beşinci ve altıncı bölümlerde anlattığımız, append-only, mmap ile okunan, soft-delete ve sağlama desteği olan sağlamlaşmış depolama

bileşenini yeniden kullanır. Yani disk-öncelik, köklü bir yeniden yazım değil, var olan dayanıklı altyapının zarif bir biçimde yeniden kullanılmasıdır.

16.2.2 Belleğe öncelikli kip: opsiyonel

İsteğe bağlı ikinci kipte, her belgenin baytları **bellekte yerleşik** olarak durur. Yani tüm koleksiyon, kelimenin tam anlamıyla, bellekteki o eş zamanlı eşlemenin içinde yaşar. Diske yazılan dosya — uzantısıyla anılırsa, `.btree` dosyası — yalnızca bu bellek içeriğinin periyodik bir **anlık görüntüsüdür**; veritabanı açıldığında bu dosya bütünüyle belleğe geri yüklenir. Bu yüzden OxiDB, bu kipe geçildiğinde, aslında disk üzerinde kalıcılığa sahip bir **bellek-içi veritabanı** gibi davranır.

Bunun sonucu, on üçüncü bölümün belleğe öncelikli felsefesinin tam karşılığıdır. Okumalar olağanüstü hızlıdır, çünkü her belge zaten bellektedir; hiçbir okuma diske gitmez. Bedeli ise belleğin **veriyle birlikte büyümesidir**: her belgenin baytları bellekte yer kapladığı için, milyon belgelik bir koleksiyon, yüzlerce megabayt yerleşik bellek tüketir. Bu, veri belleğe rahatça sığdığı sürece mükemmel bir tercihtir; ama veri büyüdükçe, bellek hem pahalı hem de sınırlayıcı bir kısıt haline gelir.

16.3 Dosyaların dili: .btree, .bdat ve .bopts

OxiDB'nin hangi koleksiyonun hangi kipte olduğunu bilmesi gerekir; bunu, dosya adlarındaki bilinçli bir ayrımla yapar. Belleğe öncelikli bir koleksiyonun anlık görüntüsü `.btree` uzantısıyla; disk-öncelikli bir koleksiyonun append-only veri dosyası ise `.bdat` uzantısıyla yazılır. Bu iki uzantı kasıtlı olarak farklıdır: veritabanı açıldığında, bir koleksiyonun yanında hangi dosyayı gördüğüne bakarak onun hangi kipte olduğunu anlar.

Buna eşlik eden üçüncü bir dosya, `.bopts`, on ikinci bölümde tanıdığımız per-koleksiyon seçeneklerini kalıcı kılar. OxiDB, bir koleksiyon için tek tek şu tercihleri saklar: koleksiyonun disk-öncelikli mi yoksa belleğe öncelikli mi olduğu; kayıtların sıkıştırılarak mı yoksa ham mı yazılacağı; otomatik sıkıştırmanın (yani ölü alanı geri kazanan bakımın) açık olup olmadığı; ve bu otomatik bakımın ne zaman tetikleneceğini belirleyen iki eşik — dosyanın en az ne kadar büyümesi gerektiği ve ölü alanın oranının hangi yüzdeyi aşması gerektiği. Bu beş tercih, bir koleksiyon ilk oluşturulurken seçilir ve `.bopts` dosyasına yazılır.

Bu kalıcılığın incelikli ama önemli bir sonucu vardır. OxiDB'nin ilk sürümlerinde, disk-öncelik gibi tercihler yalnızca ortam değişkenleriyle, yani süreç düzeyinde belirleniyordu; bu da bir veritabanını farklı bir ortamda yeniden açtığınızda koleksiyonların beklenmedik bir kipe düşmesi anlamına gelebilirdi. `.bopts` dosyası bu kırılganlığı giderir: bir koleksiyon hangi tercihle oluşturulduysa, yeniden açıldığında o tercihle açılır — sistemin o anki ortam ayarlarından bağımsız olarak. Ortam değişkenleri artık yalnızca, açıkça bir tercih belirtilmemiş yeni koleksiyonlar için bir **varsayılan** rolü oynar. Böylece aynı veritabanında, bir koleksiyon belleğe öncelikli ve sıkıştırmalı, başka biri disk-öncelikli ve sıkıştırmasız olabilir; her biri kendi kimliğini, taşındığı her ortamda korur. Bu, on ikinci bölümdeki "per-koleksiyon ayar" fikrinin somut bir uygulamasıdır.

16.4 Belleğe yansıtmanın somut etkisi

Disk-öncelikli kipte, `.bdat` veri dosyası belleğe yansıtılır — on üçüncü bölümde değindiğimiz mmap tekniği.[1] Bir belge okumak, kimlik-konum dizininden konumu bulmak ve ardından belleğe yansıtılmış dosyanın o konumundan baytları okumaktır. Veri zaten bellekteyse bu çok hızlıdır; değilse, işletim sistemi onu diskten getirir ve bellek baskı altındayken yine sessizce geri atabilir. Bu "sessizce geri atma" yeteneği, disk-öncelikli kipin yerleşik bellek vaadinin özüdür: işletim sistemi, başka bir şey için yer açması gerektiğinde, mmap'li sayfaları diskte zaten bir kopyası olduğu için doğrudan serbest bırakabilir; çünkü bu sayfalar, değiştirilmemiş, diskle birebir aynı veridir.

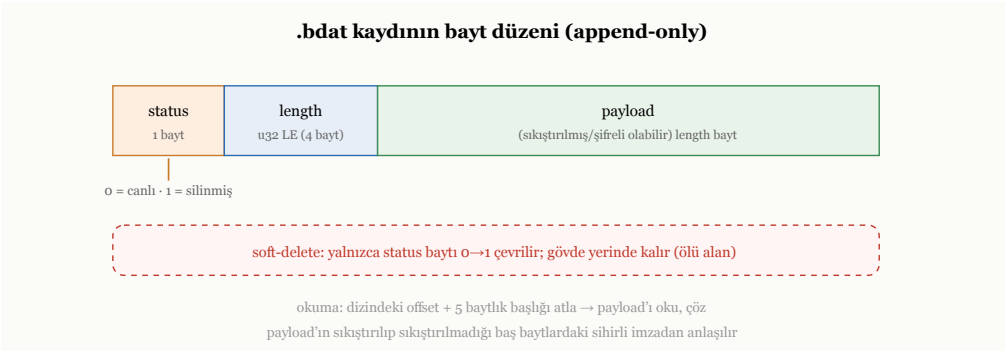
[1] Bellek-eşleme (memory-mapped files) — bir dosyayı sürecin adres uzayına yansıtarak, okuma/yazmayı sayfa hatalarıyla (page fault) işletim sisteminin sayfa önbelleğine devreden bir mekanizma. POSIX `mmap` çağrısı bunu standartlaştırır.

Bunun somut sonucu, on üçüncü bölümde tartıştığımız bellek ölçümü inceliğinin canlı bir örneğidir. Disk-öncelikli kipte, beş yüz bin belgelik bir koleksiyonu taze açtığınızda, sistemin yerleşik belleği yalnızca birkaç megabayt mertebesindedir — çünkü henüz yalnızca o küçük dizin bellektedir, belgelerin gövdesi diskte beklemektedir. Buna karşılık aynı veriyi belleğe öncelikli kipte açtığınızda, tüm belgeler belleğe yüklendiği için yerleşik bellek kat kat yüksektir. İki kip arasındaki bu taze-açılış farkı, disk-öncelik tercihinin asıl kazancıdır.

Ama on üçüncü bölümün dürüst uyarısını burada da tekrarlamak gerekir: tüm veriye dokunan büyük bir taramadan sonra, disk-öncelikli kipin belleği de yükselir — çünkü işletim sistemi dokunulan sayfaları belleğe çeker. Bu yükselen bellek geri alınabilir bir bellektir; baskı altında işletim sistemi onu serbest bırakır. Bu yüzden disk-öncelik, çalışma kümesi tüm veriden küçük olan yükler için en çok kazandırır; tüm veriyi sürekli baştan sona tarayan yükler için kazancı azalır.

16.5 Bir kaydın anatomisi: status, uzunluk, yük

Disk-öncelikli `.bdat` dosyasındaki bir kaydın bayt düzeyindeki yapısını yakından görmek, hem append-only doğanın hem de soft-delete'in neden bu kadar ucuz olduğunu açıklar. Her kayıt üç parçadan oluşur: tek baytlık bir **durum baytı** (status), ardından dört baytlık bir **uzunluk alanı** ve en sonda **yük** (payload) — yani belgenin asıl, kodlanmış baytları. Uzunluk alanı, yükün kaç bayt olduğunu küçük-uçtan-büyüğe (little-endian) düzende söyler; böylece okuyucu, bir kaydın nerede bitip bir sonrakinin nerede başladığını, içeriği çözmeden bilir.



Şekil 16.3: .bdat kaydının bayt düzeni.

Bu basit düzen, beşinci bölümde tanıdığımız iki mekanizmayı doğrudan mümkün kılar. Bir belgeyi okumak için sistem, kimlik-konum dizininden başlangıç konumunu alır, beş baytlık başlığı (bir durum baytı artı dört baytlık uzunluk) atlar ve uzunluğun söylediği kadar yükü okur. **Soft-delete** ise yalnızca durum baytını "canlı"dan "silinmiş"e çevirmektir: gövdeye hiç dokunulmaz, dosya yeniden yazılmaz — tek bir bayt yerinde değişir. Bu yüzden silme neredeyse bedavadır; bedeli, silinen kaydın yerinin bir süre ölü alan olarak kalmasıdır. Yükün sıkıştırılmış mı yoksa ham mı olduğu, ayrı bir bayrağa gerek kalmadan, yükün ilk baytlarındaki sıkıştırma biçiminin sihirli imzasından (magic bytes) anlaşılır; böylece sıkıştırılmış ve sıkıştırılmamış kayıtlar aynı dosyada karışık durabilir ve doğru okunabilir.

16.6 Append-only olmanın sonuçları

Disk-öncelikli kip, beşinci bölümdeki append-only felsefeyi izlediği için, o felsefenin iki sonucunu da devralır. Birincisi, var olan veriyi yerinde değiştirmemesidir: bir belge güncellendiğinde, eski baytlar `.bdat` dosyasında olduğu gibi kalır, yeni baytlar dosyanın sonuna eklenir ve kimlik-konum dizini yeni konuma güncellenir; eski konum ölü alana dönüşür. İkincisi, bu ölü alanın zamanla biriktiğidir ve onu geri kazanmak için sıkıştırmaya — yirmi ikinci bölümün konusuna — ihtiyaç vardır.

Bu append-only doğanın bir başka sonucu, yazma hızındadır. Beşinci bölümde gördüğümüz gibi, dosyanın sonuna ardışık ekleme, diskin en sevdiği yazma biçimidir; bu yüzden disk-öncelikli kipte yazmalar, toplu olarak ve ardışık biçimde yapıldığında oldukça verimlidir.

16.7 Sıkıştırma: yer, işlemci ve sıfır-kopya

On üçüncü bölümde sıkıştırmanın yer-işlemci-sıfırkopya üçgenini tanıtmıştık; OxiDB'nin disk-öncelikli kipi, bu üçgenin tam bir uygulamasını sunar. `.bdat` dosyasındaki kayıtlar, varsayılan olarak sıkıştırılmadan, ham baytlarıyla saklanır; isteğe bağlı bir tercihle ise, hızlı ve dengeli bir sıkıştırma algoritmasıyla[2] sıkıştırılarak da saklanabilir. Sıkıştırılmamış saklama daha çok yer kaplar, ama on üçüncü bölümde tanıttığımız sıfır-kopya erişimi mümkün kılar: baytlar zaten ham haldeyse, belleğe yansıtılmış dosyadan doğrudan, hiç kopyalamadan ve açmadan kullanılabilir. Sıkıştırılmış saklama ise diskte daha az yer kaplar, ama her okumada baytların açılmasını gerektirir.

Hangisinin kazandığı, on üçüncü bölümdeki ilkeye uyar: veriye bağlıdır. Gerçekten iyi sıkışan, seyrek taranan veri için sıkıştırma kazandırır; az sıkışan ama sık ve büyük taranan veri için sıkıştırmamak daha iyidir. OxiDB üzerinde yapılan ölçümler bu dengeyi somut biçimde gösterir: küçük, yapılandırılmış belgelerden oluşan tipik bir veri kümesinde, sıkıştırma diskte kayda değer bir yer kazandırmazken, her taramada belgeleri açma zorunluluğu — kayıt kayıt açma maliyeti taramaları ve toplamaları kat kat yavaşlattığı için — büyük bir bedel getiriyordu. OxiDB, tam da bu yüzden, sıkıştırmayı varsayılan olarak kapalı tutar ve ham saklamayı yeğler; sıkıştırma, ancak veri gerçekten iyi sıkışıp seyrek tarandığında açılmaya değer bir tercihe dönüşür. Bu, on üçüncü bölümün soyut ödünleşiminin gerçek bir sistemdeki yankısıdır.

16.8 Açılış ve kurtarmaya kısa bir bakış

Bir koleksiyon açıldığında, depolama katmanının kendini kullanılabilir bir duruma getirmesi gerekir ve bu, kipe göre değişir. Belleğe öncelikli kipte, sistem `.btree` anlık görüntüsünü bütünüyle belleğe geri yükler. Disk-öncelikli kipte ise, `.bdat` dosyasındaki yaşayan kayıtları tarayarak kimlik-konum dizinini yeniden kurar — her belge kendi kimliğini taşıdığı için bu mümkündür. Her iki kipte de, bu adımın ardından yazma-öncesi günlük devreye girip son değişiklikleri uzlaştırır; ama bu, bir sonraki bölümün konusudur.

16.9 Bu bölümün bıraktığı yer

Bu bölümde, OxiDB'nin depolama katmanını yakın plana aldık. Çekirdeğin, belgeleri bir kimlikten baytlara eşleyen, eş zamanlı erişime açık bir yapı olduğunu; belgelerin içeride zengin bir ikili biçimde tutulduğunu gördük. İki kipi tanıdık: yalnızca kompakt bir dizini bellekte tutup veriyi mmap ile diskte bırakan, bellek-tutumlu disk-öncelikli varsayılanı; ve tüm veriyi bellekte tutan, hızlı ama bellek-yoğun belleğe öncelikli opsiyonel kipi. Dosya uzantılarının (`.btree`, `.bdat`, `.bopts`) bu kipleri ve per-koleksiyon tercihleri nasıl kodladığını; belleğe yansıtmanın bellek ayak izini nasıl belirlediğini; append-only olmanın sonuçlarını; ve sıkıştırmanın somut ödünleşimini gördük.

Buraya kadar hep verinin nasıl **saklandığından** söz ettik. Ama altıncı bölümde öğrendiğimiz gibi, bir veriyi diske yazmak, onun çökmeye karşı güvende olduğu anlamına gelmez. OxiDB, bir yazmanın gerçekten dayanıklı olduğundan nasıl emin olur; çökmeden nasıl kurtulur; ve dayanıklılık ile hız arasındaki o gerilimi — her commit'i diske zorlamak ile arada bir zorlamak arasındaki tercihi — nasıl yönetir? Bir sonraki bölümde, OxiDB'nin yazma-öncesi günlüğüne ve dayanıklılık mekanizmalarına eğiliyoruz.

[2] zstd (Zstandard) — Facebook tarafından geliştirilen, yüksek sıkıştırma oranını düşük işlemci maliyetiyle birleştiren, kayıpsız bir sıkıştırma algoritması; RFC 8878 olarak standartlaştırılmıştır.

Bölüm 17

OxiDB'de WAL, Dayanıklılık ve Kurtarma; Katı ve Gevşek Senkronizasyon

Önceki bölümde, OxiDB'nin belgeleri nasıl sakladığını gördük. Ama altıncı bölümde öğrendiğimiz gibi, bir veriyi diske yazmak, onun çökmeye karşı güvende olduğu anlamına gelmez. Bu bölüm, OxiDB'nin bir yazmanın gerçekten dayanıklı olduğundan nasıl emin olduğunu, çökmeden nasıl kurtulduğunu ve altıncı bölümde tanıdığımız o dayanıklılık tayfında — her commit'i diske zorlamak ile arada bir zorlamak arasında — nereye oturduğunu ele alıyor. Bu, OxiDB'nin verdiği en temel sözlerden birinin — "tamamlandı dediysem, kaybolmaz" — ardındaki düzenektir.

Şekil 17.1: Katı ve gevşek senkronizasyon; grup commit.

17.1 OxiDB'nin yazma-öncesi günlüğü

Altıncı bölümdeki ilkeyi hatırlayalım: asıl veriyi değiştirmeden önce, ne yapacağını ayrı bir günlüğe yaz ve o günlüğü dayanıklı kıl; dayanıklılık günlükten gelir, asıl güncelleme tembelce yapılır. OxiDB, bu ilkeyi doğrudan uygular. Her koleksiyonun, yalnızca sona eklenerek büyüyen bir yazma-öncesi günlüğü vardır.

Bu günlüğe yazılan her kayıt, altıncı bölümde anlattığımız iki korumayı taşır. Birincisi, kaydın içeriğinden hesaplanan bir **sağlama toplamıdır** (CRC32); bu, kurtarma sırasında yarım kalmış ya da bozulmuş bir

kaydı yakalamayı sağlar. İkincisi, kaydın hangi işleme ait olduğunu belirten bir **işlem kimliğidir**; bu, ileride işlemleri ele alırken önemli olacak.

Bu iki korumayı kaydın bayt düzeyindeki biçiminde somut görmek öğreticidir, çünkü kurtarmanın neden güvenli olduğunu bu biçim açıklar. Her kayıt iki katmandan oluşur: bir dış **çerçeve** ve onun içindeki **yük**. Dış çerçeve, önce dört baytlık sağlama toplamını, sonra dört baytlık bir uzunluk alanını taşır; böylece okuyucu, yükün kaç bayt olduğunu ve onu okuduktan sonra hesapladığı sağlamanın beklenen değere uyup uymadığını bilir. Yükün kendisi ise, ne tür bir işlem olduğunu söyleyen tek baytlık bir işlem koduyla (insert, update ya da delete) başlar; ardından işlem kimliği ve belge kimliği — her ikisi de sekizer bayt — gelir; en sonda da, ekleme ve güncellemelerde belgenin kodlanmış baytları durur. Silme kayıtlarında belge baytı yoktur; yalnızca hangi kimliğin silindiği yazılır. Dosyanın en başında ise, sekiz baytlık küçük bir başlık — bir tanıtıcı imza, bir sürüm numarası ve bayraklar — durur; bu başlık, OxiDB'nin tanımadığı bir biçimdeki dosyayı sessizce yanlış okumak yerine açıkça reddetmesini sağlar.

Şekil 17.2: WAL kaydının bayt düzeni.

Bu biçimin zarif bir ayrıntısı, **noktasal kurtarma** (PITR) ile ilgilidir; ona yirmi üçüncü bölümde döneceğiz, ama tohumu burada atılır. İşlem kodunun en üst biti, kaydın daha zengin bir ikinci sürüm (v2) olup olmadığını ayırır: v2 kayıtları, işlem ve belge kimliğinin ardına, kayda küresel ve tekel olarak artan bir sıra numarası (GSN) ile bir duvar-saati zaman damgası ekler. Bu üst-bit hilesi sayesinde, aynı dosyada hem eski (v1) hem de yeni (v2) kayıtlar karışık bulunabilir ve hepsi tek bir geçişte doğru oynatılabilir; noktasal kurtarma kapalıyken yazılan dosya, eskisiyle bayt bayt aynı kalır. Yani bu yetenek, kapalıyken hiçbir maliyet getirmeyecek biçimde tasarlanmıştır.

Bir yazma geldiğinde, OxiDB önce bu kaydı günlüğe ekler ve onu dayanıklı kılar; ancak ondan sonra, değişikliği bir önceki bölümdeki depolama katmanına — belleğe öncelikli kipte bellekteki eşlemeye, disk-öncelikli kipte `.bdat` dosyasına — ve indekslere uygular. Yani günlük her zaman asıl veriden **önce** dayanıklı olur; çökme tam o sırada olsa bile, niyet günlükte kayıtlıdır.

Burada altıncı bölümün bir inceliğini somut görürüz. Bir kaydı "dayanıklı kılmak", yalnızca diske göndermek değil, oraya gerçekten oturduğundan emin olmaktır; altıncı bölümde, bazı sistemlerde sıradan bir yazmanın veriyi yalnızca işletim sisteminin ya da disk denetleyicisinin önbelleğine bıraktığını söylemiştik. OxiDB bu konuda tavizsizdir: dayanıklılık gerektiğinde, yazdığı veriyi fiziksel ortama gerçekten işleyen bir **boşaltma** çağrısı (fsync ailesinden) yapar ve o çağrı dönene kadar bekler — ancak ondan sonra işlemi "tamamlandı" sayar.[1] Bu güçlü garanti, az sonra göreceğimiz gibi, bir hız bedeli taşır — ama verdiği söz de o kadar güçlüdür.

17.2 Katı dayanıklılık: varsayılan tercih

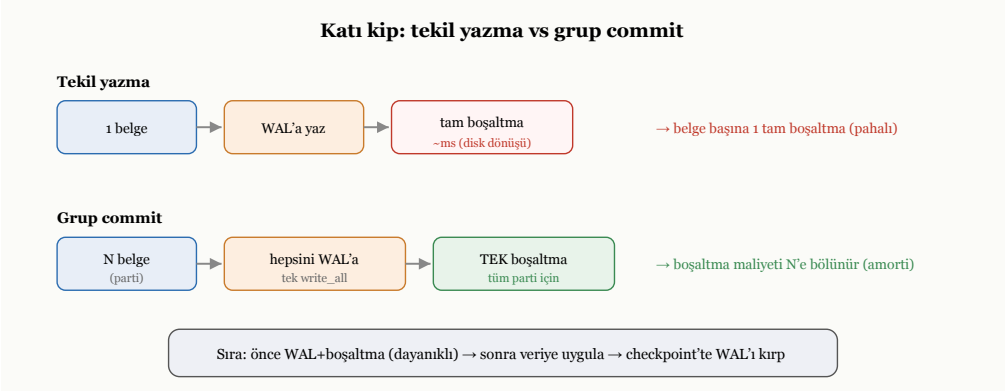
Altıncı bölümde dayanıklılığın bir tayf üzerinde tercih edildiğini söylemiştik. OxiDB, bu tayfın katı ucunu **varsayılan** olarak seçer: her commit, "tamamlandı" denmeden önce günlüğe yazılır ve diske gerçekten bo-

[1] POSIX `fsync`/`fdatasync` — bir dosyaya yazılan verinin işletim sistemi önbelleğinden çıkıp fiziksel saklama ortamına işlendiğini garantileyen çağrılar. Donanım önbellekleri devredeyken tam dayanıklılık için ek platform tedbirleri gerekebilir.

şaltılır. Bu, en güçlü güvencedir — bir işlem tamamlandı dendikten sonra hiçbir çökme onu geri alamaz — ve OxiDB'nin güvenliğe verdiği önceliği yansıtır.

Ama bu güvencenin dürüstçe konuşulması gereken bir maliyeti vardır ve bu maliyet, OxiDB üzerinde yapılan ölçümlerde açıkça görülür. Tek bir belgeyi güncelleyen tek bir işlem düşünün. Bu işlem, tek başına bir günlük yazması ve tek bir tam boşaltma gerektirir; ve tam boşaltma, fiziksel ortama gerçek bir yazma olduğu için, bu makinede milisaniyeler mertebesinde — bilgisayar ölçeğinde upuzun bir süre — alır. Ölçümlerde, katı kipte tek bir belge güncellemesi yaklaşık dört milisaniye sürüyordu. Bu, OxiDB'nin, varsayılan ayarlarıyla MongoDB'nin arkasında kaldığı az sayıdaki işlemden biridir; çünkü MongoDB varsayılan ayarında her yazmayı diske zorlamaz, bellekten onaylayıp günlüğü sonradan, toplu olarak boşaltır. Yani bu karşılaştırma, elma ile armuttur: OxiDB her tekil yazmayı gerçekten dayanıklı kılarken bir bedel öder; MongoDB o bedeli erteler, daha az dayanıklılık karşılığında daha hızlı görünür.

17.3 Grup commit: bedeli toplu yazmada eritmek



Şekil 17.3: Katı kipte tekil yazma ile grup commit'in karşılaştırması.

Peki OxiDB, her yazma için bir tam boşaltma ödüyorsa, çok sayıda belge eklerken nasıl hızlı kalır? Yanıt, altıncı bölümde tanıttığımız grup commit fikrindedir. Bir tam boşaltma, ister bir kaydı ister binlerce kaydı diske işlesin, kabaca aynı süreyi alır. Bu yüzden OxiDB, toplu bir ekleme yaparken, her belgeyi ayrı ayrı boşaltmaz; tüm partiyi günlüğe yazar ve **tek bir boşaltmayla** birden dayanıklı kılar. Böylece tam boşaltmanın maliyeti, partideki belge sayısına bölünür ve belge başına neredeyse hiçe iner.

Bunun somut sonucu çarpıcıdır. Ölçümlerde, beş binlik partiler halinde yapılan toplu eklemede OxiDB, her partiyi tek boşaltmayla dayanıklı kıldığı için, MongoDB ile başa baş ekleme hızına ulaşıyordu — hem de MongoDB'nin yapmadığı bir şeyi, her partiyi gerçekten diske boşaltmayı, yaparak. Yani katı dayanıklılık, toplu yazmalarda bir dezavantaj olmaktan çıkıyordu; çünkü grup commit, güvenceden hiç ödün vermeden maliyeti eritiyordu. Tek tek yazmada görünen dört milisaniyelik fark, toplu yazmada kayboluyordu; çünkü orada amorti edilecek bir parti vardı, tek tek güncellemede ise yoktu.

17.4 Üç aşamalı dayanıklılık: günlük, veri, denetim noktası

Bir yazmanın katı kipte izlediği yolu, dayanıklılık adımlarına ayırarak görmek, OxiDB'nin verdiği güvencenin tam olarak ne olduğunu netleştirir. Sıra üç durakta işler. Önce değişiklik **günlüğe** yazılır ve diske boşaltılır; bu boşaltma, dayanıklılığın asıl çıpasıdır — bu noktadan sonra çökme olsa bile niyet kaybolmaz. Ardından değişiklik **asıl depoya** uygulanır: belleğe öncelikli kipte bellekteki eşlemeye, disk-öncelikli kipte `.bdat` dosyasına. Son olarak, asıl depoya güvenle yansımış değişikliklere karşılık gelen günlük kayıtları, bir **denetim noktasında** (checkpoint) geri kazanılır; böylece günlük baştan büyümeye devam edebilir. Bu üç durağın

her biri, gerektiğinde diskle uzlaşmayı — boşaltmayı — içerir; bu yüzden bu düzene üç aşamalı dayanıklılık demek doğru olur. Ama bu üç adımın hepsinin her yazmada baştan sona, ayrı ayrı boşaltma yapması gerekmez: kritik olan, ilk adımın — günlüğe yazıp boşaltmanın — asıl veriden önce gelmesidir; sonraki adımlar, hızı için, gruplanıp tembelce yapılabilir.

17.5 Gevşek senkronizasyon: hızı seçmek

OxiDB, katı dayanıklılığı varsayılan yapar, ama bunu zorunlu kılmaz. Altıncı bölümdeki tayfın gevşek ucu da, isteğe bağlı bir ortam değişkeniyle açılabilir. Gevşek kipte, bir yazma günlüğe yazılır ama hemen diske boşaltılmaz; boşaltma, her commit'te değil, arka planda çalışan bir iş parçacığı tarafından belirli aralıklarla — bu makinede milisaniyeler mertebesinde bir cadansla — toplu olarak yapılır. Yazma, bellekten onaylanır ve hemen geri döner; günlük kaydı diske gerçekten oturana kadarki o kısa pencere, gevşek kipin satın aldığı hızın karşılığında ödediği risk penceresidir.

Bu kipin hızı, ölçümlerde net biçimde görülür: gevşek kipte, tek bir belge güncellemesi, katı kipteki yaklaşık dört milisaniyeden, yaklaşık seksen yedi mikrosaniyeye iniyordu — yani kırk katından fazla hızlanıyor ve MongoDB'nin varsayılan hızını bile geçiyordu. Çünkü artık her güncelleme bir tam boşaltma beklemiyordu. Bedeli, altıncı bölümde söylediğimiz risktir: bir çökme olursa, henüz boşaltılmamış son birkaç yazma kaybolabilir. Yani gevşek kip, hızı küçük bir veri kaybı riski karşılığında satın alır ve bu, MongoDB'nin varsayılan dayanıklılık modeline çok benzer bir tercihtir. OxiDB'nin yaptığı şey, bu tercihi kullanıcının eline vermektir: güvenlik mi, hız mı — duruma göre seçersiniz.

17.6 Kurtarma: çökmeden geri dönmek

OxiDB'nin dayanıklılık vaadi, asıl sınavını çökmeden sonra verir. Sistem yeniden açıldığında, kendini tutarlı bir duruma getirmek için altıncı bölümdeki kurtarma sürecini uygular; bu süreç, bir önceki bölümdeki iki depolama kipine göre küçük farklarla işler.

Önce bir **taban** kurulur. Belleğe öncelikli kipte bu, `.btree` anlık görüntüsünün belleğe yüklenmesidir. Disk-öncelikli kipte ise, `.bdat` dosyasındaki yaşayan kayıtların taranıp kimlik-konum dizininin yeniden kurulmasıdır. Bu taban hazır olduktan sonra, OxiDB **yazma-öncesi günlüğü taban üzerine oynatır**: günlükteki, henüz tabana yansımamış değişiklikleri yeniden uygular. Böylece, çökmeden hemen önce dayanıklı kılınmış her değişiklik geri gelir.

Bu yeniden oynatmanın güvenli olması, altıncı bölümdeki "etkisiz tekrarlanabilirlik" ilkesine dayanır ve OxiDB bunu somut biçimde sağlar: değişiklikler belge kimliğiyle ilişkilendirildiği için, aynı eklemeyi iki kez oynatmak zararsızdır — ikinci kez, aynı kimliğe yazılır ve sonucu değiştirmez. Bu yüzden kurtarma, bir değişikliğin tabana zaten yansıyıp yansımadığından emin olmasa bile, onu güvenle yeniden uygulayabilir.

Kurtarmanın yarım kalmış son kaydı nasıl ele aldığı, altıncı bölümdeki sağlama toplamı mekanizmasının doğrudan uygulamasıdır. Çökme, en son günlük kaydının yazılmasının ortasında olmuş olabilir; o kayıt yarım kalmış, bozulmuş olabilir. OxiDB, her kaydı oynatırken sağlama toplamını yeniden hesaplar; uyuşmuyorsa, o kaydın yarım kaldığını anlar, onu güvenle atar ve oynatmayı orada durdurur. Bu duruma, dosyanın sonundaki yarım kalmış kayıt anlamında **yırtık kuyruk** (torn tail) denir; kurtarma, yırtık kuyruğa rastladığı an, ondan önceki son sağlam kayda kadar olan her şeyi geçerli sayar ve geri kalanını atar. Bu davranışın doğru olması, günlüğün yalnızca sona eklenerek büyümesine dayanır: bozulma her zaman dosyanın en sonundadır, ortasında değil. Böylece yarım yazma, sessizce bozuk veriye dönüşmek yerine, açıkça fark edilip temizlenir.

17.7 Denetim noktası ve günlüğün dizginlenmesi

Altıncı bölümde, günlüğün sonsuza dek büyüyemeyeceğini ve periyodik olarak denetim noktalarıyla dizginlenmesi gerektiğini söylemiştik. OxiDB de, günlükteki değişiklikler asıl depoya güvenle yansıdıktan sonra, o noktaya kadarki günlük kayıtlarını geri kazanır; böylece günlük baştan büyümeye devam edebilir.

Burada OxiDB'nin bir tasarım inceliği vardır ve onu görmek öğreticidir. Performans için, asıl depoya yazma — yani denetim noktası — tembelce, arada bir yapılır; bu arada, tamamlanmış bir değişikliğin tek dayanağı bir süre yalnızca günlük olabilir. Bu, hız kazandıran bilinçli bir tercihtir; ama kurtarma mantığının buna uygun tasarlanmasını gerektirir, çünkü kurtarma, asıl depoda henüz görünmeyen ama günlükte dayanıklı olan değişiklikleri de geri getirmek zorundadır. OxiDB'nin geliştirilmesinde, bu inceliğin gözden kaçtığı bir durumun nasıl bir kurtarma hatasına yol açtığı ve nasıl düzeltildiği, dayanıklılık mantığının ne kadar dikkat gerektirdiğinin iyi bir örneğidir.

OxiDB, bu denetim noktasını, günlüğün canlı dosyası belli bir boyuta — `OXIDB_WAL_CHECKPOINT_BYTES` ile ayarlanabilen, varsayılan olarak altmış dört megabaytlık bir eşiğe — ulaştığında arka planda kendiliğinden tetikler; böylece aylarca kesintisiz çalışan bir sunucuda bile günlük sınırsızca büyümez. Asıl mesele, bu denetim noktasının **çevrimiçi** (online) olmasıdır: alınırken ne okumaları ne de yazmaları durdurur. Bu, "dünyayı durdurup" günlüğü toparlayan kaba bir bakım değildir; veritabanı denetim noktası boyunca sorgu ve yazma almaya devam eder.

Bunu mümkün kılan, OxiDB'nin **kesip atmak yerine mühürlemek** (seal-not-truncate) diyebileceğimiz bir tasarım tercihidir. Naif bir yaklaşım, denetim noktasında günlüğü kısaltmak — asıl depoya yansımış kayıtları silip dosyayı kesmek — isterdi; ama bu tehlikelidir, çünkü bir yazma, günlük kaydını asıl depoya dokunmadan önce yazar; tam o aralıkta alınan bir anlık görüntü, henüz depoda görünmeyen ama günlükten kesilmiş bir değişikliği kaybedebilir. OxiDB bunun yerine günlüğü kesmez, **mühürler**. Yalnızca kısacık bir an için — uçuştaki yazmaların işini bitirip yeni yazmaların beklediği, atomik bir yeniden adlandırma süresince — bir engel (apply barrier) devreye girer; canlı günlük numaralı bir mühürlü segmente çevrilir ve yerine boş, taze bir günlük açılır. Engel hemen kalkar, yazmalar yeni günlüğe akmaya devam eder; asıl depoya alınan o yavaş anlık görüntü ise engel kalktıktan sonra, hiçbir yazmayı bekletmeden yazılır. Mühürlenen segmentteki her kayıt, tanım gereği artık asıl depodadır; anlık görüntü yazıldıktan sonra o segment güvenle bırakılabilir. Yazmaların duraksadığı tek an, o kısacık yeniden adlandırmadır — yavaş anlık görüntü değil. Her çökme noktası da güvenlidir, çünkü kurtarma yalnızca canlı günlüğü değil, mühürlü segmentleri de tabanın üzerine oynatır: anlık görüntü yazılmadan çökülürse segment eski taban üzerine, yazıldıktan sonra çökülürse yeni taban üzerine — etkisiz tekrarlanabilirlik sayesinde zararsızca — yeniden uygulanır.

17.8 Günlüğün döndürülmesi ve mühürlü segmentler

Az önce gördüğümüz mühürleme, yalnızca denetim noktasını çevrimiçi kılmakla kalmaz; noktasal kurtarmanın da temelini atar. Denetim noktası, asıl depoya yansımış mühürlü segmentleri normalde bırakır; ama OxiDB, noktasal kurtarmayı açan kullanıcılar için onları silmek yerine **arşivlemek** ister. Günlüğün döndürülmesi (rotation) dediğimiz düzenin özü budur: canlı günlük belli bir boyuta ulaştığında güvenli bir biçimde kapatılır — günlük kilidini tutarken, canlı dosya atomik bir yeniden adlandırma ile numaralı bir **mühürlü segmente** (sealed segment) çevrilir ve yerine boş, taze bir canlı günlük açılır. Bu atomikliğin önemi şudur: onaylanmış hiçbir yazma, mühürlenen segment ile yeni canlı günlük arasındaki çatlağa düşemez; her kayıt ya birinde ya ötekindedir. Arka planda çalışan bir arşivleyici, bu mühürlü segmentleri verbatim olarak — baytı baytına — bir arşiv alanına kopyalar ve kendini onaran bir kayıt dosyasıyla (manifest) izler. Daha önce gördüğümüz, kayıtların taşıdığı küresel sıra numarası (GSN) ve zaman damgası, işte burada anlam kazanır: bir yedeğin üstüne, bu segmentleri belirli bir sıra numarasına ya da zaman noktasına kadar yeniden oynatarak, geçmişin tutarlı bir kesitine dönmek mümkün olur. Bu, yirmi üçüncü bölümün konusudur; burada yalnızca, günlüğün bu kurtarma yeteneğinin de temeli olduğunu görmek yeterli.

17.9 WAL her iki kibin de önünde durur

Altıncı bölümde, yazma-öncesi günlüğün depolama felsefesinin bir alternatifi değil, onun önünde duran ayrı bir katman olduğunu vurgulamıştık. OxiDB'de bu, açıkça görülür: aynı günlük ve aynı dayanıklılık makinesi, hem belleğe öncelikli hem de disk-öncelikli kipin önünde durur. Hangi kipte olursanız olun, bir yazma önce aynı günlüğe gider; yalnızca o yazmanın "asıl depoya" uygulanma biçimi — bellekteki eşlemeye mi yoksa `.bdat` dosyasına mı — kipe göre değişir. Böylece dayanıklılık, depolama kipinden bağımsız, ortak bir güvence olarak kalır.

17.10 Bu bölümün bıraktığı yer

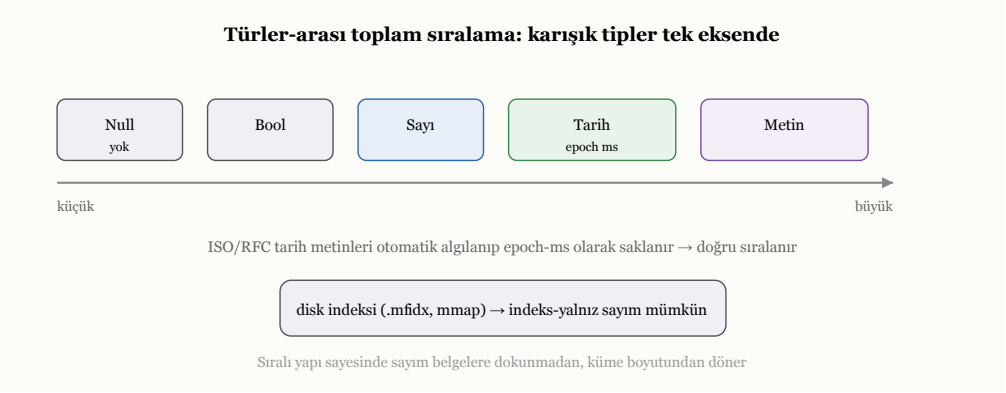
Bu bölümde, OxiDB'nin dayanıklılık mekanizmasını yakın plana aldık. Her koleksiyonun, sağlama toplamlı ve işlem kimlikli kayıtlardan oluşan bir yazma-öncesi günlük tuttuğunu; her yazmanın önce bu günlüğe gidip dayanıklı kılındığını, sonra depoya uygulandığını gördük. Kaydın bayt düzeyindeki biçimini — sağlama toplamlı çerçeve, işlem kodlu yük, ve noktasal kurtarmayı açan v2 uzantısı — yakından gördük. Katı dayanıklılığın varsayılan olduğunu ve bunun tek tek güncellemelerde bir hız bedeli taşıdığını, ama grup commit sayesinde toplu yazmalarda bu bedelin eridiğini; gevşek kipin ise hızı bir kayıp riski karşılığında nasıl artırdığını ölçümlerle gördük. Kurtarmanın taban kurma, günlük oynatma, etkisiz tekrarlanabilirlik ve yırtık-kuyruk temizleme adımlarını; üç aşamalı dayanıklılık sırasını (günlük, veri, denetim noktası); ve günlüğün mühürlü segmentlere döndürülerek noktasal kurtarmaya nasıl zemin hazırladığını inceledik.

Artık verimiz hem kalıcı hem de çökmeye dayanıklı biçimde duruyor. Ama yedinci bölümde gördüğümüz gibi, veriyi güvenle saklamak yetmez; onu taramadan hızla bulabilmek de gerekir. Bir sonraki bölümde, OxiDB'nin aramayı hızlandıran yapılarına — alan indekslerine, bileşik indekslere ve disk-öncelikli kipte belleğe yansıtılan indekslere — ve bunların yedinci bölümdeki ilkelerle nasıl örtüştüğüne eğiliyoruz.

Bölüm 18

OxiDB'de İndeksler: Alan, Bileşik ve mmap Tabanlı Disk İndeksleri

Önceki iki bölümde, OxiDB'nin veriyi nasıl sakladığını ve onu çökmeye karşı nasıl güvence altına aldığını gördük. Artık verimiz hem kalıcı hem dayanıklı; ama yedinci bölümde öğrendiğimiz gibi, onu taramadan hızla bulabilmek de gerekir. Bu bölüm, OxiDB'nin aramayı hızlandıran yapılarını — alan indekslerini, bileşik indeksleri ve disk-öncelikli kipte belleğe yansıtılan indeksleri — ele alıyor ve bunların yedinci bölümdeki ilkelerle nasıl örtüştüğünü gösteriyor.



Şekil 18.1: Türler-arası toplam sıralama ve mmap tabanlı disk indeksi.

18.1 Alan indeksleri: sıralı bir eşleme

Yedinci bölümde bir indeksi, "bir alanın değerlerinden o değere sahip belgelerin konumuna giden bir eşleme" olarak tanımlamıştık. OxiDB'nin alan indeksleri tam olarak budur: indekslenen her alan için, o alanın her değerinin altında, o değere sahip belgelerin kimliklerini tutan bir yapı. Bu yapı **sıralı** tutulur ve bu, yedinci bölümdeki üç armağanı birden getirir: belirli bir değere sahip kayıtları bulmak (eşitlik), bir değer aralığındaki kayıtları bulmak (aralık) ve sonuçları o alana göre sıralı vermek (sıralı getirme).

OxiDB, bu sıralı yapıyı, bellekte verimli çalışacak biçimde tasarlar — ve burada bilinçli bir mühendislik tercihi vardır. Bu indeks, adı çoğu sistemde anılan düğüm-ve-işaretçi ağacı (B-ağacı ya da denge ağacı) **değildir**. OxiDB, indeks değerlerini, işaretçilerle birbirine bağlı dağınık bir ağaç yerine, tek bir bitişik ve sıralı dizide tutar: her girdi bir değer ile o değere sahip belgelerin kimlik kümesidir, ve bu girdiler değere göre sıralı, yan

yana, aynı bellek bölgesinde durur. Böylece bir değeri ararken, OxiDB bu dizide **ikili arama** (binary search) yapar; belleğin oraya buraya sıçramadan, birbirine yakın bölgelerde ilerlemesi yeterli olur. Bunun neden önemli olduğu, ölçeği büyütünce ortaya çıkar: bir milyon girdilik bir indekste ikili arama, yalnızca yirmi mertebesinde adım atar ve bu adımlar, dağınık bir ağacın işaretçi zincirini kovalamasının aksine, modern işlemcilerin önbellek satırlarına çok daha iyi oturur. Bu yapıya, içeride sayfa-dostu alan indeksi (PagedFieldIndex) denir.

Ama bitişik ve sıralı bir dizinin bir zayıf noktası vardır: araya tek bir yeni değer sokmak, kötü durumda dizinin yarısını kaydırmayı gerektirebilir. OxiDB bunu, ikinci kısımda LSM ağaçlarında gördüğümüz fikrin küçük bir akrabasıyla çözer: yeni yazmalar doğrudan sıralı diziye işlenmez, önce küçük bir **yazma tamponuna** (write buffer) alınır. Tampon, sırasız ve küçüktür; ekleme ona bedavaya yakın bir maliyetle düşer. Tampon belli bir doluluğa — birkaç yüz ila bin mertebesinde bir eşiğe — ulaştığında, OxiDB onu bir kerede sıralı diziye **birleştirir** (merge): bekleyen tüm ekleme ve silmeleri tek geçişte uygular. Böylece her ekleme tek tek diziyi kaydırmaz; maliyet, birleştirmeyi bekleyen birçok yazmaya bölünür ve ekleme, amorti edilmiş biçimde ucuz kalır. Bir değer ararken hem sıralı diziye hem de küçük tampona bakılır; bu yüzden tampondaki güncel yazmalar da sonuçlara yansır.

Şekil 18.2: Yazma tamponunun sıralı diziye birleşmesi.

18.2 Türler arası sıralama: IndexValue inceliği

Yedinci bölümde "sıralı indeks" derken, üstü örtük bir varsayım vardı: değerlerin sıralanabilir olması. İlişkisel bir sütunda bu kolaydır, çünkü her sütunun tek bir türü vardır. Ama belge dünyasında, dördüncü bölümde gördüğümüz gibi, aynı alan farklı belgelerde farklı türde değerler taşıyabilir — birinde sayı, birinde metin, birinde tarih. Sıralı bir indeks kurabilmek için, OxiDB'nin bu **farklı türleri bile tek bir bütünsel sıraya** koyabilmesi gerekir.

OxiDB bunu, türler arasında sabit ve kesin bir sıralama tanımlayarak çözer: en önce boş değerler (null), sonra mantıksal değerler (bool), sonra sayılar — tam ve ondalık sayılar kendi aralarında doğal sayısal sıralarıyla — sonra tarihler ve en sonda metinler gelir. Bu sıra, indeksin iç değer türünde (IndexValue) gömülüdür: iki değer karşılaştırılırken önce türleri bu basamağa göre, tür aynıysa değerleri kendi içinde kıyaslanır. Böylece tür karışık olsa bile — aynı alan bir belgede sayı, başkasında metin olsa bile — indeks tutarlı, tek bir bütünsel sıra koruyabilir ve aralık sorguları anlamını yitirmez. Özellikle zarif bir ayrıntı, tarihlerle ilgilidir. Tarihler belgelerde çoğu zaman metin olarak — yaygın tarih biçimlerinde — yazılır; ama metin olarak sıralandığında, tarihler doğru kronolojik sıraya girmeyebilir. OxiDB, yaygın biçimlerdeki — ISO 8601 / RFC 3339 ya da yıl-ay-gün gibi — tarih metinlerini **otomatik olarak tanır** ve onları, dönemden bu yana geçen milisaniye sayısına (epoch-ms) çevirip bir tam sayı olarak indeksler. Bunun iki kazancı vardır: tarih karşılaştırması artık metin karşılaştırması değil, çok daha hızlı bir tam sayı karşılaştırmasıdır; ve tarihler, metin olarak yan yana dizildiğinde bozulabilecek kronolojik sıraya, sayısal değerleriyle her zaman doğru girer. Böylece tarih

aralığı sorguları doğru biçimde sıralanır. Bu, yedinci bölümün soyut "sıralı indeks" fikrini, belge verisinin tür çeşitliliğine uydurmak için yapılmış somut bir mühendislik tercihidir.

18.3 Bileşik indeksler ve önek kuralı

Yedinci bölümde, birden çok alanı birlikte süzen sorgular için bileşik indeksleri ve onların önek kuralını tanımıştık. OxiDB, birden çok alanın birleşimi üzerine kurulu bileşik indeksleri destekler ve bunlar, tıpkı yedinci bölümdeki telefon rehberi gibi davranır: alanların belirli bir sırayla dizildiği bu indeks, sıralamanın baştan başlayan bir önekini kullanan sorgulara yarar. Böylece "şu bölgedeki, şu yaş aralığındaki" gibi çok koşullu sorgular, tüm koleksiyonu taramak yerine, doğrudan bileşik indeksten hızlıca yanıtlanır.

18.4 İndeksten yanıt: belgeye dokunmadan saymak

Yedinci bölümdeki kapsayan indeks fikrinin — sorgunun ihtiyaç duyduğu her şeyi indeksin tek başına sağlaması — OxiDB'deki en çarpıcı uygulaması, saymadır. "Şu değere sahip kaç belge var" ya da indeksli bir alana göre "her grupta kaç belge var" sorularını yanıtlamak için, OxiDB belgelerin hiçbirine dokunmaz; indeksteki her değerin altında kaç kimlik olduğunu zaten bildiği için, yalnızca bu sayıları okur. Dokuzuncu bölümde, indeksli bir alana göre yalnızca sayan gruplamaların belgelere hiç dokunmadan yanıtlanabileceğini söylemiştik; işte bunun somut karşılığı budur. Ölçümlerde bu, OxiDB'nin en belirgin üstünlüklerinden biriydi: indeksli sayım, belgeleri okuyan bir yaklaşıma kıyasla onlarca kat hızlıydı.

Bu hızlı yolun OxiDB'nin gelişimindeki öyküsü, mühendislik tercihlerinin nasıl incelikler içerdiğini güzel gösterir. Bu sayım kestirmesi, başlangıçta yalnızca belleğe öncelikli indekslerde çalışıyordu; disk-öncelikli, belleğe yansıtılmış indeksler için ise devreye girmiyor, sistem belgeleri taramaya düşüyordu. Bu, disk-öncelikli kipte sayım gruplamalarını gereksiz yere yavaşlatıyordu. Bu hızlı yolun disk-öncelikli indeksleri de kapsayacak biçimde yeniden etkinleştirilmesi, o yükleri yeniden onlarca kat hızlandırdı.

Aynı fikir, saymanın ötesine de geçer. Bir bileşik indeks, bir gruplamanın ihtiyaç duyduğu tüm alanları — hem gruplama anahtarını hem de toplanan değeri — kapsıyorsa, OxiDB o gruplamayı da belgelere hiç dokunmadan, yalnızca indeksi yürüyerek yanıtlayabilir: her grubun toplamını, ortalamasını, en küçüğünü ya da en büyüğünü indeksteki değerlerden doğrudan hesaplar. Bu kapsayan yol, gruplama tüm koleksiyonu tarasa da, bir koşulla — bileşik indeksin önekine binen bir süzgeçle — daraltılsa da işler; eşitlik öneki, saf aralık, eşitlik artı aralık ve çoklu-eşitlik biçimindeki süzgeçlerin hepsi bu kapsamaya girer. Toplama işleyicisini yirminci bölümde ele alırken, bu indeks-yalnız yola oradan da değineceğiz.

18.5 İndeks destekli sıralama ve erken sonlanma

Sekizinci bölümde, sıralı bir indeksin "şu alana göre sıralı ilk on kayıt" sorusunu nasıl hızlandırdığını ve erken sonlanmanın gücünü görmüştük. OxiDB, bunu doğrudan uygular: böyle bir sorgu, ilgili alanın sıralı indeksini baştan gezerek, onuncu kaydı bulduğu an durur; geri kalanı hiç üretmez. Milyonlarca kayıt arasından en büyük ya da en küçük birkaçını bulmak, bu sayede neredeyse anlık olur.

Burada da, OxiDB'nin gelişiminden öğretici bir öykü vardır. Disk-öncelikli kipte, indeks destekli sıralamanın dayandığı iç gezinme, belleğe yansıtılmış indekslerde boş sonuç veriyordu; yani disk-öncelikli bir koleksiyonda sıralı sorgular, sessizce hiçbir şey döndürmüyordu — bir hata, üstelik fark edilmesi zor bir hata, çünkü sistem çökmüyor, yalnızca yanlışlıkla boş sonuç veriyordu. Bu durumun fark edilip düzeltilmesi, hem doğru sonuçları geri getirdi hem de bize, sessiz yanlışlıkların çöken sistemlerden daha tehlikeli olabileceğini bir kez daha hatırlattı. Bu yüzden bu davranış, sonradan, her iki kipte de çalıştığını güvence altına alan bir sınama ile korumaya alındı.

18.6 Disk-öncelikli indeksler: indeksi de bellekten çıkarmak

On üçüncü ve on altıncı bölümlerde, disk-öncelikli kipin belge gövdelerini bellekten diske taşıyarak bellek ayak izini küçülttüğünü gördük. Ama bir incelik kalmıştı: indeksler de bellekte yer kaplar ve büyük bir koleksiyonda birçok indeks, kayda değer bir bellek tüketebilir. OxiDB, disk-öncelikli kipte bu sorunu da çözer: indeksleri de diske taşır.

Önemli olan, bunun yeni bir indeks türü olmamasıdır: az önce tanıdığımız aynı bitişik-sıralı-dizi artı yazma-tamponu tasarımı, yalnızca farklı bir yerde yaşar. Bu kipte, indeksin sıralı dizisi bellekte değil, kendi belleğe yansıtılmış dosyasında — uzantısıyla anılırsa `.mfidx` dosyasında — durur ve gerektiğinde sayfa sayfa belleğe getirilir; bellekte yerleşik kalan tek şey, son yazmaları tutan o küçük yazma tamponudur. Aynı düzen bileşik indeksler için de geçerlidir: onlar da disk-öncelikli kipte kendi belleğe yansıtılmış dosyalarında — `.mcidx` dosyalarında — yaşar, yalnızca güncel yazmaları tutan küçük tamponları bellekte kalır. Böylece hem tek alanlı hem çok alanlı indeksler, disk-öncelikli felsefeyi eksiksiz izler. İkili arama, artık bu yansıtılmış dosya üzerinde yürür; aranan değerin bulunduğu sayfalar işletim sistemi tarafından getirilir, bellek baskı altındayken yine sessizce geri atılabilir — tıpkı bir önceki bölümde belge gövdeleri için gördüğümüz gibi. Daha da önemlisi, veritabanı yeniden açıldığında, bu indeksler baştan kurulmaz; dosya doğrudan belleğe yansıtılarak neredeyse anında yüklenir. Bunun sonucu, on üçüncü bölümün disk-öncelikli felsefesinin indekslere kadar uzanmasıdır: beş yüz bin belgelik ve birkaç indeksli bir koleksiyonu taze açan bir süreç, yalnızca birkaç megabayt yerleşik bellekle açılır — hem belge gövdeleri hem de indeksler, yerleşik bellekten çıkmış, gerektiğinde diskten getirilen, geri alınabilir veriye dönüşmüştür.

Dürüst olmak gerekir ki bu yaklaşımın sınırları da vardır. Bir indeksi ilk kez **kurmak** — sıfırdan oluşturmak — yine de tüm veriyi gezmeyi ve bir miktar geçici bellek kullanmayı gerektirir; bu, sürekli değil, kuruluş anına özgü bir yüktür. Ayrıca bazı indeks türleri henüz tümüyle belleğe yansıtılmış değildir. Bunlar, disk-öncelikli kipin zamanla olgunlaşan, ama temel kazancını — yerleşik belleği veriyle birlikte büyümekten kurtarmayı — bugün de sunan yönleridir.

18.7 Diğer indeks türleri

OxiDB'nin indeks ailesi, temel alan ve bileşik indekslerin ötesine geçer ve yedinci bölümde değindiğimiz birkaç türü daha kapsar. **Benzersizlik indeksleri**, bir alanda aynı değerin iki kez yazılmasını engeller; yani yedinci bölümde saydığımız o yan faydayı — aramayı hızlandırırken bir bütünlük kuralı dayatmayı — sunar. **Süreyle dolan indeksler**, belgelere bir ömür biçer: belirli bir süre sonra dolan kayıtlar, arka planda çalışan bir süreç tarafından otomatik olarak temizlenir; oturum verisi ya da geçici kayıtlar için biçilmiş kaftandır. **Tam metin indeksleri**, yedinci bölümdeki ters indeks fikrini hayata geçirir ve metin içinde sözcük aramayı, alaka puanlamasıyla birlikte mümkün kılar; bunlara yirmi üçüncü bölümde döneceğiz. **Vektör indeksleri** ise, tam eşleşme yerine "benzerlik" aramasını — birbirine yakın anlamlı ya da yakın özellikli kayıtları bulmayı — destekler.

18.8 İndekslerin bedeli ve dayanıklılığı

Yedinci bölümün iki dersini OxiDB bağlamında tekrar etmekte yarar var. Birincisi, indekslerin bedavaya gelmediğidir: her yazma, yalnızca belgeyi değil, o belgeyi ilgilendiren her indeksi de güncellemek zorundadır. Bir koleksiyonda ne kadar çok indeks varsa, her yazma o kadar çok ek iş doğurur; bu yüzden OxiDB'de de, "her ihtimale karşı her şeyi indeksleme" değil, erişim örüntülerine göre seçici indeksleme doğru yaklaşımdır.

İkincisi, indekslerin de dayanıklı olması gerektiğidir. OxiDB, indeksleri diske kalıcı kılar; disk-öncelikli kipte yeniden açılışta belleğe yansıtarak anında yükler, belleğe öncelikli kipte ise gerektiğinde yeniden kurar ya da kayıtlı biçiminden okur. Bir önceki bölümdeki kurtarma süreciyle el ele, indekslerin asıl veriyle tutarlı kalması — bir çökme sonrası bile — sessiz ama kritik bir görevdir.

18.9 Bu bölümün bıraktığı yer

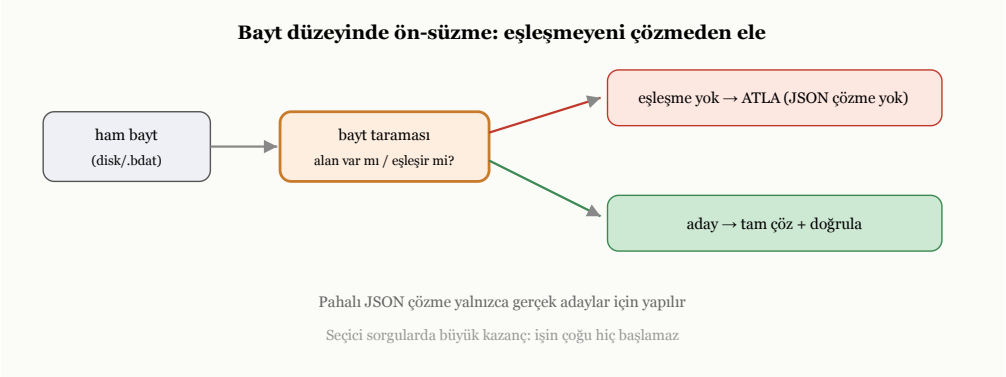
Bu bölümde, OxiDB'nin aramayı hızlandıran yapılarını yakın plana aldık. Alan indekslerinin, bitişik-sıralı bir dizi, ikili arama ve birleştirilen bir yazma tamponundan oluşan bellek-dostu bir yapıyla eşitlik, aralık ve sıralı getirmeyi birden desteklediğini; türler arası kesin sıralamanın ve tarihleri epoch-ms tam sayısına çeviren otomatik tanımanın bu sıralamayı belge verisine nasıl uyarladığını; bileşik indekslerin önek kuralını; indeksten doğrudan saymanın ve toplamanın gücünü; indeks destekli sıralamayı ve erken sonlanmayı; disk-öncelikli kipte indekslerin de belleğe yansıtılarak bellekten çıkarıldığını; ve diğer indeks türlerini gördük. Bu arada, OxiDB'nin gelişiminden iki öğretici öyküyü — disk indekslerinde sayım kestirmesinin yeniden etkinleştirilmesini ve disk sıralamasındaki sessiz boş-sonuç hatasının düzeltilmesini — bu mekanizmaların ne kadar dikkat gerektirdiğinin örnekleri olarak izledik.

Ama indeksler, yedinci bölümde söylediğimiz gibi, tek başlarına yalnızca araçtır. Bir kullanıcının sorusunu alıp hangi indeksin işe yarayacağına karar veren, veriyi en az iş yaparak süzecek bir plana dönüştüren akıl, sorgu işleyicidir. Bir sonraki bölümde, OxiDB'nin sorgu motorunu — operatörlerini, indeks destekli yollarını ve eşleşmeyen belgeleri hiç çözmeden atlayan o ince bayt düzeyinde süzme tekniğini — sekizinci bölümdeki ilkelerle bağlayarak ele alacağız.

Bölüm 19

OxiDB'nin Sorgu Motoru: Operatörler, İndeks Destekli Yollar ve Bayt Düzeyinde Süzme

Önceki bölümde, OxiDB'nin aramayı hızlandıran indekslerini tanıdık ve yedinci bölümde olduğu gibi, indekslerin tek başına yalnızca araç olduğunu söyledik. Bir kullanıcının sorusunu alıp hangi indeksin işe yarayacağına karar veren, veriyi en az iş yaparak süzecek bir plana dönüştüren akıl, sorgu işleyicidir. Bu bölüm, OxiDB'nin sorgu motorunu — desteklediği operatörleri, indeks destekli yürütme yollarını ve eşleşmeyen belgeleri hiç çözmeden atlayan o ince bayt düzeyinde süzme tekniğini — sekizinci bölümdeki ilkelerle bağlayarak ele alıyor.



Şekil 19.1: Byte düzeyinde ön-süzme: eşleşmeyeni çözmeden eleme.

19.1 Sorgunun ayrıştırılması ve operatör dağarcığı

Sekizinci bölümde, sorgu işlemenin ilk adımının, ham sorguyu koşulların ve mantıksal bağların bir ağacına çevirmek olduğunu görmüştük. OxiDB de tam olarak bunu yapar: gelen JSON tabanlı sorguyu, üzerinde akıl yürütebileceği yapısal bir koşul ağacına ayrıştırır. Bu ağaçtaki koşullar, sekizinci bölümde saydığımız zengin çeşitliliği yansıtır.

En temelde **karşılaştırma** koşulları vardır: bir alanın bir değere eşit ya da eşit olmaması; bir değerden büyük, küçük ya da bir aralıkta olması. **Üyelik** koşulları, bir alanın belirli bir değerler kümesine girip girmediğini sorar. **Varlık** koşulu, bir alanın belgede bulunup bulunmadığını denetler. Belgeler iç içe ve listeli

olabildiği için, belgeye özgü koşullar da vardır: bir listenin belirli bir öğeyi içermesi, belirli bir uzunlukta olması, belirli koşullara uyan bir öğe barındırması; bir alanın belirli bir türde olması; bir metnin bir kalıba uyması. Tüm bu koşullar, **mantıksal bağlarla** — ve, veya, ne…ne, değil — birleştirilebilir. Son olarak, bir alanı başka bir alanla karşılaştıran, yani tek bir belge içinde alanlar arası bir ilişki kuran koşullar da vardır. Bu dağarcık, sekizinci bölümün soyut "koşul ağacı"nı, belge verisinin tüm zenginliğiyle somutlaştırır.

19.2 İndeksli koşullar ile süzgeç koşulları

OxiDB'nin sorgu motorunu anlamanın anahtarı, bu koşulları **iki sınıfa** ayırmaktır. Bazı koşullar bir indeksten yararlanabilir; bazıları yararlanamaz. Eşitlik, aralık ve üyelik gibi koşullar, eğer ilgili alanın bir indeksi varsa, indeks üzerinden hızlıca az sayıda aday belgeye inebilir. Buna karşılık, bir listenin belirli koşullara uyan bir öğe barındırıp barındırmadığı, bir alanın türü, bir sayının belirli bir bölme kalanına sahip olup olmadığı, alanlar arası bir karşılaştırma ya da bir olumsuzlama gibi koşullar, doğaları gereği bir indeksle yanıtlanamaz; bunlar yalnızca belgeleri tek tek inceleyen bir **süzgeç** olarak uygulanabilir.

İşte sekizinci bölümdeki "indeks daraltır, süzgeç inceltir" iş bölümü, OxiDB'de tam olarak bu ayrım üzerine kuruludur. OxiDB, bir sorguyu aldığında onu, indeksle çözülebilen bir parça ile yalnızca süzgeçle çözülebilen bir parçaya ayırır. İndeksli parçayı kullanarak az sayıda aday belge belirler; sonra, kalan süzgeç koşullarını yalnızca bu adaylar üzerinde dener. Bir milyon belgeyi taramak yerine, indeksin daralttığı küçük aday kümesini süzmek, sekizinci bölümde gördüğümüz gibi kıyaslanamayacak kadar ucuzdur.

Bu ikili sınıflandırmayı somutlaştırmakta yarar var; çünkü hangi operatörün hangi sınıfa düştüğü, OxiDB'nin yürütme planını doğrudan belirler. **İndeksli** sınıfa şunlar girer: eşitlik, dört yönlü aralık (büyük/büyük-eşit/küçük/küçük-eşit) ve üyelik. Bunların ortak yanı, bir alanın **tek bir değerine** ya da **bitişik bir değer aralığına** karşılık gelmeleri, yani bir B-ağacı indeksinde tek bir nokta ya da tek bir aralık olarak ifade edilebilmeleridir. **Süzgeç-yalnız** sınıfa ise eşitsizlik (bir değere eşit *olmama*), listeye-koşullu-öğe-arama, "tümünü içerir", liste uzunluğu, alan türü, bölme kalanı, olumsuzlama, ne…ne bağı ve alanlar arası karşılaştırma girer. Bunların ortak yanı da, bir indeksin verdiği "şu değer şu belgelerde" eşlemesiyle yanıtlanamamalarıdır: bir liste uzunluğu indekste tutulmaz; bir olumsuzlama, indeksin sıralı yapısından yararlanamaz; bir eşitsizlik, o alanın hiç bulunmadığı belgeleri de kapsamak zorunda olduğu için — ki indeks yalnızca alanı taşıyan belgeleri tutar — indeksle tam karşılanamaz ve bu yüzden süzgeç tarafına düşer; alanlar arası bir karşılaştırma ise, tek bir alanın indeksiyle değil, belgenin iki alanının aynı anda görülmesiyle çözülür. Varlık koşulu da, kavramsal olarak indekslenebilir görünse de, OxiDB onu bu sürümde süzgeç tarafında değerlendirir. Bu ayrım keyfi değildir; her operatörün doğasından çıkar ve sorgu motorunun bütün plan mantığı bunun üzerine oturur.

19.3 Yürütme: indeks, süzgeç ve tarama

Bu ayrımdan, sekizinci bölümdeki yürütme mantığının somut karşılığı doğar. OxiDB, bir sorguyu yürütürken önce indeksli parçayı çalıştırıp aday belgelerin kimliklerini elde eder. Eğer indeks, sorgunun **tüm** koşullarını zaten karşılıyorsa — yani süzgeçle inceltilecek bir şey kalmamışsa — adaylar doğrudan sonuçtur ve hiçbir ek denetim gerekmez. Aksi halde, kalan süzgeç koşulları adaylar üzerinde uygulanır ve yalnızca uyanlar sonuca alınır.

Sorgunun hiçbir koşulu indeksli bir alana denk gelmezse, sekizinci bölümde gördüğümüz son çare devreye girer: tarama. OxiDB tüm belgeleri gezer ve her birinde süzgeç koşullarını dener. Bu, kaçınılması istenen tam tarama maliyetidir; ama uygun bir indeks yoksa başka yol yoktur. İşte tam burada, OxiDB'nin en ayırt edici tekniği devreye girer.

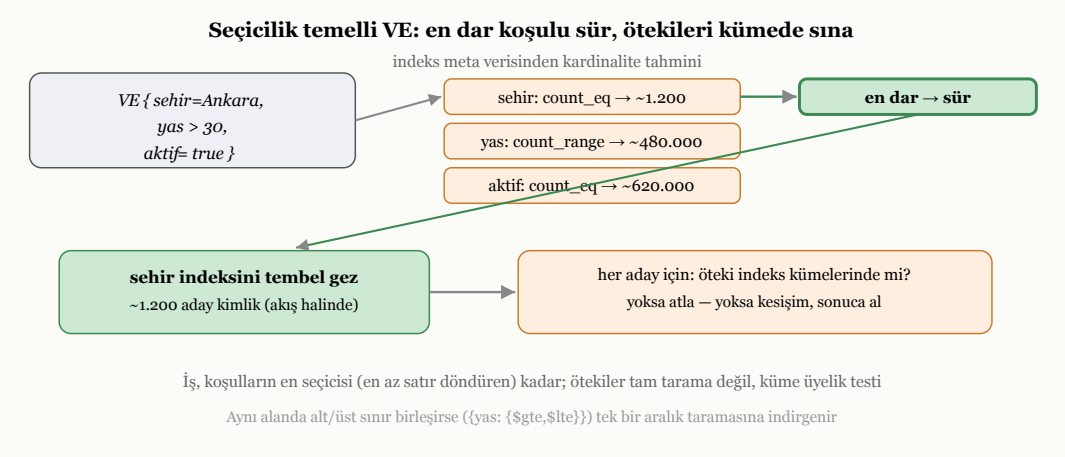
19.4 Seçicilik temelli koşul sıralaması

Bir VE (AND) sorgusunda birden çok koşulun, üstelik birden çoğunun indeksli olduğu sık görülen bir durumdur — örneğin "şehri Ankara olan, yaşı 30'dan büyük ve aktif olan kullanıcılar". Her üç alanın da bir indeksi varsa, OxiDB hangisini kullanmalıdır? Bu seçim, sorgunun hızını kat kat değiştirebilir; çünkü amaç, **en az sayıda aday** üreten koşulu sürücü (driving) yapmaktır. Eğer "aktif=true" bir milyon kullanıcının altı yüz binini, "şehir=Ankara" ise yalnızca binini döndürüyorsa, şehir indeksini sürmek bin adayla başlamak, aktiflik indeksini sürmek ise altı yüz bin adayla başlamak demektir. İlki, ikincisinden yüzlerce kat daha az iş yapar.

OxiDB bu kararı, **seçicilik** (selectivity) temelinde verir. Her indeksli alt koşul için, indeks meta verisinden ucuz bir **kardinalite tahmini** (cardinality estimate) çıkarır: bir eşitlik koşulu için, o değere kaç belgenin denk geldiğini indeksten sayar; bir aralık koşulu için, aralığın iki ucu arasına kaç belgenin düştüğünü, B-ağacının sıralı yapısından yararlanarak sayar; bir üyelik koşulu için ise kümedeki her değerin sayımlarını toplar. Bu sayımlar, belgeleri hiç çözmeden, yalnızca indeksin kendi sayaçlarından elde edilir; dolayısıyla tahmin neredeyse bedavadır. OxiDB, tahmini en küçük olan — yani en az aday döndüren — koşulu **sürücü** seçer ve onu tembel (lazy) biçimde, aday kimliklerini akış halinde üreterek gezer.

Kalan indeksli koşullar ise birer **kesişim süzgeci** olarak iş görür. OxiDB, bu diğer koşulları kendi indekslerinden kimlik kümeleri olarak çıkarır; sonra, sürücü koşulun ürettiği her aday kimliğini bu kümelerde sorgular: aday tüm kümelerde varsa kesişime girer ve sonuca alınır, yoksa atlanır. Önemli olan şudur: bu kesişim sınaması, tam tarama değildir — yalnızca sürücünün ürettiği az sayıdaki aday üzerinde, küme üyelik testleriyle yapılır. Böylece toplam iş, koşulların **en seçicisi** kadar küçük kalır.

Bir özel durum, bu mekanizmayı daha da keskinleştirir. Eğer bir VE sorgusunda **aynı alanın** hem alt hem üst sınırı verilmişse — örneğin "yaş 25 ile 35 arasında" — OxiDB bunu iki ayrı koşul olarak değil, tek bir birleşik aralık taraması olarak tanır ve indeksin o aralığını bir kerede gezer. Bu, aynı alana iki kez bakıp sonuçları kesiştirmekten daha doğrudan ve hızlıdır.



Şekil 19.2: En seçici koşulu sürücü seçmek ve diğerlerini kesişim süzgeci yapmak.

Burada, sekizinci bölümün bir sorgu eniyileyicisi (query optimizer) için çizdiği soyut resmin somut bir köşesini görürüz: motor, kullanıcının yazdığı koşul sırasına körü körüne uymaz; koşulları, beklenen maliyetlerine göre kendisi yeniden sıralar. OxiDB'nin bu yeniden sıralaması, klasik bir maliyet temelli eniyileyicinin (cost-based optimizer) sade ama etkili bir biçimidir: tek bir ölçüt — beklenen aday sayısı — üzerinden, en ucuz başlangıç noktasını seçer.

19.5 İki indekssiz hızlı yol: sayım ve indeks destekli sıralama

Tarama her zaman son çare değildir; bazı sorgular, belgelere hiç dokunmadan, yalnızca indeksin yapısından yanıtlanabilir. Bunun en saf örneği **sayımdır**. "Şu koşula uyan kaç belge var?" sorusu, eğer koşul indeksliyse, belgelerin içeriğine bakmayı hiç gerektirmez: OxiDB, indeksteki ilgili nokta ya da aralığa düşen kimliklerin **sayısını** doğrudan okur. Bu, az önceki kardinalite tahmininin aynı mekanizmasıdır — ama burada tahmin değil, kesin yanıttır. Bir milyon belgeli bir koleksiyonda "kaç tane aktif kullanıcı var?" sorusu, tek bir belge çözülmeden, indeksin sayacından milisaniyenin altında yanıtlanabilir.

İkinci hızlı yol, **indeks destekli sıralamadır**. Sekizinci bölümde, sıralamanın doğası gereği pahalı — tüm sonucu toplayıp karşılaştırarak dizmeyi gerektiren — bir iş olduğunu söylemiştik. Ama eğer sıralama alanının bir indeksi varsa, OxiDB sıralamayı hiç yapmaz: indeksin kendisi zaten o alana göre sıralı tutulduğu için, indeksi baştan (artan) ya da sondan (azalan) gezmek, belgeleri doğrudan sıralı düzende verir. Üstelik bir üst sınır (limit) verilmişse, OxiDB indeksi yalnızca o sınır kadar ilerletir ve durur. Yani "en yeni 20 kayıt" gibi bir sorgu, bir milyon belgeyi sıralamak yerine, indeksin yalnızca ilk 20 girdisine dokunur. Bu, sekizinci bölümde değindiğimiz, sıralama maliyetini belge sayısından (n log n) yalnızca istenen sonuç sayısına (sınır kadar) indiren önemli bir eniyilemedir. OxiDB hem tekil alan indekslerinde hem de bileşik (composite) indekslerde bu sıralı-gezinme yolunu kullanır.

Bu sıralı-gezinme yolunun bile ilk biçimi, göründüğünden fazla iş yapıyordu: motor, bir üst sınır verilmiş olsa da koleksiyonun tümünü bir kez sayıyor ve sıralı düzeni baştan sona kuruyor, sınırı ancak ondan sonra uyguluyordu — yani "en yeni 20 kayıt" için bile bir milyon belgeyi dokunuyordu. Bu kitap yazılırken bu iki gizli maliyet de giderildi: sayım artık indeksin sayacından doğrudan, sabit zamanda okunuyor; sıralı gezinme ise gerçekten tembel (lazy) hale getirilerek yalnızca istenen sınır kadar girdiye dokunuyor. Etkisi ölçülebilirdi — böyle bir sorgu, milisaniyeler mertebesinden milisaniyenin çok altına indi. Bu da, sekizinci bölümde değindiğimiz "istenen sonuç sayısı kadar iş" hedefinin, ancak son gizli maliyetler de temizlendiğinde tam olarak yakalanabildiğini gösteren küçük bir derstir.

19.6 Bayt düzeyinde süzme: çözmeden elemek

Sekizinci bölümde, tarama ya da süzme sırasında gözden kaçan ama kritik bir inceliğe değinmiştik. Bir belgenin koşula uyup uymadığını denemek, normalde onu diskteki ya da bellekteki kodlanmış biçiminden, kullanıma hazır bir nesneye **çözmeyi** gerektirir. Eğer milyonlarca belgenin hepsini baştan sona çözüp sonra çoğunu eler atarsanız, çok büyük bir emek boşa gider. Sekizinci bölümde, akıllı bir tasarımın bir belgenin koşula uyup uymadığını onu tümüyle çözmeden kestirmeye çalıştığını söylemiştik; OxiDB'nin **bayt düzeyinde süzme** tekniği, tam olarak budur ve bu bölümün vitrin konusudur.

Fikir şudur. OxiDB, bir belgeyi süzgeçten geçirirken, onu önce nesneye çevirmez; bunun yerine, belgenin kodlanmış baytları üzerinde, doğrudan koşulu kontrol eder. Belgenin yapılandırılmış ikili biçimi — dördüncü ve on altıncı bölümlerde değindiğimiz, JSON'un daha zengin akrabası — bir alanın değerine, tüm belgeyi çözmeden atlamaya olanak tanır. Böylece koşula uymayan belgeler, hiç çözülmeden, yalnızca baytlarına bakılarak elenir. Yalnızca koşula **uyan** belgeler, sonuç için gereken biçime dönüştürülür. Yani sistem, sonunda atacağı belgeler için çözme emeğini hiç harcamaz.

Bu yolun teknik kalbi, ikili biçimin **kısmi çıkarım** (partial extraction) yeteneğidir. JSON'un düz metin biçiminde bir alanın değerine ulaşmak için, metni baştan sona ayrıştırmak gerekir; çünkü alanların nerede başlayıp bittiği ancak okunarak anlaşılır. OxiDB'nin kullandığı ikili biçim ise, bir belgenin hangi alanı nerede tuttuğunu, baytlar üzerinde gezilerek bulunabilir biçimde kodlar. Böylece motor, bir koşulun ilgilendiği **yalnızca o alanın** değerini, belgenin geri kalanına hiç dokunmadan, doğrudan baytlardan okuyabilir. Bir koşul "yaş > 30" diyorsa, motor belgenin baytlarında "yaş" alanına gider, oradaki sayısal değeri çıkarır ve karşılaştırır; belgenin adresi, sipariş geçmişi, iç içe nesneleri — hiçbiri çözülmez. Aranan alan belgede yoksa, bu da baytlardan anlaşılır ve belge yine çözülmeden elenir.

Bu kısmi çıkarımın bir sınırı, OxiDB'nin onu yürütme planına ne zaman yerleştirdiğini de belirler. Bayt düzeyinde süzgeç, ancak süzme yolunun belgeleri sırayla taradığı ve her belgenin baytlarına eriştiği durumlarda

işler — yani indeksin tek başına yetmediği, kalan koşulların bayt üzerinden sınanacağı tarama ya da aday-süzme adımlarında. Eğer bayt yolu bir belge için kesin bir karar veremezse — örneğin koşul, ikili biçimde doğrudan değerlendirilemeyecek kadar karmaşıksa — motor o belge için nesne yoluna geri düşer; ama bu, kuralın değil istisnanın yoludur.

Bunun somut etkisi, OxiDB üzerinde yapılan ölçümlerde çarpıcı biçimde görülür. İndeksin yardım edemediği, çok sayıda belgeyi süzen bir sorgu düşünün — örneğin bir koşula uyan yüz binlerce belge. Eski yaklaşım, eşleşen tüm bu belgeleri — hatta eşleşmeyenleri de denerken — nesnelere çevirip bellekte yüzlerce megabayt tutuyordu. Bayt düzeyinde süzme ise, eşleşmeyenleri hiç çözmeden atladığı için, hem belleği çok daha az kullanıyor hem de daha hızlı çalışıyordu — çünkü çözme, işin en pahalı kısmıdır ve bu yaklaşım onu çoğu belge için tümüyle ortadan kaldırır. Bu, sekizinci bölümün "atacağın şeyi çözme" içgörüsünün gerçek bir sistemdeki, ölçülmüş karşılığıdır.

Aynı bayt düzeyinde fikir, süzmenin ötesine — toplu güncellemelere — de taşındı. Bir güncelleme, önce süzgecine uyan belgeleri bulmak zorundadır; ve OxiDB, bu ilk eşleştirme fazını da, güncellenecek belgeyi baştan sona nesneye çözmeden, doğrudan ham ikili baytlar üzerinde yürütür. Böylece süzgece uymayan belgeler, güncelleme yolunda da hiç çözülmeden elenir; yalnızca gerçekten değişecek belgeler nesneye çevrilir. Bunun ölçülen etkisi çarpıcıdır: çok sayıda belgeyi tek hamlede güncelleyen toplu güncelleme iş yüklerinde, bu ham-bayt birinci-faz eşleştirmesi sayesinde OxiDB, olgun belge veritabanlarını geçebilir hale geldi — çünkü burada da işin en pahalı kısmı, çoğu belge için tümüyle ortadan kalkar.

Bu tekniğin OxiDB'deki gelişim öyküsü de öğreticidir, çünkü doğru çözümün ilk denemede bulunmadığını gösterir. İlk bir deneme, belgeleri yine çözüyor ama bunu farklı bir biçimde yapıyordu; ölçümler bu denemenin aslında **yavaşlattığını** ortaya koydu ve geri alındı. Asıl kazanç, ancak eşleşmeyen belgelerin hiç çözülmediği — yalnızca baytlarına bakılıp atlandığı — doğru tasarımla geldi. Burada, kitap boyunca tekrarlanan bir ders bir kez daha belirir: bir eniyilemenin gerçekten işe yarayıp yaramadığı, ancak ölçülerek bilinir; sezgi, çoğu zaman yanıltıcıdır.

Dürüst bir sınır da belirtmek gerekir. Bu bayt düzeyinde yol, süzülebilir koşullar için geçerlidir. Sonuçların sıralanması, atlanması ya da sınırlanması gerektiğinde, sistem yine de nesne tabanlı yola döner; çünkü sıralama, belgeleri karşılaştırılabilir bir biçimde elde etmeyi gerektirir. Yani bayt düzeyinde süzme, büyük süzme işlerini hızlandırır, ama sıralama gibi işlerin sahibi hâlâ nesne yoludur.

19.7 Tekil işlemlerde erken sonlanma

Sekizinci bölümde, "şu koşula uyan bir kaydı getir" ya da "ilk eşleşeni güncelle" gibi tekil işlemlerin, ilk eşleşmeyi bulduğu an durabileceğini görmüştük. OxiDB bunu doğrudan uygular: tekil okuma, tekil güncelleme ve tekil silme işlemleri, aradıkları ilk belgeyi bulduğu an dururlar; kalan belgeleri taramaya gerek yoktur. Bu, çok sayıda belge arasından tek bir kaydı bulup işlem yapmayı, tüm koleksiyonu gezmeden, hızlı hale getirir.

19.8 MongoDB uyumlu güncelleme anlamları

Sorgu motoru yalnızca okumanın değil, güncellemenin de kalbindedir: bir güncelleme, önce hangi belgelerin değişeceğini bulmak için tam da bu bölümde anlattığımız süzme yollarını kullanır, sonra bulduklarına güncelleme kurallarını uygular. Bu kitap yazılırken OxiDB'nin güncelleme yüzeyi, yaygın belge veritabanı istemcilerinin beklediği anlamlarla — MongoDB'nin kendi uyum testleri OxiDB'ye karşı koşularak — hizalandı; bu hizalama, motora birkaç ince yetenek kazandırdı.

Birincisi, **var-yoksa-ekle** (upsert) anlamıdır: bir güncelleme, süzgecine uyan hiçbir belge bulamazsa, isteğe bağlı olarak süzgeç ile güncellemeden türetilmiş yeni bir belgeyi ekleyebilir. Buna eşlik eden ince ama önemli bir ayrım, **kaç belgenin eşleştiği** ile **kaç belgenin gerçekten değiştiği** sayılarının ayrı ayrı bildirilmesidir; çünkü süzgece uyan bir belge, uygulanan güncelleme onun değerlerini zaten taşıdığı için hiç

değişmemiş olabilir. Bu ayrım, çağıran tarafa "aradığım kayıt var mıydı, zaten güncel miydi, yoksa onu ben mi değiştirdim" sorusunu net biçimde yanıtlar.

İkincisi, **pipeline biçiminde güncellemedir**. Sıradan bir güncelleme, alan alan operatörlerden oluşur; ama OxiDB, güncellemenin bir küçük dönüşüm hattı olarak da verilebilmesine izin verir — belgeyi bir dizi aşamadan (`$set`, `$unset`, `$project` ve kök belgeyi yeniden şekillendiren `$replaceRoot` gibi) geçirerek dönüştürmek. Bu, bir belgenin yeni değerini kendi eski alanlarından hesaplamak gibi, sıradan operatörlerin zor ifade ettiği güncellemeleri doğrudan yazılabilir kılar. Bir dürüstlük notu: `$replaceRoot`, yalnızca bu pipeline-biçimli güncellemenin içinde bir aşama olarak vardır; bir sonraki bölümde göreceğimiz bağımsız toplama pipeline'ında henüz bir aşama değildir.

Üçüncüsü, **dizi süzgeçleridir** (arrayFilters). Bir belgenin içindeki bir dizinin yalnızca belirli koşullara uyan öğelerini güncellemek, klasik bir güçlüktür. OxiDB, konumsal dizi güncellemelerini destekler: bir güncelleme, bir dizinin tüm öğelerine ya da yalnızca adlandırılmış bir süzgece uyan öğelerine — `$[]` ve `$[ident]` biçimleriyle — uygulanabilir. Böylece "şu siparişin, durumu 'bekliyor' olan tüm satırlarını 'iptal' yap" gibi bir güncelleme, diziyi elle gezmeden, tek bir bildirimsel ifadeyle yazılır.

19.9 Sunucu yoluyla ilişki

OxiDB sunucu kipinde çalıştığında, sorgu yürütme ile ağ üzerinden yanıt gönderme birbirine bağlanır. Bir okuma isteğinin yanıtı, mümkün olan her yerde bayt düzeyinde yoldan üretilir: eşleşen belgelerin baytları, çözülüp yeniden kodlanmadan, doğrudan ağ biçimine aktarılır ve istemciye gönderilir. Bayt düzeyinde yolun uygulanamadığı durumlarda — örneğin sıralama gerektiğinde — sistem nesne tabanlı yola geri döner. Sorgu motorunun bu ağ yoluyla nasıl bütünleştiği, sunucu protokolünü ele aldığımız ileriki bölümde daha net olacak; şimdilik akılda tutulacak nokta, bayt düzeyinde süzmenin yalnızca bir iç eniyileme değil, ağ üzerinden gönderilen yanıtın da daha hafif ve hızlı olmasını sağlayan, uçtan uca bir kazanç olduğudur.

19.10 Bildirimsel özgürlüğün yansıması

Sekizinci bölümü, bildirimsel sorgunun tanıdığı özgürlükle kapatmıştık: kullanıcı yalnızca *ne* istediğini söylediği için, sistem *nasıl* sorusunu özgürce yanıtlayabilir. OxiDB'de bu özgürlük somut biçimde görülür. Aynı sorgu, ilgili alanın bir indeksi varsa indeks destekli yoldan, yoksa bayt düzeyinde süzmeli taramadan yürütülür; ve siz sorgunuzu hiç değiştirmeden bir indeks eklediğinizde, OxiDB o indeksi kullanmaya başlar. Kullanıcı "ne", motor "nasıl" sorusuyla ilgilenir; bu ayrım, sekizinci bölümde söylediğimiz gibi, bildirimsel sorgu ile akıllı yürütmenin aynı madalyonun iki yüzü olduğunu bir kez daha gösterir.

19.11 Bu bölümün bıraktığı yer

Bu bölümde, OxiDB'nin sorgu motorunu yakın plana aldık. Sorgunun zengin operatör dağarcığını ve nasıl bir koşul ağacına ayrıştırıldığını; koşulların indeksli ve süzgeç sınıflarına — operatör operatör — ayrılmasını ve "indeks daraltır, süzgeç inceltir" iş bölümünün bu ayrım üzerine kurulduğunu; bir VE sorgusunda en seçici koşulu kardinalite tahminiyle sürücü seçip ötekileri kesişim süzgecine çeviren seçicilik temelli sıralamayı; indeks destekli sayım ve sıralamanın belgelere hiç dokunmadan yanıt veren iki hızlı yolunu; ve OxiDB'nin vitrin tekniği olan bayt düzeyinde süzmeyi — ikili biçimin kısmi çıkarımıyla eşleşmeyen belgeleri hiç çözmeden eleyerek hem belleği hem hızı iyileştiren yaklaşımı — gördük. Tekil işlemlerdeki erken sonlanmayı; sorgu motorunun güncellemenin de kalbinde oluşunu ve MongoDB uyumlu güncelleme anlamlarını — var-yoksa-ekle, pipeline biçiminde güncelleme ve dizi süzgeçleri; sunucu yoluyla bütünleşmeyi ve bildirimsel özgürlüğün bu motordaki yansımasını izledik.

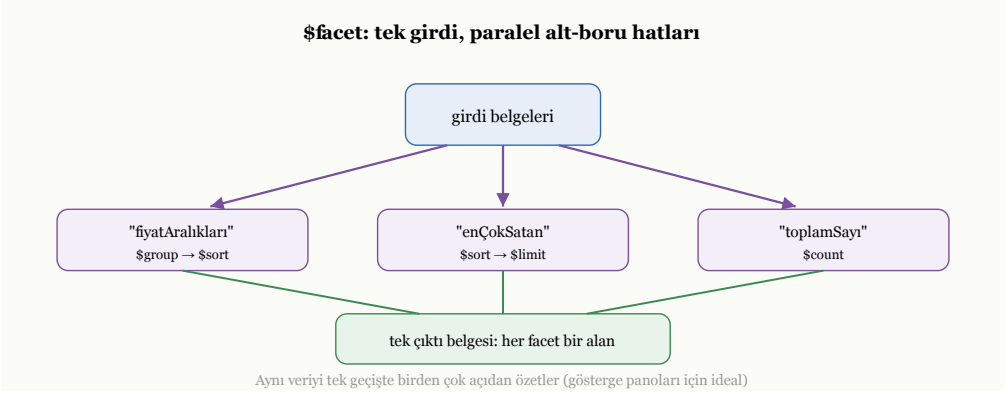
Buraya kadar hep tek tek belgeleri bulup süzmekle ilgilendik. Ama dokuzuncu bölümde gördüğümüz gibi, veriye sorabileceğimiz daha zengin bir soru türü daha vardır: belgeleri gruplayan, özetleyen ve dönüştüren

toplama. Bir sonraki bölümde, OxiDB'nin toplama pipeline'ını — gruplamayı, çok-yönlü analizi ve pencere fonksiyonlarını, ki bunların bazılarını bu kitap yazılırken motora ekledik — dokuzuncu bölümdeki ilkelerle bağlayarak ele alacağız.

Bölüm 20

OxiDB'nin Toplama Pipeline'ı: Gruplama, $facet ve Pencere Fonksiyonları

Önceki bölümlerde, OxiDB'nin tek tek belgeleri nasıl bulup süzdüğünü gördük. Ama dokuzuncu bölümde öğrendiğimiz gibi, veriye sorabileceğimiz daha zengin bir soru türü vardır: belgeleri gruplayan, özetleyen ve dönüştüren toplama. Bu bölüm, OxiDB'nin toplama pipeline'ını dokuzuncu bölümdeki ilkelerle bağlayarak ele alıyor. Bu bölümün ayrı bir özelliği var: anlatacağımız yeteneklerin bir kısmı — çok-yönlü analiz ve pencere fonksiyonları — tam da bu kitap yazılırken OxiDB'ye eklendi; bu yüzden onları yalnızca bir kullanıcı olarak değil, gelişim öyküleriyle birlikte anlatabiliyoruz.



Şekil 20.1: $facet: tek girdi üzerinde paralel alt-boru hatları.

20.1 Pipeline ve aşamaları

Dokuzuncu bölümde toplamayı bir montaj hattına benzetmiştik: belgeler bir uçtan akar, bir dizi aşamadan geçer, her aşama akışı dönüştürüp bir sonrakine verir, diğer uçtan nihai sonuç çıkar. OxiDB'nin toplama motoru tam olarak böyle çalışır ve dokuzuncu bölümde tanıdığımız aşamaların hepsini sunar: süzme, gruplama, sıralama, atlama, sınırlama, yeniden şekillendirme, sayma, listeyi belgelere açma, başka koleksiyondan ilişkili veri getirme, çok-yönlü analiz ve pencere fonksiyonları. Bu aşamalar arka arkaya dizilerek, basit yapı taşlarından karmaşık analizler kurulur — dokuzuncu bölümdeki bileşilebilirlik ilkesinin somut karşılığıdır bu.

20.2 Baştaki süzmeyi öne çekmek

Dokuzuncu bölümde, en temel toplama eniyilemesinin süzmeyi olabildiğince erkene almak olduğunu söylemiştik; çünkü akışı erkenden daraltmak, sonraki her aşamanın yükünü azaltır. OxiDB bunu somut biçimde yapar: bir pipeline'ın başında bir süzme aşaması varsa, OxiDB onu pipeline'ın geri kalanından **ayırıp**, belgeleri depodan okurken doğrudan uygular. Üstelik bu baştaki süzme, on dokuzuncu bölümde gördüğümüz indeks destekli yollardan yararlanabilir; yani pipeline'ın girdisi, tüm koleksiyon taranmadan, indeksle daraltılmış bir aday kümesi olabilir. Böylece toplama, daha işin başında, üzerinde çalışacağı veri miktarını en aza indirir.

20.3 Akış halinde gruplama

Dokuzuncu bölümde gruplamanın toplamanın kalbi olduğunu ve doğası gereği çok sayıda belgeye dokunduğunu söylemiştik. OxiDB, bu maliyeti yönetmek için gruplamayı **akış halinde** yapar: tüm belgeleri belleğe yığıp sonra gruplamak yerine, belgeleri depodan tek tek akıtarak, her birinin grup anahtarına ve biriktiricilere katkısını anında işler. Böylece dev bir koleksiyon, bütünüyle belleğe alınmadan gruplanabilir.

Burada, on dokuzuncu bölümdeki bayt düzeyinde fikrin gruplamaya uygulanmış bir biçimini görürüz. Bir grup hesabı, çoğu zaman belgenin yalnızca birkaç alanına ihtiyaç duyar: grup anahtarına ve toplanan ya da ortalaması alınan alana. OxiDB, gruplama sırasında belgeyi bütünüyle çözmek yerine, kodlanmış baytlarından **yalnızca gereken alanları** çıkarır; gerisini hiç dokunmadan geçer. Bu, on dokuzuncu bölümdeki "atacağın şeyi çözme" ilkesinin gruplamadaki yankısıdır: gruplama için gereksiz olan alanlar, hiç çözülmez. Biriktiriciler — sayma, toplama, ortalama, en büyük, en küçük — bu akış üzerinde, her grup için bir özet biriktirerek çalışır.

Dokuzuncu bölümün bir kestirmesi de OxiDB'de doğrudan karşımıza çıkar. Eğer gruplama yalnızca **saymaysa** ve grup anahtarının bir indeksi varsa, OxiDB belgelere hiç dokunmaz; on sekizinci bölümde gördüğümüz gibi, indeksten doğrudan her grubun belge sayısını okur. Bu, dokuzuncu bölümde değindiğimiz "indeksli sayma gruplaması, belgelere dokunmadan yanıtlanabilir" fikrinin somut karşılığıdır ve ölçümlerde belirgin bir hız kazancı sağlar.

Bu kestirme, yalnızca saymayla da sınırlı kalmadı. Bu kitap yazılırken, indeksin kapsadığı (covered) gruplama daha ileri götürüldü: eğer bir bileşik indeks hem grup anahtarını hem de toplanan alanı birlikte tutuyorsa, yalnızca sayma değil, toplama, ortalama, en büyük ve en küçük gibi biriktiriciler de tek bir belge çözülmeden, salt indeks yürüyüşüyle hesaplanabilir. On sekizinci bölümde değindiğimiz bu indeks-yalnız yol, tüm koleksiyonu kapsayan bir gruplamayı, gereken bütün değerler zaten indekste durduğu için belgelere hiç inmeden yanıtlar — disk-öncelikli kipte, belge gövdelerini bellekten hiç okumamak demek olan bu kazanç, bu bölümün sonunda döneceğimiz performans öyküsünün de bir parçasıdır.

20.4 Çok-yönlü analiz: $facet

Dokuzuncu bölümde çok-yönlü analizi, aynı süzülmüş veri kümesinden birden çok özeti tek bir geçişte üretmek olarak tanımlamıştık — bir alışveriş sitesinin sonuç sayfasındaki, kategoriye göre sayılar, fiyat dağılımı ve en iyi birkaç ürünün hep birlikte istenmesi gibi. OxiDB'nin bu yeteneği, tam da bu kitap yazılırken eklendi ve fikri dokuzuncu bölümdekiyle birebir aynıdır: aynı girdi üzerinde birkaç alt-pipeline'ı çalıştırıp her birinin sonucunu ayrı bir alanda toplamak.

Uygulaması zarif biçimde sadedir. OxiDB, bu aşamaya gelen belgeleri bir kez hazırlar ve her alt-pipeline'ı, bu aynı girdi üzerinde, var olan pipeline motorunu yeniden kullanarak çalıştırır. Sonuç, tek bir belgedir; o belgenin her alanı, bir alt-pipeline'ın sonuç listesidir. Bu sayede, üç ayrı sorgu yapıp veriyi üç kez taramak yerine, tek bir geçişte üç özet birden üretilir. Sadeliğin bir gereği olarak, bir alt-pipeline'ın içine yine bir çok-yönlü analiz koymak ya da oraya bir yan etki — örneğin başka bir koleksiyona yazma — sığdırmak engellenir; çünkü bunlar, çok-yönlü analizin "aynı veriyi birden çok açıdan özetle" amacına aykırıdır.

Bu sadeliğin altında öğretici bir uygulama ayrıntısı yatar. Çok-yönlü analiz, her alt-pipeline'ı yürütürken, ona aynı girdi belgelerinin **kendi kopyasını** verir; çünkü her alt-pipeline veriyi kendince dönüştürecek — biri sayıp gruplayacak, biri sıralayıp ilk birkaçını alacak — ve bu dönüşümler birbirine karışmamalıdır. OxiDB küçük ama hoş bir tasarrufla, son alt-pipeline'a kopya yerine asıl girdiyi **taşıyarak** (move) verir; böylece N alt-pipeline için yalnızca N−1 kopya yapılır, sonuncusu için gereksiz bir kopya hiç çıkarılmaz. Daha temel olansa, çok-yönlü analizin **ayrı bir motor gibi davranmayıp**, kitabın bu bölümünde anlattığımız pipeline yürütücüsünü olduğu gibi yeniden çağırmasıdır: her alt-pipeline, sıradan bir pipeline'ın "şu aşamadan itibaren şu belgeler üzerinde çalış" çağrısıyla yürütülür. Yani çok-yönlü analiz, yeni bir yetenek eklemekten çok, var olan yürütücüyü kendi üzerine katlayan bir bileşendir — dokuzuncu bölümdeki bileşilebilirlik ilkesinin, motorun kendi iç tasarımına yansımış halidir bu.

20.5 Pencere fonksiyonları

Dokuzuncu bölümde, gruplamanın yanında ikinci büyük analitik biçimi — satırları çökertmeden, her satıra komşularından türeyen bir değer ekleyen pencere fonksiyonlarını — tanımıştık. Kümülatif toplam, hareketli ortalama, sıralama, bir önceki satırın değeri gibi hesaplar bu aileye girer. OxiDB'nin bu yeteneği de bu kitap yazılırken eklendi ve dokuzuncu bölümdeki üç adımlı modeli doğrudan izler.

OxiDB, pencere fonksiyonunu uygularken önce belgeleri bir alana göre **bölümlere** ayırır — örneğin her bölgeyi ayrı bir bölüm yapar. Sonra her bölümü, belirtilen sıralama alanına göre düzenler — örneğin tarihe göre. En sonra, her belge için, o belgenin çevresindeki bir pencereye bakarak istenen değeri hesaplar ve onu belgeye yeni bir alan olarak ekler — ama belgeyi silmeden, her satırı koruyarak. Pencere üzerindeki biriktiriciler — toplam, ortalama ve benzerleri — kümülatif toplamı ya da hareketli ortalamayı üretir; sıralama işlevleri, satırın bölüm içindeki yerini verir; kaydırma işlevi ise bir önceki ya da sonraki satırın değerini getirir. Dokuzuncu bölümde söylediğimiz gibi, gruplama "her grup için tek özet ver", pencere fonksiyonu ise "her satırı koru, ona komşularından bir değer ekle" der; OxiDB bu ikisini ayrı ama bütünleyici araçlar olarak sunar.

Şekil 20.2: Bölümle, kararlı sırala, çerçeve üzerinde biriktir.

Bu üç adımın her birinde, dikkat edilmesi gereken mühendislik incelikleri vardır. **Sıralamanın kararlı (stable) olması** kritiktir: aynı sıralama anahtarına sahip iki belge, sıralamadan önceki göreli düzenlerini korumalıdır. Bu, sıralama işlevleri ve özellikle kaydırma için belirleyicidir — "bir önceki satır" ancak "önceki"nin iyi tanımlı olmasıyla anlam taşır; kararsız bir sıralama, eşit anahtarlı satırlarda "öncesi"ni belirsiz bırakırdı.

Asıl ayrıntı ise **çerçevededir** (window frame). Her satır için "çevresindeki pencere" dediğimiz şey, en yalın biçimde belge sayısıyla tanımlanan bir çerçevedir: örneğin "kendisi ve önceki iki belge" ya da "bölümün başından bu satıra kadar". Çerçeve belirtilmezse varsayılan, bölümün tamamıdır. Motor, her satır için bu

çerçevenin kapsadığı belgeleri belirler ve biriktiriciyi yalnızca o aralık üzerinde çalıştırır. "Bölümün başından bu satıra kadar" çerçevesiyle bir toplam, kümülatif toplamı; "kendisi ve önceki k belge" çerçevesiyle bir ortalama, k+1 genişliğinde bir hareketli ortalamayı verir. Sıralama ve kaydırma işlevleri ise çerçeveye değil, satırın bölüm içindeki sıralı konumuna bakar.

Bu belge-tabanlı çerçeve, uzun süre OxiDB'nin tek çerçeve türüydü; "son yedi günlük pencere" ya da "değeri şu aralıkta olan satırlar" gibi, çerçeveyi belge sayısına değil sıralama değerinin **kendisine** — bir zamana ya da bir büyüklüğe — göre tanımlayan **aralık tabanlı çerçeveler** eksikti. Bu kitap yazılırken bu boşluk da kapatıldı ve pencere çerçevesi artık iki biçimde verilebiliyor. Belge çerçevesi, satırın konumuna göre sayar — "kendisi ve önceki iki belge" gibi. **Aralık çerçevesi** ise sıralama alanının değerine göre bir alt ve üst sınır alır; isteğe bağlı bir zaman birimiyle, sıralama alanı bir tarih olduğunda "son yedi gün" gibi zaman pencerelerini de ifade eder. Böylece kümülatif toplam ve sabit-genişlikli hareketli ortalamadan, değer ve zaman ekseninde kayan pencerelere kadar pencere fonksiyonlarının en sık kullanılan biçimleri bugün eksiksiz karşılanır. Dürüst bir küçük sınır kalır: aralık çerçevesinin zaman birimleri sabit uzunlukludur — gün, saat, dakika gibi — ; ay, çeyrek, yıl gibi uzunluğu takvime göre değişen birimler, bu bölümün ileride değineceğimiz zaman-serisi aşamalarında olduğu gibi, burada da kapsam dışıdır.

20.6 Zaman-serisi aşamaları ve boşluksuz zincir

OxiDB'nin toplama dağarcığı, gruplama ve pencere fonksiyonlarının yanında, bir de zamanla akan veriye — ölçümlere, tiklere, günlük kayıtlarına — özel bir aşama ailesi taşır; bunların çoğu da bu kitap yazılırken eklendi. Bu aşamalar, dokuzuncu bölümdeki toplama ilkelerinin zaman eksenine uygulanmış halidir.

En temeli, **zaman kovalarına bölmedir**: belgeleri, bir zaman alanına göre eşit aralıklı kovalara (örneğin saatlik ya da günlük) toplayıp her kova için bir özet üretmek — üstelik hiç belge düşmeyen boş kovaları da sıfırla doldurarak, sonuç dizisinde delik bırakmadan. Bunun bir üstünde, borsa verisinin klasik biçimi olan **mum çubuğu** özeti durur: ham tik ya da işlem belgelerini, her zaman aralığı için açılış–en yüksek–en düşük–kapanış–hacim beşlisine çökerten aşama; birden çok sembolü ayrı bölümler olarak ele alır ve veri olmayan aralıklarda son bilinen değeri taşıyarak boşlukları örter.

Bu aşamaların asıl gücü, **boşluksuz bir zincir** halinde birleşmelerinde ortaya çıkar. Gerçek zaman-serisi verisi seyrektir: bazı zaman noktalarında hiç ölçüm yoktur. OxiDB, bu boşlukları iki adımda kapatır. Önce, bir **yoğunlaştırma** aşaması, sayısal ya da tarih ekseninde eksik olan zaman noktaları için yeni belgeler üretir — ekseni sabit adımlarla doldurarak. Sonra, bir **doldurma** aşaması, bu yeni üretilmiş boş noktaların değerlerini anlamlı biçimde tamamlar: bir önceki bilinen değeri taşıyarak, iki bilinen değer arasında doğrusal ara değer hesaplayarak ya da sabit bir değer koyarak. Böylece mum çubuğundan yoğunlaştırmaya, oradan doldurmaya uzanan bir zincir — önce özetle, sonra ekseni doldur, en sonra değerleri tamamla — seyrek, delikli bir zaman serisini, hiç boşluğu olmayan düzgün bir seriye dönüştürür. Burada da dürüst bir sınır vardır: bu eksen adımları sabit uzunluklu zaman birimleriyle (gün, saat gibi) tanımlanır; ay ve üzeri, uzunluğu takvime göre değişen birimler henüz desteklenmez.

20.7 Dürüst bir envanter: henüz olmayan aşamalar

Dokuzuncu bölümde toplamanın zenginliğini överken, bir kitabın görevinin yalnızca var olanı anlatmak değil, sınırları da dürüstçe çizmek olduğunu söylemiştik. OxiDB'nin toplama dağarcığı geniştir; ama olgun, yıllar içinde büyümüş belge veritabanlarının sunduğu her aşamayı henüz kapsamaz. Otomatik aralıklara bölme, çizge biçiminde özyinelemeli ilişki gezme, iki koleksiyonun sonuçlarını birleştirme, kök belgeyi bir alt-belgeyle değiştirme, sonucu doğrudan bir koleksiyona yazma ve rastgele örnekleme gibi aşamalar, bu sürümün toplama motorunda henüz yoktur. Bunları saymak, OxiDB'yi küçümsemek değil; bir mühendisin bir aracı seçerken neyi sayıp neyi sayamayacağını bilmesini sağlamaktır. Var olan aşamalar — süzme, gruplama, sıralama, yeniden şekillendirme, listeyi açma, ilişkili veri getirme, çok-yönlü analiz ve pencere fonksiyonları — analitik sorguların büyük çoğunluğunu karşılar; eksik olanlar, çoğunlukla bu yapı taşlarının dışarıdan birkaç adımla kurulabildiği, daha özel ihtiyaçlardır.

20.8 Toplamanın performans gerçeği: disk-öncelikli durum

Dokuzuncu bölümde toplamanın, tekil aramalardan farklı bir performans karakteri taşıdığını ve doğası gereği çok sayıda, kimi zaman tüm belgelere dokunduğunu söylemiştik. OxiDB'de bu gerçek, on altıncı bölümdeki depolama kipleriyle ilginç bir biçimde etkileşir ve burada dürüstçe ele alınmayı hak eder.

Belleğe öncelikli kipte, gruplamanın akış halinde okuduğu her belge zaten bellektedir; bu yüzden toplama hızlıdır. Disk-öncelikli kipte ise her belge, on altıncı bölümde gördüğümüz gibi, belleğe yansıtılmış `.bdat` dosyasından okunur. Toplama tüm belgelere dokunduğu için, bu okuma maliyeti her belge için tekrarlanır. Eğer `.bdat` sıkıştırılmışsa, her belgenin okunması bir de açma — yani işlemci işi — gerektirir; ve toplama tüm koleksiyonu gezdiği için, bu açma maliyeti birikir. OxiDB üzerinde yapılan ölçümler bunu net biçimde gösterdi: disk-öncelikli, sıkıştırılmış kipte tüm koleksiyonu tarayan bir gruplama, belleğe öncelikli kipe kıyasla belirgin biçimde — milyon belgelik bir veride kat kat — yavaşlıyordu.

Bu yavaşlığın iki ayrı kökü vardı ve ikisi de bu kitabın yazımı sırasında giderildi. Birincisi, sayma gruplamalarında indeksten yararlanan o hızlı yolun disk-öncelikli indekslerde devre dışı kalmasıydı; bunu on sekizinci bölümde anlattık ve yeniden etkinleştirilmesi, sayma gruplamalarını taramadan kurtardı. İkincisi, sıkıştırmanın getirdiği açma maliyetiydi; on altıncı bölümde değindiğimiz sıkıştırmasız saklama seçeneği, her belgeyi açma zorunluluğunu ortadan kaldırarak — ve sıfır-kopya erişimi mümkün kılarak — tüm koleksiyonu tarayan gruplamaları büyük ölçüde hızlandırdı. Böylece disk-öncelikli kip, sıkıştırmasız ve indeks destekli haliyle, toplama performansında belleğe öncelikli kibe çok yaklaştı. Bu öykü, dokuzuncu, on üçüncü ve on altıncı bölümlerdeki ilkelerin — toplamanın çok şeye dokunması, sıkıştırmanın işlemci bedeli ve indeksten yararlanma — gerçek bir sistemde nasıl iç içe geçtiğinin güzel bir örneğidir.

20.9 Bu bölümün bıraktığı yer

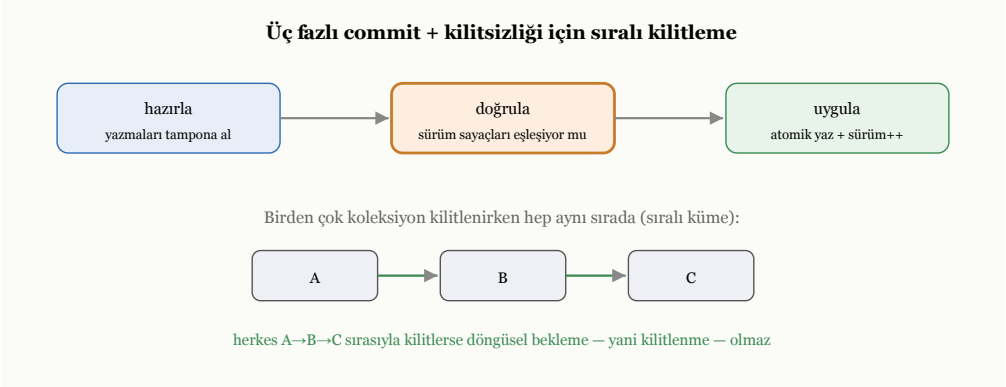
Bu bölümde, OxiDB'nin toplama pipeline'ını yakın plana aldık. Aşamalarının dokuzuncu bölümdeki montaj hattını nasıl somutlaştırdığını; baştaki süzmenin öne çekilip indeksle daraltıldığını; gruplamanın akış halinde, yalnızca gereken alanları çözerek çalıştığını; sayma gruplamalarının indeksten doğrudan yanıtlandığını; çok-yönlü analizin aynı girdiyi birkaç açıdan tek geçişte özetlediğini; ve pencere fonksiyonlarının üç adımla — bölümle, kararlı sırala, belge ya da aralık tabanlı çerçeve üzerinde biriktir — satırları koruyarak komşulardan değer türettiğini gördük. Zamanla akan veriye özel aşamaların — zaman kovalarına bölme, mum çubuğu özeti ve seyrek bir seriyi düzgünleştiren özetle-doldur-tamamla zincirinin — nasıl boşluksuz bir zaman serisi kurduğunu da izledik. Çok-yönlü analizin pipeline yürütücüsünü kendi üzerine katlayan zarif uygulamasını, pencere çerçevesinin belge ve aralık tabanlı iki biçimini ve henüz olmayan aşamaların dürüst envanterini izledik. Ayrıca disk-öncelikli kipte toplamanın performans gerçeğini ve onu iyileştiren iki düzeltmeyi — bu kitap yazılırken yapılan iki işi — dürüstçe gördük.

Buraya kadar, Kısım III boyunca hep okuma tarafıyla — saklama, bulma, süzme, özetleme — ilgilendik. Ama onuncu bölümde gördüğümüz gibi, bir veritabanının asıl zorlu sınavı, eşzamanlı yazmalar ve "ya hep ya hiç" güvencesidir. Bir sonraki bölümde, OxiDB'nin işlem mekanizmasını — onuncu bölümde tanıdığımız iyimser eşzamanlılık denetimini ve onun üç fazlı tamamlama düzenini — somut olarak ele alacağız.

Bölüm 21

OxiDB'de İşlemler: İyimser Eşzamanlılık ve Üç Fazlı Commit

Kısım III boyunca, buraya kadar hep okuma tarafıyla ilgilendik: OxiDB'nin veriyi nasıl sakladığını, dayanıklı kıldığını, indekslediğini, sorguladığını ve özetlediğini gördük. Ama onuncu bölümde öğrendiğimiz gibi, bir veritabanının asıl zorlu sınavı, eşzamanlı yazmalar ve "ya hep ya hiç" güvencesidir. Bu bölüm, OxiDB'nin işlem mekanizmasını — onuncu bölümde tanıdığımız iyimser eşzamanlılık denetimini ve onun üç fazlı tamamlama düzenini — somut olarak ele alıyor.



Şekil 21.1: Üç fazlı commit ve kilitlenmeyi önleyen sıralı kilitleme.

21.1 OxiDB neden iyimser yolu seçti

Onuncu bölümde, yalıtımı sağlamanın üç felsefesini görmüştük: kötümser kilitleme, çok sürümlü MVCC ve iyimser OCC. OxiDB, bunlardan **iyimser eşzamanlılık denetimini** seçer. Bu seçimin altında, onuncu bölümde tanımladığımız iyimser varsayım yatar: çoğu zaman çatışmalar nadirdir; iki işlem aynı belgeye aynı anda dokunmaz. Madem öyle, baştan kilitleyip herkesi bekletmek yerine, işlemleri serbestçe çalıştırıp çatışmayı yalnızca tamamlama anında denetlemek daha verimlidir.

Onuncu bölümdeki mağaza-kasası benzetmesini hatırlayalım: ürünleri sepete koyarken kimseye sormazsınız, kasada almak istediğinizin hâlâ uygun olup olmadığı kontrol edilir, bir sorun çıkarsa o turu baştan yaparsınız. OxiDB'nin işlemleri tam olarak böyle davranır. Bu yaklaşım, çatışmaların nadir olduğu tipik belge iş yüklerinde — ki belgeler çoğu zaman bağımsız bütünler olduğu için çatışmalar gerçekten nadirdir — kimseyi boşuna bekletmediği için hızlıdır.

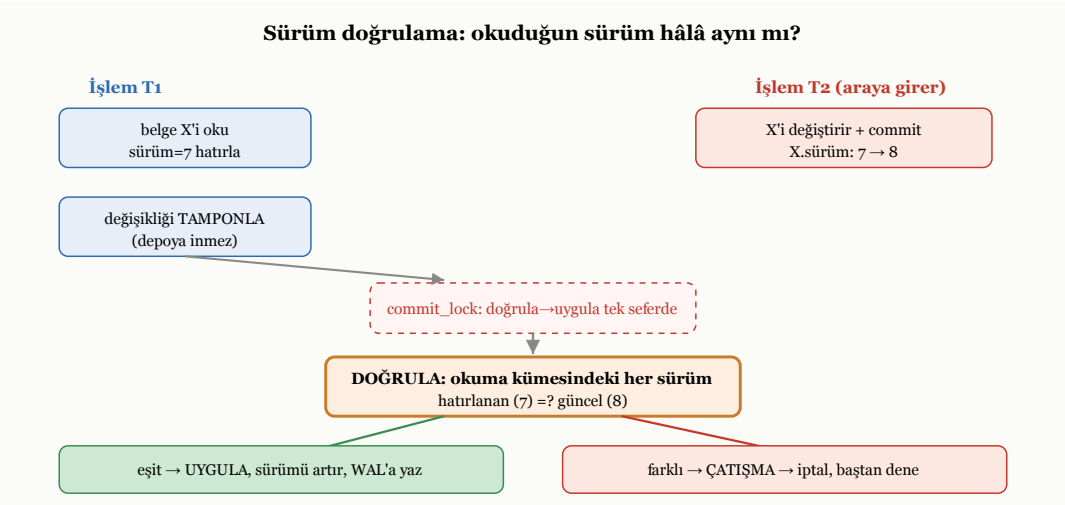
21.2 İyimser akışın üç fazı

OxiDB'nin işlemleri, onuncu bölümdeki iyimser akışı **üç fazlı bir tamamlama** düzeniyle hayata geçirir. Bu üç fazı tek tek görelim, çünkü OCC'nin somut işleyişi tam olarak buradadır.

İşlem çalışırken, yaptığı değişiklikleri hemen asıl depoya uygulamaz; onları bir kenarda **biriktirir**. Yani işlem boyunca, dışarıdan bakan hiç kimse bu yarım değişiklikleri görmez; depo, işlem tamamlanana dek dokunulmamış gibi durur. Bu, onuncu bölümde OCC'nin "değişiklikleri biriktir" adımının karşılığıdır. İşlem, aynı zamanda dokunduğu her belgenin **sürüm numarasını** hatırlar; çünkü OxiDB, her belgeye, her değiştiğinde artan bir sürüm sayacı iliştirir.

İşlem "tamamla" dediğinde, üç faz devreye girer. Birinci faz, **hazırlıktır**: biriktirilen tüm değişiklikler bir araya getirilir. İkinci faz, **doğrulamadır**: işlemin dokunduğu her belgenin sürüm numarasının, işlem onu okuduğundan bu yana değişip değişmediği kontrol edilir. Eğer tüm sürümler hâlâ aynıysa — ki iyimser varsayıma göre çoğu zaman böyledir — hiçbir çatışma yok demektir ve üçüncü faza geçilir. Üçüncü faz, **tamamlamadır**: biriktirilen değişiklikler asıl depoya uygulanır, ilgili belgelerin sürüm numaraları artırılır ve değişiklikler, bir önceki bölümlerde gördüğümüz yazma-öncesi günlüğe yazılarak dayanıklı kılınır.

Ya doğrulama fazı başarısız olursa? Yani işlem çalışırken, dokunduğu bir belgeyi başka biri değiştirmiş ve onun sürüm numarasını artırmışsa? O zaman bir **çatışma** saptanmış demektir; işlem iptal edilir ve değişiklikleri uygulanmaz. Çağıran taraf, işlemi baştan deneyebilir. Bu, onuncu bölümde anlattığımız "çatışmada iptal et ve yeniden dene" davranışının tam karşılığıdır; sürüm numaraları, çatışmayı saptamanın aracıdır.



Şekil 21.2: Okuduğun sürüm hâlâ aynı mı? Eşitse uygula, farklıysa çatışma.

21.3 Doğrulama ile uygulamanın bölünmezliği

Burada, OCC'nin kâğıt üzerinde basit görünen ama gerçek bir uygulamada en kolay yanlış yapılan yerine gelmemiz gerekir: doğrulama ile uygulamanın **kendi arasında atomik** olması. Düşünün ki iki işlem aynı belgeye dokunuyor ve her ikisi de neredeyse aynı anda tamamlanmaya çalışıyor. Birinci işlem belgenin sürümünü okuyup "hâlâ aynı, çatışma yok" diyor; tam uygulamaya geçecekken, ikinci işlem de aynı sürümü okuyup aynı kararı veriyor. Sonra ikisi de yazıyor. İkisi de doğrulamayı geçti, ama biri diğerinin yazdığının üzerine yazdı — bu, onuncu bölümde "kayıp güncelleme" (lost update) dediğimiz tam da o çatışmadır ve OCC'nin önlemek için var olduğu şeydir. Eğer doğrulama ve uygulama arasında bir boşluk kalırsa, OCC kendi amacını ıskalar.

OxiDB bunu, tüm tamamlamaları seri hale getiren tek bir **tamamlama kilidiyle** (commit lock) çözer. Bir işlem tamamlanmaya başladığında bu kilidi alır ve onu doğrulamanın **ve** uygulamanın sonuna dek tutar; ancak işin bütünü bitince bırakır. Böylece "sürümleri kontrol et, değişiklikleri uygula, sürümleri artır" üçlüsü, başka hiçbir tamamlamanın araya giremeyeceği, bölünmez bir kritik kesit (critical section) haline gelir. İki işlem aynı belgeye yarışsa bile, biri kilidi tutarken öteki bekler; bekleyen işlem sırası geldiğinde sürümü yeniden okur, bu kez artmış bulur ve usulünce çatışma verir. Bu, iyimser bir sistemde bile, tamamlama anının neden küçük bir seri kesit gerektirdiğini gösteren güzel bir örnektir: işlemler boyunca kimse beklemez — yalnızca son, kısacık tamamlama adımı seriye alınır.

21.4 Dayanıklılıkla bağ

İşlemler, on yedinci bölümdeki dayanıklılık mekanizmasıyla doğrudan bütünleşir. Bir işlem tamamlandığında, biriktirilmiş değişiklikler tek seferde yazma-öncesi günlüğe yazılır; on yedinci bölümde, her günlük kaydının bir işlem kimliği taşıdığını söylemiştik — işte o kimlik, bir kaydın hangi işleme ait olduğunu belirtir ve kurtarmada işlemleri bir bütün olarak ele almayı sağlar. İşlemin atomikliği — ya hep ya hiç — iki şeyden gelir: değişikliklerin yalnızca tamamlama anında, hep birlikte uygulanması ve bunların günlükle dayanıklı kılınması. Bir çökme olursa, kurtarma, tamamlanmış işlemlerin değişikliklerini bir bütün olarak geri getirir; yarım kalmış, hiç tamamlanmamış bir işlemin biriktirilmiş ama uygulanmamış değişiklikleri ise zaten depoya hiç inmediği için kaybolur — ki bu da istenen davranıştır.

Bu mekanizmanın iki ince ayrıntısı, tamamlamanın gerçekte nasıl işlediğini aydınlatır. Birincisi, üçüncü fazda değişiklikler doğrudan depoya yazılmaz; önce **hazırlanmış mutasyonlar** (prepared mutations) olarak somutlaştırılırlar. Yani işlemin "şu sorguya uyan belgeyi güncelle" gibi yüksek seviyeli niyeti, tamamlama anında, hangi belgenin hangi yeni baytlarla yazılacağına dair somut bir işlemler listesine çevrilir; bu liste hem yazma-öncesi günlüğe yazılan kayıtları hem de asıl depo değişikliklerini içerir. Böylece günlüğe yazılan ile depoya uygulanan şey, birbirinin tıpatıp karşılığı olur — kurtarmada birinin ötekini eksiksiz yeniden üretebilmesi tam da buna dayanır.

İkincisi, işlemin "tamamlandı" sayıldığı kesin an — yani **tamamlama noktası** (commit point) — belgeler asıl depoya uygulanmadan **önce**, işlemin kimliğinin küresel bir tamamlama günlüğüne (commit log) dayanıklı biçimde işlendiği andır. Bu sıralama bilinçlidir ve kurtarmanın anlamını belirler. Bir işlemin kayıtları yazma-öncesi günlüğe inmiş ama tamamlama noktası daha aşılmamışken sistem çökerse, kurtarma o işlemi tamamlanmamış sayar ve değişikliklerini geri getirmez — çünkü kullanıcıya "tamamlandı" yanıtı hiç verilmemiştir. Tamamlama noktası aşıldıktan sonra çökme olursa, kurtarma işlemi tamamlanmış sayar ve günlükteki kayıtlarından eksiksiz yeniden uygular. Bu küresel tamamlama günlüğü, ayrıca aynı anda tamamlanan birçok işlemi tek bir disk eşitlemesinde (fsync) toplayan bir **grup tamamlama** (group commit) düzeniyle çalışır: N eşzamanlı tamamlama, N ayrı diske-yazma yerine tek bir paylaşılan eşitlemeyle dayanıklı kılınır; bu, on yedinci bölümde değindiğimiz, dayanıklılığın en pahalı adımını — fsync'i — amorti etme fikrinin işlem katmanındaki yankısıdır.

21.5 Okuma anlık görüntüleri: yazarı bekletmeden tutarlı okuma

Buraya kadar anlattığımız her şey **yazma** yoluyla ilgiliydi: değişiklikleri biriktirmek, sürümleri doğrulamak, tamamlama kilidini tutmak. Ama onuncu bölümde gördüğümüz gibi, eşzamanlılığın bir de okuma yüzü vardır. Uzun süren bir okuma — diyelim ki tüm hesapların bakiyesini toplayan bir analiz — sürerken, arada bir para transferi tamamlanırsa ne olur? Naif bir okuma, transferin bir yarısını (borcun düşüldüğü hesabı) görüp öteki yarısını (alacağın eklendiği hesabı) kaçırabilir ve tutarsız bir toplam üretir. İşte OxiDB'ye bu kitap yazılırken eklenen **okuma anlık görüntüleri** (read snapshots), tam olarak bunu önler.

Bu mekanizmanın en zarif yanı, **yalnızca okuma yolunu** değiştirmesidir; az önce anlattığımız yazma yolu — iyimser denetim, sürüm doğrulama, grup tamamlama — hiç dokunulmadan kalır. Fikir, onuncu bölümdeki

çok sürümlü denetimin (MVCC) hafif bir biçimidir: bir okuma, başladığı andaki durumu görür ve o okuma sürerken tamamlanan yazmalar, onun gördüğü resmi değiştirmez.

Bunun en görünür meyvesi, bir önceki bölümdeki toplama pipeline'ındadır. OxiDB'de bir toplama, **varsayılan olarak anlık-görüntü tutarlıdır**: bir para transferinin yarısını asla göremez. Uygulaması iyimserlik ruhuna sadıktır. Toplama önce en son durumu iyimserce okur; eğer o koşu sırasında hiçbir yazma araya girmemişse, sonuç doğrudan geçerli sayılır ve fazladan hiçbir iş yapılmaz. Yalnızca bir yazma gerçekten yarışmışsa, motor her zaman doğru olan bir yedeğe düşer — durumu çözüp yeniden kontrol eder. Böylece yaygın durumda (yarış yok) hiçbir maliyet ödenmez, nadir durumda (yarış var) doğruluk yine de garanti altına alınır.

Bu örtük tutarlılığın yanında, açık bir anlık görüntü arayüzü de vardır. Bir okuyucu bir anlık görüntü **başlatabilir**, onun üzerinde birden çok bulma, sayma ve toplama yapabilir — hepsi aynı donmuş ana bakarak — ve işi bitince onu **kapatabilir**. Bu, birden çok okumanın tutarlı tek bir görüntüden yapılmasını gerektiren raporlar için biçilmiş kaftandır. Yalnızca-okuma bir yetenek olduğu için, en düşük yetki katmanındaki — salt-okuma rolündeki — bir kullanıcıya bile açıktır; ve anlık görüntünün kimliği, oturuma bağlı bir durum değil, elden ele taşınabilen bir jetondur.

Tasarımın belkemiği, tek bir dürüst ödünleşimdir: **okuyucular yazarları asla bekletmez, yazarlar da okuyucuları**. Bir anlık görüntü açıkken, o görüntünün göreceği eski baytlar bir kenarda tutulur; hiçbir anlık görüntü açık değilken — ki olağan durum budur — bu ek defter hiç yaşamaz ve her yazmaya düşen maliyet, tek bir hafif atomik işaretten ibaret kalır. Bir okuma sonsuza dek açık kalıp eski sürümleri sınırsızca biriktirmesin diye, anlık görüntülerin bir ömür sınırı vardır; bu süreyi aşan bir anlık görüntü kendiliğinden geçersizleşir ve onun üzerinden yapılan bir okuma açıkça hata verir — ama hiçbir yazar, hiçbir okuyucuyu beklemek zorunda kalmaz. Bu, iyimser yazma yolunun "kimseyi boşuna bekletme" felsefesinin, okuma yoluna taşınmış halidir.

21.6 Kilitlenmeye karşı tasarımla bağışıklık

OxiDB iyimser bir sistem olduğu için, çoğu zaman hiç kilit almaz; bu, onuncu bölümdeki kilitlenme tehlikesini büyük ölçüde ortadan kaldırır. Ama bir işlemin, birden çok koleksiyona birden dokunması gereken durumlar vardır ve bu gibi yerlerde, koleksiyonların kilitlerinin alınması gerekebilir. İşte burada OxiDB, onuncu bölümde tanıttığımız en zarif disipline başvurur: kilitleri her zaman **aynı, belirli bir sırada** almak.

Onuncu bölümde, kilitlenmenin döngüsel bir bekleme olduğunu — birinci işlemin A'yı tutup B'yi, ikincinin B'yi tutup A'yı beklemesini — anlatmıştık. Eğer her işlem, kilitleri her zaman aynı sırayla alırsa, bu döngü hiç oluşamaz; çünkü iki işlem de önce aynı kilidi almaya çalışır ve biri diğerini beklerken ters bir bağımlılık kurulmaz. OxiDB, koleksiyon kilitlerini sıralı bir düzende aldığı için, kilitlenme **tasarım gereği imkânsızdır** — onu sezip çözmeye çalışan bir mekanizmaya bile gerek kalmaz. Bu, onuncu bölümdeki soyut "sıralı kilit disiplini" fikrinin, gerçek bir sistemde bir kilitlenme sınıfını tümüyle ortadan kaldıran somut bir uygulamasıdır.

Bu disiplinin OxiDB'deki uygulaması, küçük ama öğretici bir veri yapısı seçimine dayanır. Bir işlem boyunca, dokunduğu koleksiyonların adları **sıralı bir kümede** biriktirilir — adların kendiliğinden alfabetik düzende tutulduğu bir yapıda. Bu seçim, kilit sırasını ayrıca hesaplamayı gereksiz kılar: işlem tamamlanırken kümeyi gezmek, koleksiyon adlarını zaten her zaman aynı, belirli (alfabetik) düzende ziyaret etmek demektir. Hangi işlem hangi koleksiyonlara dokunmuş olursa olsun, hepsi aynı adı aynı sırada görür; dolayısıyla biri "A sonra B", öteki "B sonra A" sırasıyla kilit almaya asla çalışmaz. Döngüsel beklemenin önkoşulu — kilitlerin farklı işlemlerde farklı sırayla istenmesi — ortadan kaldırılmıştır. Sıralı kümenin değerinin "her zaman düzenli" oluşu, kilitlenmesizliği bir koşul değil, veri yapısının doğal bir sonucu haline getirir; bu, doğru veri yapısının bir doğruluk güvencesini nasıl bedavaya çevirebildiğinin zarif bir örneğidir.

21.7 Tek belge ile çok belge

Dördüncü ve onuncu bölümlerde, belge dünyasında atomikliğin doğal sınırının tek bir belge olduğunu vurgulamıştık. OxiDB'de bu doğrudan görülür: tek bir belgeyi değiştiren bir işlem, zaten atomiktir, çünkü o belge tek bir bütün olarak yazılır; burada karmaşık bir işlem makinesine gerek yoktur. Buraya kadar anlattığımız üç fazlı, sürüm-doğrulamalı işlem düzeneği, asıl olarak **birden çok belgeye ya da birden çok işleme** birden dokunan, hepsinin birlikte tutarlı kalması gereken durumlar içindir. Bu, dördüncü bölümdeki "tutarlı kalması gereken birim ne kadar büyük" sorusunun OxiDB'deki yankısıdır: birimi tek bir belgeye sığdırabiliyorsanız işler basit kalır; birim birçok belgeye yayıldığında, bu işlem düzeneği devreye girer.

21.8 Küme durumunda işlemler

Onuncu ve on ikinci bölümlerde, işlemlerin tek makinede zor, birçok makinede daha da zor olduğunu görmüştük. OxiDB tek bir düğümde, az önce anlattığımız iyimser düzeneği kullanır. Bir kümede çalıştığındaysa, tamamlanmış bir işlemin biriktirilmiş değişiklikleri, on ikinci bölümdeki konsensüs katmanına **tek bir bütün olarak** verilir; böylece işlemin tüm değişiklikleri ya birlikte replikasyona girer ya da hiçbiri girmez. Bunun ayrıntılarına, ölçeklendirmeyi ele aldığımız ileriki bölümde döneceğiz; şimdilik akılda tutulacak nokta, işlemin "ya hep ya hiç" niteliğinin, tek düğümden kümeye taşındığında da korunduğudur.

21.9 İyimserliğin bedeli

Onuncu bölümün dürüst dersini OxiDB bağlamında tekrar etmek gerekir: iyimser yaklaşım her zaman en iyisi değildir. Çatışmaların nadir olduğu durumlarda muhteşemdir; kimse boşuna beklemez ve işlemler hızla tamamlanır. Ama çatışmaların sık olduğu, birçok işlemin aynı belgeye saldırdığı durumlarda israflı olabilir; çünkü o işlemler sona kadar çalışıp, doğrulama fazında çatışma bulup iptal edilir ve yeniden denenir — yapılan iş boşa gider. OxiDB'nin iyimser tercihi, belge veritabanlarının tipik iş yüküne — çoğunlukla bağımsız belgelere dokunan, düşük çatışmalı yüklere — uygundur; ama herkesin aynı birkaç belgeye yarıştığı bir senaryoda, bu tercihin bedeli artar. Onuncu bölümde söylediğimiz gibi, doğru seçim her zaman iş yüküne bağlıdır.

21.10 Kötümser bir kaçış: satır kilidiyle okuma

Peki ya iyimserliğin bedelinin gerçekten ağır bastığı, herkesin aynı birkaç sıcak belgeye yarıştığı senaryolar? Böyle durumlarda, işlemleri sona kadar çalıştırıp çatışmada iptal etmek yerine, baştan bir kilit alıp beklemek daha ucuz olabilir. OxiDB, iyimser olmakla birlikte, bu kötümser kaçışı da sunar: bir okuma, okuduğu belgeleri aynı anda **güncelleme için kilitleyebilir**. İlişkisel dünyanın `SELECT ... FOR UPDATE` deyimiyle aynı anlamı taşıyan bu yol, bir belgeyi okuyup onun üzerinde bir karar verecek ve hemen ardından güncelleyecek bir işlemin, bu iki adım arasında başkasının araya girmeyeceğinden emin olmasını sağlar. Sıcak bir hesabın bakiyesini okuyup ondan düşen bir işlem, belgeyi güncelleme için okuyarak, çatışma-ve-yeniden-dene döngüsüne hiç girmeden ilerler. Böylece OxiDB, çoğunlukla iyimser kalırken, çatışmanın yoğunlaştığı o özel noktalarda kötümser bir aracı da elin altında bırakır — yine onuncu bölümdeki dersin bir yankısı: doğru araç, iş yükünün şekline göre seçilir.

21.11 Bu bölümün bıraktığı yer

Bu bölümde, OxiDB'nin işlem mekanizmasını yakın plana aldık. OxiDB'nin iyimser eşzamanlılık denetimini neden seçtiğini; değişiklikleri biriktirip, tamamlama anında üç fazlı bir düzenle — hazırlık, sürüm doğru-

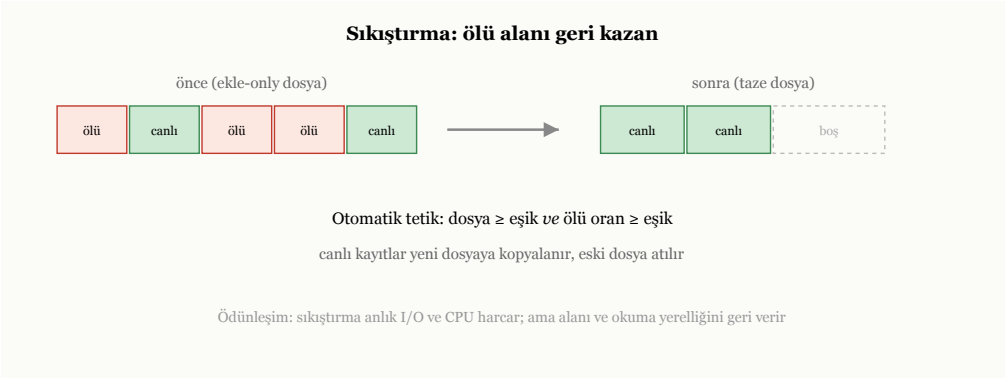
lama ve tamamlama — işleyişini; çatışmanın sürüm numaralarıyla nasıl saptanıp iptale yol açtığını; doğrulama ile uygulamanın bir tamamlama kilidiyle bölünmez kılınarak kayıp güncellemenin nasıl önlendiğini; işlemlerin, hazırlanmış mutasyonlar ve diske-uygulamadan-önce gelen bir tamamlama noktası aracılığıyla dayanıklılıkla nasıl bütünleştiğini ve grup tamamlamanın fsync'i nasıl amorti ettiğini; kilitlenmenin, sıralı bir kümenin doğal düzeninden gelen kilit disipliniyle nasıl tasarımdan dışlandığını; tek belge ile çok belge ayrımını; küme durumundaki davranışı; ve iyimserliğin bedelini gördük. Yazma yolunu hiç değiştirmeden, yalnızca okuma yolunda tutarlı bir görüntü sunan — bir toplamanın bir transferin yarısını asla görmemesini varsayılan kılan — okuma anlık görüntülerini; ve iyimserliğin pahalıya geldiği sıcak noktalarda başvurulacak kötümser bir kaçışı, satırı güncelleme için kilitleyerek okuma yolunu da izledik.

İşlemleri ele alırken, OxiDB'nin disk-öncelikli kipinin append-only doğasına birkaç kez değindik. Beşinci ve on altıncı bölümlerde söylediğimiz gibi, append-only depolama veriyi asla üzerine yazmaz; her güncelleme yeni bir kayıt ekler ve eskisi ölü alana dönüşür. Bu ölü alan zamanla birikir ve onu geri kazanmak gerekir. Bir sonraki bölümde, OxiDB'nin bu temizlik işini — sıkıştırmayı (compaction), onu ne zaman ve nasıl yaptığını, hatta bu kitap yazılırken eklenen otomatik tetikleyicisini — ele alacağız.

Bölüm 22

OxiDB'de Sıkıştırma: Ölü Alan ve Otomatik Tetikleme

İşlemleri ele alırken, OxiDB'nin disk-öncelikli kipinin append-only doğasına birkaç kez değindik. Beşinci ve on altıncı bölümlerde söylediğimiz gibi, append-only depolama veriyi asla üzerine yazmaz; her güncelleme yeni bir kayıt ekler ve eskisi ölü alana dönüşür. Bu ölü alan zamanla birikir ve onu geri kazanmak gerekir. Bu bölüm, OxiDB'nin bu temizlik işini — sıkıştırmayı, onu güvenle nasıl yaptığını ve bu kitap yazılırken eklenen otomatik tetikleyicisini — ele alıyor. Sıkıştırma, append-only bir motorun ayrılmaz, sessiz bakım işidir.



Şekil 22.1: Sıkıştırma: ölü alanın taze dosyaya kopyalanarak geri kazanılması.

22.1 Ölü alan nereden gelir

Beşinci bölümdeki muhasebe defteri benzetmesini hatırlayalım: append-only bir depo, hiçbir eski satırı silmez; bir kaydı güncellemek için onun yeni hâlini defterin sonuna yazar ve en son yazılanı geçerli sayar. Bunun kaçınılmaz sonucu, eski hâllerin defterde öylece durmaya devam etmesidir. Bir belgeyi yüz kez güncellerseniz, disk-öncelikli kipin veri dosyasında o belgenin yüz kopyası birikir; yalnızca sonuncusu geçerlidir, gerisi **ölü alandır**. Silmeler de benzer biçimde, kaydı fiziksel olarak çıkarmak yerine ölü olarak işaretler.

Bu olgunun kritik bir sonucu vardır: veri dosyasının boyutu, **yaşayan verinin** boyutuyla değil, **yapılan yazma sayısıyla** büyür. Yoğun güncelleme alan bir koleksiyonun veri dosyası, içindeki gerçek, geçerli veri küçük kalsa bile, zamanla çoğu ölü kayıttan oluşan bir dev haline gelebilir. Bu hem yer israfıdır hem de on altıncı bölümde gördüğümüz okuma ve tarama maliyetlerini artırır; çünkü daha büyük bir dosyada gezinmek gerekir. İşte bu şişkinliği gidermek, sıkıştırmanın görevidir.

22.2 Sıkıştırma ne yapar

Sıkıştırma, beşinci bölümde tanımladığımız temizlik işidir: veri dosyasını baştan yazıp, yalnızca **yaşayan** kayıtları taze bir dosyaya kopyalamak; ölü kayıtları geride bırakmak. Muhasebe defteri dolup taştığında oturup yalnızca hâlâ geçerli satırları temiz bir deftere geçirmeye benzer. İşlem bittiğinde, taze dosya yalnızca geçerli veriyi içerir ve eski, şişmiş dosyanın yerini alır; biriken tüm ölü alan geri kazanılmıştır. OxiDB üzerinde yapılan ölçümlerde, yoğun güncelleme sonrası şişmiş bir veri dosyasının, sıkıştırmayla birkaç kat küçüldüğü — ve bu sırada tüm yaşayan verinin ve indeksli sorguların hem sıkıştırma sırasında hem de yeniden açılışta eksiksiz kaldığı — doğrulandı.

Bu işin adımlarını yakından görmekte yarar var; çünkü sıkıştırmanın hem güvenliği hem de zarafeti bu adımların düzeninden gelir. Önce OxiDB, kimlik-konum dizinini gezerek **yaşayan kayıtların** kim olduğunu — hangi kimliğin dosyanın hangi konumunda durduğunu — bir anlık görüntü olarak çıkarır. Dizinde yer alan her kimlik, tanımı gereği canlıdır; ölü kayıtların hiçbiri dizinde değildir, çünkü güncelleme ya da silme, dizini her zaman en son geçerli konuma (ya da yokluğa) çevirmiştir. İkinci adımda, bu canlı kayıtların her birinin baytları eski dosyadan okunur ve yan yana, taze bir geçici dosyaya yazılır — on altıncı bölümde gördüğümüz gibi, bugün bu baytlar varsayılan olarak sıkıştırılmadan, olduğu gibi kopyalanır; sıkıştırma yalnızca açıkça istendiğinde bu sırada uygulanır. Her yazma, kaydın taze dosyadaki yeni konumunu döndürür ve yeni bir kimlik-konum listesi böylece kurulur. Üçüncü adımda, geçici dosya tek bir atomik yeniden-adlandırmayla (rename) asıl dosyanın yerine geçer; eski dosya kapanır, kaynakları serbest kalır. Son adımda, dizin temizlenip yeni konumlarla baştan kurulur — artık her kimlik, taze dosyadaki yeni yerine işaret eder. Yaşayan veri miktarı da güncellenir; çünkü sonraki ölü-alan ölçümleri buna dayanacaktır.

Belleğe öncelikli kipte ise sıkıştırma farklı bir anlam taşır. O kipte append-only bir veri dosyası yoktur; belgeler bellekteki eşlemede yerinde güncellenir, dolayısıyla biriken bir ölü alan da yoktur. Orada "sıkıştırma", yalnızca güncel bellek içeriğinin taze bir anlık görüntüsünü diske yazmaktan ibarettir. Yani sıkıştırma, asıl olarak disk-öncelikli kipin bir ihtiyacıdır.

22.3 Zor kısım: yaşayan sistemde güvenle yapmak

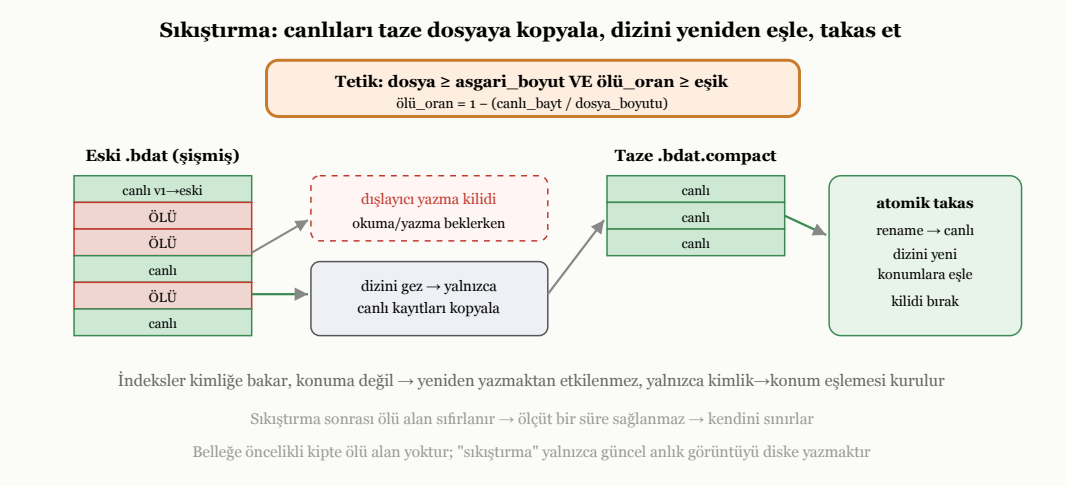
Sıkıştırmanın asıl zorluğu, onu veritabanı çalışmaya devam ederken, okuma ve yazmalar sürerken güvenle yapmaktır. Burada, on birinci bölümdeki eşzamanlılık kaygılarının somut bir örneği belirir ve OxiDB'nin çözümü öğreticidir.

Sorunu görelim. Disk-öncelikli kipte bir belgeyi okumak iki adımdır: kimlik-konum dizininden belgenin dosyadaki konumunu bulmak ve sonra o konumdan baytları okumak. Şimdi düşünün: bir okuyucu konumu öğrenmiş, tam o baytları okuyacakken, sıkıştırma araya girip dosyayı baştan yazsa ve belgeler yeni dosyada başka konumlara taşınsa ne olur? Okuyucunun elindeki konum, artık eski dosyaya aittir; yeni dosyada o konumda bambaşka bir şey olabilir. Bu, sessiz bir veri bozulmasıdır.

OxiDB bunu, veri dosyası tutamağı üzerinde bir okuma-yazma engeliyle çözer. Normal işlemler — okumalar ve yazmalar — bir **paylaşımlı okuma kilidi** tutar ve bu kilit, hem konumu bulma hem de baytları okuma adımlarının ikisini birden kapsar; yani bir konum, her zaman onu okuduğu dosyaya karşı kullanılır, asla başka bir dosyaya karşı değil. Sıkıştırma ise **dışlayıcı bir yazma kilidi** alır: bu kilidi tutarken hiçbir okuma ya da yazma araya giremez; sıkıştırma, taze dosyayı kurar, kimlik-konum dizinini yeni konumlara göre yeniden eşler ve dosyayı bütünüyle değiştirir; ancak bundan sonra kilidi bırakır. Böylece hiçbir konum, hiçbir zaman değiştirilmiş bir dosyaya karşı kullanılmaz. Bu, on birinci bölümdeki eşzamanlılık ilkelerinin — paylaşımlı ve dışlayıcı erişimin dikkatli düzenlenmesinin — gerçek bir doğruluk sorununu nasıl çözdüğünün somut bir örneğidir.

Bu çözümün ödünleşimi açıktır ve kasıtlıdır. Sıkıştırma, dışlayıcı kilidi tuttuğu süre boyunca koleksiyona giden tüm okuma ve yazmaları bekletir; yani sıkıştırma, koleksiyona kısa bir duraklama bindirir. OxiDB bu duraklamayı, doğru yere ödünleşim yaparak kabul eder: alternatif olan, sıkıştırmayı eşzamanlı işlemlerle iç içe yürütmeye çalışmak, hem çok daha karmaşık hem de yukarıda gördüğümüz konum-bozulması sınıfından hataya çok daha açık olurdu. Bunun yerine, sıkıştırmayı tam, dışlayıcı ama **kısa** bir kesit içine kapatmak —

ve onu sık değil, yalnızca ölü alan gerçekten biriktiğinde çalıştırmak — hem doğruluğu güvene alır hem de duraklamanın toplam maliyetini, az sonra göreceğimiz tetik ölçütüyle en aza indirir.



Şekil 22.2: Canlıları taze dosyaya kopyala, dizini yeniden eşle, dışlayıcı kilit altında takas et.

22.4 İndeksler neden sağ kalır

Sıkıştırmanın zarif bir yanı, indeksleri bozmadan çalışmasıdır ve bunun nedeni öğreticidir. On sekizinci bölümde gördüğümüz gibi, OxiDB'nin alan indeksleri belgeleri **kimliğe** göre tutar, fiziksel konuma göre değil. Sıkıştırma ise yalnızca belgelerin dosyadaki **konumlarını** değiştirir — kimliklerini değil. Bu yüzden veri dosyasını baştan yazmak, indeksleri geçersiz kılmaz; yalnızca kimlikten konuma giden o küçük eşlemenin yeniden kurulması yeterlidir. Kimlik ile fiziksel konum arasındaki bu ayrışma — indekslerin kimliğe, dizinin konuma bakması — sıkıştırmayı, indekslere hiç dokunmadan yapılabilir kılar. İyi bir soyutlama ayrımının, karmaşık bir işi nasıl basitleştirdiğinin güzel bir örneğidir bu.

22.5 Açık çağrıdan otomatik tetiklemeye

Sıkıştırmanın OxiDB'deki gelişimi, bu kitap yazılırken yaşanan iki aşamalı bir öyküdür ve onu anlatmak, mühendislik kararlarının nasıl olgunlaştığını gösterir.

İlk aşamada, sıkıştırma **açıkça çağrılan** bir işlemdi: ne zaman gerek duyduğunuza siz karar verir ve sıkıştırmayı elle başlatırdınız. Bu, az önce anlattığımız tüm güvenlik özelliklerine — yaşayan sistemde güvenli yeniden yazma, indekslerin sağ kalması — sahipti ve ölçümlerle doğrulanmıştı; ama ne zaman çalıştırılacağına karar vermek kullanıcıya kalıyordu.

İkinci aşamada, bir **otomatik tetikleyici** eklendi. Artık OxiDB, periyodik bakım sırasında ucuz bir ölü-alan ölçütüne bakar ve gerektiğinde sıkıştırmayı kendisi başlatır. Ölçüt şöyledir: veri dosyası hem belirli bir **asgari boyutu** aşmışsa hem de içeriğinin belirli bir **oranından fazlası ölüyse**, sıkıştırma tetiklenir. Ölü oran, basitçe, dosyanın ne kadarının yaşayan veri **olmadığıyla** ölçülür — yaşayan baytların dosya boyutuna oranı ne kadar düşükse, ölü oran o kadar yüksektir.

Bu ölçütü biraz daha somutlaştıralım, çünkü onun ucuzluğu, otomatik tetiklemeyi mümkün kılan şeydir. OxiDB, sıkıştırma kararını verirken iki sayıya bakar: dosyanın toplam boyutu — bir dosya-boyutu sorgusuyla anında öğrenilir — ve yaşayan verinin toplam boyutu — motorun her yazma ve silmede güncel tuttuğu bir sayaçtan okunur. Ölü oran, bu ikisinin oranından bir çıkarmayla bulunur: birden, yaşayan baytların dosya

boyutuna bölümü çıkarılır. Yani dosyanın yarısı hâlâ canlıysa ölü oran yarımdır; yalnızca beşte biri canlıysa ölü oran beşte dörttür. Bu hesap, tek bir dosya-boyutu okuması ile tek bir sayaç okumasından ibarettir; hiçbir belgeyi çözmez, dosyayı baştan sona taramaz. Bu yüzden tetik denetimi periyodik bakımda sürekli yapılabilecek kadar ucuzdur — sıkıştırmanın kendisi pahalı olsa da, *ona gerek olup olmadığını sormak* neredeyse bedavadır. Varsayılan eşikler de bu mantığa uygun seçilmiştir: dosyanın birkaç megabaytı aşması ve yarısından fazlasının ölü olması, makul bir tetik noktasıdır.

İki koşulun birlikte aranması bilinçlidir. Asgari boyut koşulu, küçük dosyalar için boşuna sıkıştırma yapılmasını engeller; çünkü küçük bir dosyada kazanılacak yer azdır, sıkıştırmanın maliyetine değmez. Ölü oran koşulu ise, ancak gerçekten kayda değer bir ölü alan biriktiğinde sıkıştırma yapılmasını sağlar; az miktarda ölü alan için dosyayı baştan yazmak israf olurdu. İki eşik birden sağlandığında, yani dosya hem yeterince büyük hem de yeterince ölüyse, sıkıştırma devreye girer.

Bu otomatik tetikleyicinin zarif bir özelliği, **kendini sınırlamasıdır**. Bir sıkıştırma yapıldığında, ölü alan sıfırlanır; dolayısıyla ölçüt bir süre daha sağlanmaz ve sıkıştırma boşuna tekrar tekrar tetiklenmez. Sistem, yalnızca ölü alan yeniden birikip eşiği aştığında tekrar sıkıştırır. Bu eşikler ayarlanabilir ve on altıncı bölümde gördüğümüz gibi per-koleksiyon belirlenebilir; istenirse otomatik tetikleme tümüyle kapatılıp sıkıştırma elle yönetilebilir.

22.6 Kütlesel silmenin gizli tuzağı

Otomatik tetikleyiciyi anlatırken, bu kitap yazılırken yakalanan öğretici bir hatadan söz etmek yerinde olur; çünkü append-only bir motorda ölü alanın en hızlı biriktiği an, tam da yüz binlerce belgenin bir anda silindiği andır. Silme, gördüğümüz gibi, kaydı fiziksel olarak çıkarmaz; onu ölü olarak işaretler. Bir koleksiyondan yüz binlerce belgeyi tek seferde sildiğinizde, geride yüz binlerce ölü işaret — mezar taşı (tombstone) — kalır ve bunların bir noktada süpürülüp alandan geri kazanılması gerekir.

İlk uygulamada, bu süpürme işi ölü işaretlerin sayısıyla **ikinci dereceden** büyüyordu: her ölü kaydı ele alırken, geri kalanların tümü yeniden gözden geçiriliyordu; yani n silme, n-kare düzeyinde iş doğuruyordu. Küçük silmelerde bu fark edilmezdi, ama üç yüz binlik bir kütlesel silmede sonuç çarpıcıydı — arka plandaki bakım iş parçacığı, işlemciyi tam kapasite meşgul ederek yarım saati aşkın süre dönüp durdu. Bu, on altıncı bölümde değindiğimiz asimptotik maliyet kaygısının, sessizce arka planda pusuya yatmış bir örneğidir: doğru çalışan ama ölçekle kötü büyüyen bir kod yolu, yalnızca yeterince büyük bir girdi gelene dek kendini gizler. Çözüm, süpürme işini ölü kayıt sayısıyla **doğrusal** büyüyecek biçimde yeniden kurmaktı; böylece kütlesel bir silme, artık orantılı — ve kısa — bir bakım işine yol açar. Bu tür tuzakların ders verici yanı, ancak gerçek ölçekte ölçüm yapıldığında ortaya çıkmalarıdır; birim testlerinin küçük girdileri, ikinci dereceden bir eğriyi doğrusaldan asla ayırt edemez.

22.7 Sıkıştırmanın bedeli ve dengesi

Sıkıştırma bedava değildir: tüm yaşayan veriyi baştan yazmayı gerektirir, yani bir disk yükü taşır; ve dışlayıcı kilidi tuttuğu kısa süre boyunca diğer işlemleri bekletir. Bu yüzden onu sürekli değil, ölü alan gerçekten büyüdüğünde yapmak gerekir — az önceki eşiklerin amacı tam olarak budur. Sıkıştırma, beşinci bölümde söylediğimiz gibi, arka planda yürüyen, amorti edilmiş bir bakım işidir: dosyayı sürekli temiz tutmaya çalışmak yerine, kirlilik belli bir düzeye ulaştığında toplu bir temizlik yapmak, hem maliyeti hem de kesintiyi en aza indirir.

22.8 Bu bölümün bıraktığı yer

Bu bölümde, append-only bir motorun ayrılmaz bakım işini — sıkıştırmayı — OxiDB bağlamında ele aldık. Ölü alanın nereden geldiğini ve veri dosyasını yazma sayısıyla nasıl şişirdiğini; sıkıştırmanın dört adımını

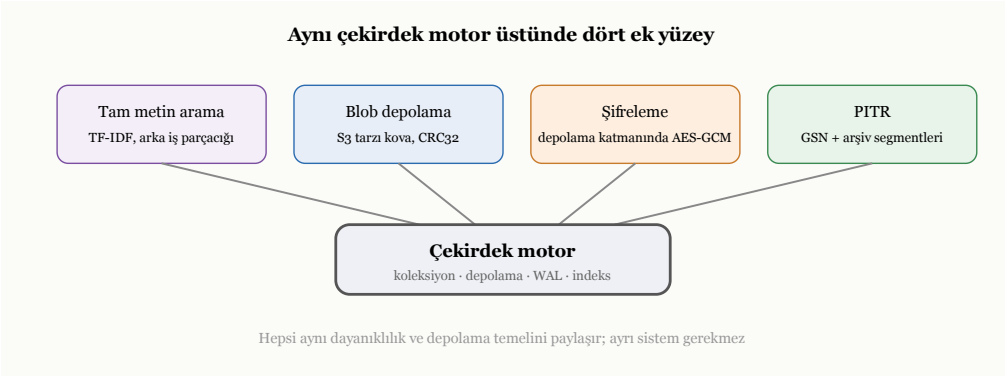
— canlı dizini görüntüle, canlı kayıtları taze dosyaya kopyala, atomik takas, dizini yeniden eşle — yalnızca yaşayan kayıtları geçirip ölü alanı geri kazanmasını; bunu yaşayan bir sistemde güvenle yapmak için kullanılan okuma-yazma engelini ve onun kısa-ama- dışlayıcı kesit ödünleşimini; indekslerin kimliğe dayandığı için neden sağ kaldığını; açık çağrıdan otomatik tetiklemeye uzanan gelişim öyküsünü ve otomatik tetikleyicinin — tek bir dosya-boyutu ile tek bir sayaç okumasından ibaret, bu yüzden neredeyse bedava — ölü-oran ölçütünü, kendini sınırlamasını ve dengesini gördük.

Buraya kadar Kısım III'te, OxiDB'nin çekirdek belge motorunu — depolama, dayanıklılık, indeks, sorgu, toplama, işlem ve sıkıştırma — eksiksiz dolaştık. Ama on beşinci bölümde söylediğimiz gibi, OxiDB klasik bir belge veritabanının ötesine geçen birkaç ek yetenek de sunar. Bir sonraki bölümde, bu ek yüzeylerden başlıcalarını — tam metin aramayı, büyük ikili nesneler için nesne deposunu, dururken şifrelemeyi ve zamanın bir noktasına geri dönmeyi sağlayan kurtarmayı — ele alacağız.

Bölüm 23

OxiDB'nin Ek Yüzeyleri: Tam Metin Arama, Blob Depolama, Şifreleme ve PITR

On beşinci bölümde, OxiDB'nin klasik bir belge veritabanının ötesine geçen birkaç ek yetenek sunduğunu söylemiştik. Kısım III boyunca buraya kadar çekirdek belge motorunu — depolama, dayanıklılık, indeks, sorgu, toplama, işlem ve sıkıştırma — dolaştık. Bu bölüm, çekirdeğin etrafındaki dört önemli ek yüzeyi ele alıyor: tam metin aramayı, büyük ikili nesneler için blob depolamayı, dururken şifrelemeyi ve zamanın bir noktasına geri dönmeyi sağlayan kurtarmayı. Bu dört yüzeyin ortak özelliği, hepsinin aynı çekirdek motorun üzerine oturması ve isteğe bağlı olmasıdır; kullandığınız kadarının bedelini ödersiniz.



Şekil 23.1: Tek çekirdek motor üstündeki dört ek yüzey.

23.1 Tam metin arama: ters indeksin somutu

Yedinci bölümde, metnin içinde sözcük aramayı mümkün kılan **ters indeksi** — "sözcük, o sözcüğü içeren belgeler" eşlemesini — tanımıştık ve metin aramanın alaka düzeyiyle sıralama gerektirdiğini söylemiştik. OxiDB'nin tam metin arama yeteneği, bu fikrin doğrudan uygulamasıdır.

Bu yetenek birkaç bakımdan dikkate değerdir. Birincisi, yalnızca düz metni değil, **zengin biçimleri** de indeksleyebilmesidir: web sayfalarından, ofis belgelerinden, taşınabilir belge dosyalarından, hatta görüntülerden metin çıkarıp indeksleyebilir. Yani bir belgeye eklenmiş bir dosyanın içeriği bile aranabilir hale gelir. İkincisi, indeklemeyi **arka planda** yapmasıdır. Metni sözcüklere ayırıp ters indeksi güncellemek pahalı bir

iştir; bunu her yazmanın ortasında yapmak, yazmaları yavaşlatırdı. Bunun yerine OxiDB, indeksleme işlerini bir kuyruğa koyar ve onları ayrı bir çalışan iş parçacığında, yazma yolunun dışında işler. Bunun bir sonucu vardır ve onu dürüstçe söylemek gerekir: arama indeksi, yazmalarla **nihai olarak tutarlıdır** — yani yeni yazılan bir belge, çok kısa bir gecikmeyle aranabilir hale gelir. Bu, on birinci bölümdeki nihai tutarlılık fikrinin, bir alt sistem düzeyindeki küçük bir yankısıdır: hız uğruna, anlık tutarlılıktan biraz ödün verilir.

Üçüncüsü, **alaka puanlamasıdır**. Yedinci bölümde, bir sözcüğü içeren yüzlerce belgenin hepsinin aynı ölçüde ilgili olmadığını söylemiş, terim sıklığı ile ters belge sıklığını birleştiren puanlamayı tanımıştık. OxiDB, sonuçları, o yaklaşımın bugünkü olgun biçimiyle — arama motorlarında fiilen standart haline gelmiş bir puanlama yöntemiyle — sıralar. Bu yöntem, klasik terim-sıklığı fikrinin iki önemli iyileştirmesini içerir. Birincisi **doygunluktur**: bir sözcüğün belgede on kez yerine yüz kez geçmesi, alakayı on kat artırmaz; katkı, bir eğriyle yavaşça doyuma ulaşır, böylece bir sözcüğü saplantılı biçimde tekrarlayarak puanı şişirmek mümkün olmaz. İkincisi **uzunluk normalleştirmesidir**: bir sözcüğün uzun bir belgede geçmesi, kısa bir belgede geçmesinden daha az şey ifade eder; çünkü uzun belgede her sözcüğün geçme olasılığı zaten yüksektir. Puanlama, belge uzunluğunu, koleksiyondaki ortalama belge uzunluğuna göre tartar. Bu iki ayarın da, davranışı belirleyen iki katsayısı vardır; OxiDB bunları, yaygın arama motorlarıyla aynı varsayılan değerlere ayarlar ve gerektiğinde ortam değişkenleriyle değiştirilebilir kılar. Sonuç olarak, sözcüğün yoğun geçtiği kısa bir belge, onu bir kez içeren uzun bir belgenin önüne çıkar. Bu, yedinci bölümün "alaka düzeyi" kavramının somut, ince ayarlanmış karşılığıdır.

Bir ayrıntı daha, çok dilli metin için önemlidir: indeksleme, sözcükleri **köklerine indirger** (stemming). Yani "kitaplar", "kitabı" ve "kitap" gibi biçimler, ortak bir köke eşlenerek aynı sözcükmüş gibi aranabilir hale gelir. OxiDB, aralarında Türkçenin de bulunduğu pek çok dil için kök bulma desteği taşır ve dili bir ortam değişkeniyle seçtirir; yapısı bakımından Türkçe gibi sondan eklemeli diller için bu, doğru sonuçların ön koşuludur.

23.2 Tam metin aramanın iç mekaniği

Bu yüzeyin nasıl çalıştığını biraz daha yakından görelim; çünkü ayrıntıları, yedinci bölümün soyut ters indeksini somuta bağlar. İndeksin kalbinde iki eşleme durur. Birincisi, her sözcük için, o sözcüğü içeren belgelerin **gönderim listesidir** (posting list): her gönderim, hangi belgede sözcüğün kaç kez geçtiğini ve hangi konumlarda geçtiğini tutar. Konum bilgisini saklamak, ileride sözcüklerin ardışıklığına bakan tümce aramalarını mümkün kılar; sıklık ise alaka puanlamasının girdisidir. İkinci eşleme, her belge için, ait olduğu kova ve anahtar ile toplam terim sayısını tutar; toplam terim sayısı, az önce anlattığımız uzunluk normalleştirmesinin paydası olur. İndeks ayrıca, koleksiyon genelindeki toplam terim sayısını da tutar, ki ortalama belge uzunluğu hesaplanabilsin.

Arka plandaki çalışma modeli, dürüstçe söylenmesi gereken nihai tutarlılığın kaynağıdır. İndeksleme istekleri, sınırlı kapasiteli — en fazla iki yüz elli altı işlik — bir kuyruğa konur; ayrı bir çalışan iş parçacığı bu kuyruğu boşaltır. Kuyruğun **sınırlı** olması bilinçli bir tercihtir: yazmalar indekslemeden çok daha hızlı gelirse, kuyruk dolar ve yazma yolu kısa süreliğine bekler. Böylece sınırsız büyüyen, belleği tüketen bir birikim önlenir; geri-basınç (back-pressure) ile sistem kendini dengeler. Çalışan iş parçacığı, her belgeyi alır, içeriğinden metni çıkarır, sözcüklere ayırır, köklerine indirger ve gönderim listelerini günceller. Bu işin yazma yolunun dışında olması, asıl ekleme işleminin hızlı kalmasını sağlar; bedeli, yeni belgenin aranabilir olmasındaki küçük gecikmedir.

İndeksin **kalıcılığı** da kayda değerdir. Ters indeks, veri dizini altında ayrı bir dosyaya yazılır; böylece sunucu yeniden başladığında indeks sıfırdan kurulmak zorunda kalmaz. Çok sayıda yazma ardı ardına geldiğinde, indeksi her güncellemede baştan sona diske dökmek savurgan olurdu; bu yüzden OxiDB, güncellemeleri toplu hale getirip belirli aralıklarla bir kez yazar. Bu, on altıncı ve on yedinci bölümlerde gördüğümüz "yazmayı toplulaştır, sonra topluca boşalt" örüntüsünün, bu alt sistemdeki yankısıdır.

23.3 Blob depolama: büyük ikili nesnelere yer açmak

Dördüncü bölümde, sınırsız büyüyen ya da çok büyük verileri bir belgenin içine gömmenin tuzağına değinmiştik: belge şişer, her okumada o yük taşınır. Büyük ikili nesneler — görüntüler, dosyalar, medya — bu tuzağın en belirgin örneğidir; bir megabaytlık bir görüntüyü bir belgenin içine gömmek, o belgeyi her okuduğunuzda o megabaytı da taşımak demektir.

OxiDB, bu ihtiyaca ayrı bir **blob depolama** yüzeyiyle yanıt verir. Bu yüzey, büyük ikili nesneleri belgelerin dışında, kovalar (bucket) halinde düzenlenmiş bir nesne deposunda tutar; her nesne, içeriği taşıyan bir veri dosyası ile onu betimleyen ayrı bir üst veri dosyasından oluşur. Üst veri dosyası, nesnenin anahtarını, ait olduğu kovayı, boyutunu, içerik türünü, oluşturulma zamanını, kullanıcı tanımlı etiketlerini ve içeriğin bozulmadığını doğrulayan bir **bütünlük damgasını** (etag) tutar. Damga, bir sağlama toplamıdır (checksum): nesne okunduğunda yeniden hesaplanıp saklananla karşılaştırılarak, baytların yolda ya da diskte sessizce bozulmadığı doğrulanabilir. Bu, yaygın bir bulut nesne deposu arayüzüyle aynı mantığı izler: yapılandırılmış belge verisini bir yerde, büyük opak baytları başka bir yerde tutmak.[1]

İki ince mühendislik kararı bu yüzeyi pratikte verimli kılar. Birincisi, **seçici sıkıştırmadır**: blob deposu, on altıncı bölümde gördüğümüz sıkıştırmayı nesnelere de uygulayabilir, ama görüntü, ses, video ya da sıkıştırılmış ofis belgeleri gibi zaten doygun-entropili türleri tanır ve onları yeniden sıkıştırmaya kalkmaz; çünkü böyle veride sıkıştırma neredeyse hiç yer kazandırmaz, yalnızca işlemci harcar. İkincisi, **biçim sürümlemesidir**: her üst veri dosyası bir sürüm numarası taşır ve eski bir motor, tanımadığı daha yeni bir sürümü okumayı reddeder. Bu, ileriye dönük bir emniyet supabıdır — yeni alanlar ekleyen bir motorun yazdığı veriyi, eski bir ikili dosyanın yanlış yorumlayıp sessizce bozmasını önler. Tüm bunlar, dördüncü bölümün ilkesiyle tam örtüşür: belgeleri yalın ve hızlı okunur tutmak için, büyük ve gömülmesi sakıncalı veriyi belgeden ayırıp ona referansla işaret etmek. Böylece belgeleriniz küçük kalır, büyük nesneler ise onlara uygun, ayrı bir depoda verimli biçimde yönetilir.

Burada anlattığımız, blob deposunun çekirdeğe gömülü yüzüdür; bu depoya bir ağ üzerinden, yaygın araçların doğrudan konuşabildiği tam bir bulut nesne deposu protokolüyle de erişilebilir. O uyumluluk katmanını — kova, anahtar ve etag kavramlarını bir HTTP arayüzüne taşıyan yüzeyi — otuz birinci bölümde ayrıntısıyla ele alacağız; burada yalnızca, aynı deponun iki kapısı olduğunu belirtmekle yetinelim.

23.4 Dururken şifreleme: depolama sınırında koruma

On dördüncü bölümde, şifrelemenin iki cephesinden söz etmiştik: aktarım sırasında ve dururken. OxiDB, dururken şifrelemeyi, depolama katmanına **saydam** biçimde yerleştirir. "Saydam" olması şu anlama gelir: üstteki katmanlar — sorgu, indeks, işlem — şifrelemenin varlığından habersizdir; şifreleme, yalnızca veri diske inmeden hemen önce ve diskten okunduktan hemen sonra, depolama sınırında devreye girer. On altıncı bölümde, baytların diske yazılmadan önce bir hazırlık adımından — sıkıştırma ve ardından şifreleme — geçtiğine değinmiştik; işte dururken şifreleme tam o adımda yapılır.

Kullanılan şifreleme, kimliği doğrulanmış bir simetrik şifredir: yani yalnızca veriyi gizlemekle kalmaz, aynı zamanda her şifreli parçanın yanına, onun değiştirilmediğini doğrulayan bir doğrulama etiketi koyar. Bu önemlidir, çünkü gizlilik ile bütünlük ayrı şeylerdir: birincisi baytların okunamamasını, ikincisi ise baytlarla oynanmadığını güvence altına alır. Diskteki şifreli veriyi gizlice değiştirmeye kalkan bir saldırgan, bu etiket sayesinde yakalanır; çözme sırasında etiket tutmazsa, veri kabul edilmez. Şifreleme, on altıncı bölümde değindiğimiz hazırlık adımında, sıkıştırmadan **sonra** uygulanır; bu sıra anlamlıdır, çünkü şifrelenmiş veri rastgeleye yakın görünür ve sonradan sıkıştırılamaz — bu yüzden önce sıkıştırılır, sonra şifrelenir.

Bu yetenek isteğe bağlıdır: motora bir şifreleme anahtarı verirseniz açılır, vermezseniz hiçbir bedeli olmaz. Açıldığında, diske yazılan her şey şifrelenir; böylece diski ya da dosyaları çalan biri, on dördüncü bölümde söylediğimiz gibi, yalnızca anlamsız baytlar görür. Ama o bölümün dürüst uyarısı burada da geçerlidir: şifreleme yalnızca anahtarı kadar güvenlidir, ve dururken şifreleme çalınan diske karşı korur, çalışan sisteme

[1] Bu model, S3 tarzı nesne depolama arayüzlerini izler: kova, anahtar, üst veri ve etag kavramları aynı soydan gelir.

zaten meşru biçimde girmiş bir saldırgana karşı değil. Anahtarın güvenli tutulması, bu korumanın ayrılmaz parçasıdır.

23.5 Zaman-noktasına kurtarma: çökmenin ötesinde

Altıncı bölümde, kurtarmanın sizi en son tamamlanmış duruma — çökmeden hemen öncesine — geri getirdiğini görmüştük. Ama bazen istenen, çökmeden geri dönmek değil, **zamanda geriye gitmektir**. Hatalı bir güncelleme tüm bir koleksiyonu bozmuş olabilir; yanlış bir komut önemli veriyi silmiş olabilir; kötü bir dağıtım, saatler boyunca hatalı veri yazmış olabilir. Bu durumlarda, sistemin son tutarlı haline değil, **belirli bir geçmiş ana** dönmek istersiniz. OxiDB'nin zaman-noktasına kurtarma yeteneği — kısaca PITR — tam da bunu sağlar.

Bu yeteneğin nasıl çalıştığını kavramsal olarak görelim. OxiDB, bu yetenek açıkken, dayanıklı kılınan her yazmaya **küresel, sürekli artan ve duvar saatiyle damgalanmış** bir sıra numarası verir; buna küresel sıra numarası (GSN, *global sequence number*) diyelim. Bu numaranın çözdüğü ince bir sorun vardır. Koleksiyonların her birinin kendi yazma-öncesi günlüğü ayrı bir bayt akışıdır; aralarında doğal bir küresel sıra yoktur. Belirli bir ana geri dönebilmek için ise, tüm koleksiyonlar boyunca **tek bir zaman ekseni** gerekir. GSN, bu ekseni sağlar: hangi koleksiyona ait olursa olsun, her dayanıklı yazma, bu tek sayaçtan artan bir numara alır.

Bu sayacın **çökmeler boyunca da artan kalması** gerekir; aksi halde pazartesi günkü 5 numara ile salı günkü 5 numara karışır. Ama canlı günlük temiz kapanışta kırpıldığı için, en yüksek değeri ondan okuyamayız. OxiDB bu yüzden, sayacın tavanını küçük bir dosyaya **kira (lease) bloklarıyla** yazar: açılışta, tek bir disk eşitlemesiyle on binlik bir blok rezerve eder ve numaraları bellekten dağıtır; blok tükenince dosyaya yeniden dokunur. Böylece numara başına değil, on bin yazma başına bir disk eşitlemesi yapılır. Bir çökme, en fazla bir blok kadar numarayı boşa harcar — bu zararsızdır, çünkü ileride göreceğimiz oynatma, atlanan numaralara tahammül eder.

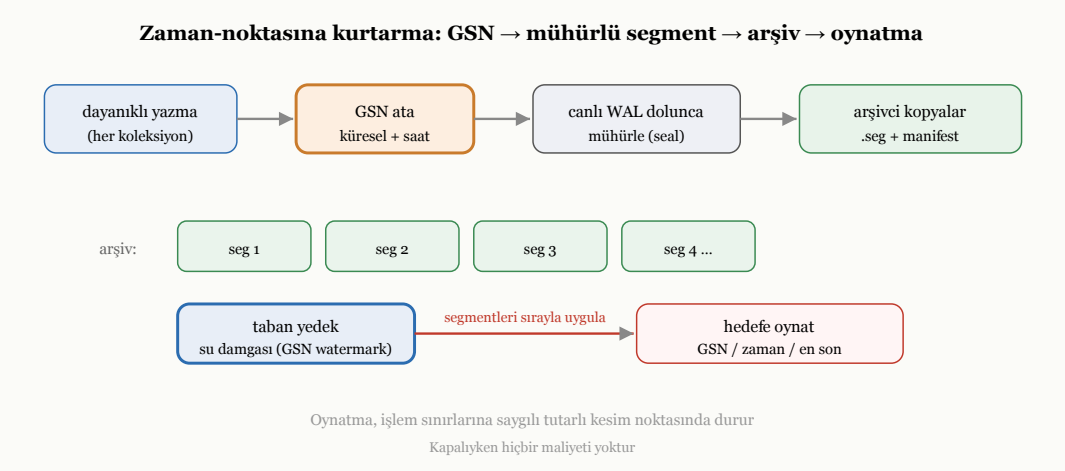
Yazma-öncesi günlük dolup döndükçe, eski bölümleri silinmek yerine **mühürlenir** (seal): canlı günlük belirli bir boyuta ulaştığında, OxiDB onu kilit altında, atomik bir yeniden adlandırmayla numaralı bir mühürlü parçaya çevirir ve yepyeni bir canlı günlük açar. Arka planda çalışan bir **arşivci**, bu mühürlü parçaları, baytları birebir kopyalayarak — sonlarına bir doğrulama eki ekleyerek — arşiv dizinine taşır. Arşivin dizini, parçaların kendi eklerinden **yeniden türetilebilir**: yani dizin dosyası bozulsa bile, parçalar tarandığında kendini onarır. Bu, altıncı bölümdeki "asıl gerçek günlüktedir, türetilenler ondan yeniden kurulur" ilkesinin bir başka uygulamasıdır.

Bir yedek alındığında, o yedek bir taban anlık görüntüsü ile birlikte, "bu yedek şu sıra numarasına kadarki durumu içerir" diyen bir **su damgası** (watermark) taşır. Belirli bir noktaya geri dönmek istediğinizde, OxiDB bu tabandan başlar ve arşivlenmiş günlüğü, seçtiğiniz ana — bir sıra numarasına, bir zaman damgasına ya da "en sona" — kadar ileri **oynatır**. Su damgası ile hedef arasında kalan parçalar sırayla uygulanır; tabandan önceki numaraları taşıyan girdiler atlanır. Üstelik bu oynatma, işlem sınırlarına saygı gösteren, tutarlı kesim noktalarında durur; yani yarım kalmış bir işlemin ortasında değil, temiz bir noktada geri dönersiniz.

Bu model — küresel sıra numarası, arşivlenen günlük, taban yedek ve ileri oynatma — altıncı bölümdeki dayanıklılık fikrinin zengin bir uzantısıdır. Çökmeden kurtarma, günlüğü taban üzerine son ana kadar oynatır; PITR ise aynı günlüğü, **seçtiğiniz** bir ana kadar oynatır. İkisi de aynı temel fikre — niyeti günlüğe yaz, sonra oynat — dayanır; PITR, o fikri zaman boyunca esnetir. Bu da isteğe bağlıdır; kapalıyken hiçbir maliyeti yoktur.

23.6 Ortak iplik: tek motor, isteğe bağlı yetenekler

Bu dört yüzeyin hepsinin ortak bir özelliği vardır ve onu görmek, OxiDB'nin tasarım felsefesini özetler. Hepsi, aynı çekirdek motorun üzerine oturur; hiçbiri ayrı, kopuk bir sistem değildir. Ve hepsi isteğe bağlıdır: tam



Şekil 23.2: GSN'den arşive ve oynatmaya: zaman-noktasına kurtarma yolu.

metin araması istemiyorsanız indeksleme yapılmaz; büyük nesneleriniz yoksa blob deposu boş kalır; şifreleme istemiyorsanız anahtar vermezsiniz; zaman-noktasına kurtarmaya ihtiyacınız yoksa arşivleme hiç çalışmaz. Kullanmadığınız bir yetenek, bir bedel getirmez. Bu, on beşinci bölümde değindiğimiz "tek çekirdek, çok yüz" felsefesinin bir başka yüzüdür: çekirdek sade ve odaklı kalır, ek yetenekler ise onun üzerine, gerektiğinde devreye giren katmanlar olarak eklenir.

Bir başka ortak nokta, her birinin bu kitapta daha önce kurduğumuz bir kavramın somut bir örneği olmasıdır. Tam metin arama, yedinci bölümün ters indeksidir; blob depolama, dördüncü bölümün "büyüyü gömme, ayır" ilkesidir; dururken şifreleme, on dördüncü bölümün şifreleme cephesidir; zaman-noktasına kurtarma, altıncı bölümün dayanıklılık fikrinin zamana yayılmış halidir. Yani bu ek yüzeyler, OxiDB'ye özgü tuhaflıklar değil, kitabın ilk iki kısmında öğrendiğimiz ilkelerin gerçek dünyadaki uygulamalarıdır.

23.7 Bu bölümün bıraktığı yer

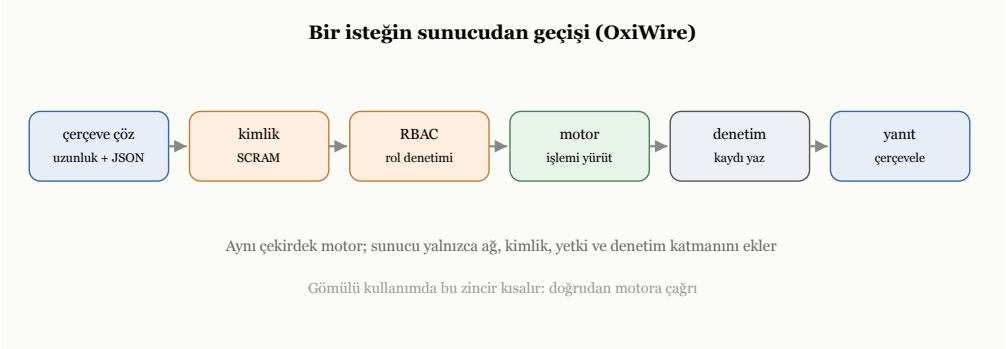
Bu bölümde, OxiDB'nin çekirdeğin ötesindeki dört ek yüzeyini ele aldık. Tam metin aramanın, arka planda çalışan, zengin biçimleri bile indeksleyen ve alaka puanlamasıyla sıralayan bir ters indeks olduğunu; blob depolamanın büyük ikili nesneleri belgelerden ayrı, kova tabanlı bir nesne deposunda tuttuğunu; dururken şifrelemenin depolama sınırında saydam biçimde çalışan, isteğe bağlı bir koruma olduğunu; ve zaman-noktasına kurtarmanın, dayanıklılık fikrini zamanda geriye gidebilecek biçimde genişlettiğini gördük. Hepsinin aynı motora oturduğunu, isteğe bağlı olduğunu ve daha önce öğrendiğimiz ilkelerin somut örnekleri olduğunu da gördük.

Buraya kadar OxiDB'nin yeteneklerinden söz ettik, ama bu yeteneklere uzaktan, bir ağ üzerinden nasıl erişildiğine hiç değinmedik. Gömülü kipte her şey doğrudan işlev çağrılarıyla olur; ama OxiDB bir sunucu olarak çalıştığında, istemcilerle bir ağ protokolü üzerinden konuşmalı, kimlik doğrulamalı, yetkilendirmeli ve güvenliği sağlamalıdır. Bir sonraki bölümde, OxiDB'nin sunucu katmanını — kendi ikili iletişim protokolünü, kimlik doğrulamasını, rol tabanlı erişim denetimini ve denetim günlüğünü — on dördüncü bölümdeki güvenlik ilkeleriyle bağlayarak ele alacağız.

Bölüm 24

OxiDB'nin Sunucu Katmanı: OxiWire Protokolü, Kimlik Doğrulama, RBAC ve Denetim

Buraya kadar OxiDB'nin yeteneklerinden söz ettik, ama bu yeteneklere uzaktan, bir ağ üzerinden nasıl erişildiğine değinmedik. Birinci ve on beşinci bölümlerde gördüğümüz gibi, OxiDB gömülü kipte çalışırken her şey doğrudan işlev çağrılarıyla olur; orada ağ, kimlik doğrulama, protokol diye bir sorun yoktur. Ama OxiDB bir **sunucu** olarak çalıştığında, istemcilerle bir ağ protokolü üzerinden konuşmalı, kimlik doğrulamalı, yetkilendirmeli ve güvenliği sağlamalıdır. Bu bölüm, OxiDB'nin sunucu katmanını — kendi iletişim protokolünü, kimlik doğrulamasını, rol tabanlı erişim denetimini ve denetim günlüğünü — on dördüncü bölümdeki güvenlik ilkeleriyle bağlayarak ele alıyor.



Şekil 24.1: Bir isteğin sunucu katmanlarından geçişi.

24.1 Çerçeveleme sorunu ve OxiWire protokolü

Bir ağ bağlantısı, özünde, kesintisiz bir bayt akışıdır; mesajlar arasında doğal bir sınır yoktur. Bu yüzden bir protokolün çözmesi gereken ilk sorun, **çerçevelemedir** (framing): alıcının, bir mesajın nerede bitip diğerinin nerede başladığını bilmesi. OxiDB bunu, her mesajın önüne onun uzunluğunu dört baytlık bir tamsayı olarak yazarak çözer: alıcı önce bu dört baytı okuyup uzunluğu öğrenir, sonra tam o kadar bayt okuyarak mesajı bütün olarak alır, ardından bir sonraki mesajın uzunluğunu bekler. Uzunluğun bir **üst sınırı** vardır — on altı mebibayt; bu sınırı aşan bir uzunluk bildirimi, daha bir bayt yük okunmadan reddedilir. Bu, güvenlik açısından önemlidir: kötü niyetli ya da bozuk bir istemcinin, "şu kadar gigabayt gelecek" diyerek sunucuyu

o kadar bellek ayırmaya zorlamasını ve böylece bir hizmet engelleme saldırısı düzenlemesini önler. Bu basit ama sağlam çerçeveleme, akış üzerinde mesajları net biçimde ayırır.

Mesajın içeriği iki biçimde kodlanabilir ve sunucu, gelen mesajın ilk baytına bakarak hangisinin kullanıldığını anlar. Birincisi, insan tarafından okunabilir, JSON tabanlı bir biçimdir; hata ayıklamak ve basit istemciler yazmak kolaydır. İkincisi, **OxiWire** adı verilen, sıkı paketlenmiş bir ikili biçimdir; bu biçim, bir baytlık ayırt edici bir önek ile başlar ve gövdesini, yaygın bir ikili serileştirme biçimiyle kodlar. İkili biçimin avantajı, JSON'un metinsel ağırlığını taşımamasıdır: alanları çift tırnak, virgül ve boşlukla işaretlemek yerine, değerleri kompakt ikili etiketlerle kodlar; böylece hem hat üzerinde daha az bayt gider, hem de ayrıştırma daha ucuzdur. Burada dürüst bir nitelik gerekir: ikili biçim, hattaki bayt sayısını ve ayrıştırma maliyetini azaltır, ama yine de değerleri kodlayıp çözmek için bir dönüşüm adımı içerir; sıfır-kopya bir aktarım değildir. Kazanç, biçimin metinden kompaktlığında ve ucuz ayrıştırılabilirliğindedir.

Bir istek, özünde, bir **komut** ve onun argümanlarından oluşur: ne yapılacağı (ekle, bul, güncelle, topla…) ve hangi koleksiyon üzerinde, hangi sorguyla. Sunucu bu isteği alır, çözer ve birazdan göreceğimiz aşamalardan geçirerek işler. OxiDB'nin neden genel amaçlı bir web protokolü yerine kendi protokolünü kullandığı, verimlilikledir: veritabanı yüküne uygun, hafif bir çerçeveleme ve ikili bir seçenek sunar. Yine de, on beşinci ve yirmi üçüncü bölümlerde değindiğimiz gibi, web istemcileri için ayrı bir HTTP arayüzü de mevcuttur; her istemci, kendine en uygun kapıdan girer.

24.2 Bağlantı modeli ve el sıkışma

Bir sunucu, aynı anda birçok istemci bağlantısını karşılamak zorundadır. OxiDB, gelen istekleri işlemek için bir çalışan havuzu kullanır: belirli sayıda iş parçacığı, gelen isteklere paralel olarak hizmet verir. Uzun süre boşta kalan bağlantılar, kaynakları boşa tutmamak için bir süre sonra kapatılır. Bu ayarlar — kaç çalışan, ne kadar boşta kalma süresi, hangi adres ve veri dizini — ortam değişkenleriyle yapılandırılır; sunucuyu işletmenin pratik düğmeleridir bunlar.

Bir istemci ilk bağlandığında, kimlik doğrulamadan önce küçük bir **el sıkışma** gerçekleşir: sunucu kendini tanıtır, yeteneklerini bildirir ve iki taraf nasıl konuşacaklarını kararlaştırır. Bu, kimlik doğrulamanın öncesinde, herhangi bir gizli bilgi gerektirmeyen bir adımdır; asıl güvenlik denetimleri ondan sonra başlar.

24.3 Kimlik doğrulama: SCRAM ile parolayı hattan geçirmeden

On dördüncü bölümde, kimlik doğrulamanın iki temel kuralını görmüştük: parolalar asla düz metin saklanmaz, yavaş ve tuzlanmış bir özetle tutulur; ve parolanın kendisi ağ üzerinden gönderilmez, bir meydan-okuma yanıt yöntemiyle bilindiği kanıtlanır. OxiDB, sunucu kimlik doğrulamasında tam olarak bu iki kuralı uygular.

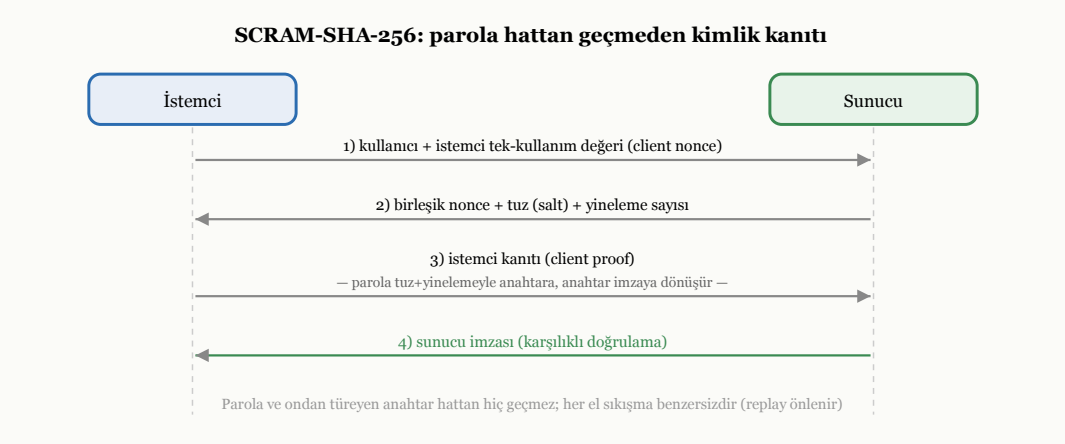
OxiDB, parolaları doğrularken **SCRAM-SHA-256** adlı, yaygın olarak kullanılan, standartlaştırılmış bir meydan-okuma yanıt protokolü kullanır.[1] Bu protokolün iç işleyişi, on dördüncü bölümdeki iki kuralın zarif bir somutlaşmasıdır; adımlarını görelim.

İstemci, kullanıcı adıyla birlikte rastgele bir sayı — bir **istemci tek-kullanım değeri** (client nonce) — gönderir. Sunucu yanıtında, istemci tek-kullanım değerine kendi rastgele sayısını — **sunucu tek-kullanım değerini** ekler, ayrıca bu kullanıcıya ait **tuzu** (salt) ve özet için kaç yineleme yapılacağını bildirir. İki tarafın da birleşik tek-kullanım değerini paylaşması, her el sıkışmayı benzersiz kılar ve daha önce yakalanmış bir yanıtın yeniden oynatılmasını (replay) önler. İstemci, parolasını tuz ve yineleme sayısıyla yavaş bir özete sokarak bir anahtar türetir; bu anahtarla, el sıkışmanın tüm mesajlarını kapsayan bir imza — bir **istemci kanıtı** (client proof) — hesaplar ve yalnızca bu kanıtı gönderir. Parolanın kendisi de, ondan doğrudan türetilen anahtar da hattan hiç geçmez. Sunucu, kendi sakladığı doğrulayıcıdan aynı hesabı yaparak kanıtı denetler;

[1] T. Hansen, "SCRAM-SHA-256 and SCRAM-SHA-256-PLUS Simple Authentication and Security Layer (SASL) Mechanisms," RFC 7677, 2015. SCRAM çatısının kendisi RFC 5802'de (A. Menon-Sen v.d., 2010) tanımlanır.

isterse, istemcinin de sunucuyu doğrulayabilmesi için karşılık bir imza döndürür, böylece doğrulama **karşılıklı** olur.

Sunucunun sakladığı şey de önemlidir: parolanın kendisi değil, tuz, yineleme sayısı ve ondan türetilen anahtarlardır. Tuzlama, aynı parolayı kullanan iki kullanıcının diskte aynı görünmesini önler; yüksek yineleme sayısı, özetlemeyi kasıtlı olarak yavaşlatarak kaba kuvvet (brute force) denemelerini pahalı kılar. Böylece veritabanı sızsa bile parolalar geri elde edilemez. Bu, on dördüncü bölümdeki soyut güvenlik ilkelerinin gerçek bir sistemde nasıl somutlaştığının net bir örneğidir.



Şekil 24.2: SCRAM el sıkışması: tek-kullanım değerleri ve kanıtın akışı.

24.4 Yetkilendirme: rol tabanlı geçit

Kimlik doğrulandıktan sonra, on dördüncü bölümdeki ikinci soru gelir: bu kullanıcı ne yapabilir? OxiDB, bunu rol tabanlı bir erişim denetimiyle yanıtlar ve üç temel rol tanımlar. **Yönetici** rolü her şeyi yapabilir. **Okuma-yazma** rolü, belge ekleme, güncelleme, silme, indeks oluşturma, toplama, işlem gibi veriyle çalışmanın olağan işlemlerini yapabilir, ama kullanıcı yönetimi gibi yönetimsel işlemleri yapamaz. **Okuma** rolü ise yalnızca okuyabilir — bulma, sayma, toplama gibi veriyi değiştirmeyen işlemler.

Her gelen komut, işlenmeden önce bu rol denetiminden geçer: kullanıcının rolünün, o komutu çalıştırmaya izni var mı? İzni yoksa, komut daha motora ulaşmadan reddedilir. Bu, on dördüncü bölümdeki en az ayrıcalık ilkesinin somut bir geçididir: her kullanıcı, yalnızca rolünün izin verdiği işlemleri yapabilir. OxiDB ayrıca, bir kullanıcının rolünü veritabanı düzeyinde geçersiz kılmaya da izin verir; böylece aynı kullanıcı, farklı veritabanlarında farklı yetkilere sahip olabilir.

Bu geçidin gerçekten işlediğine dair, bu kitap yazılırken yaşanan küçük ama öğretici bir örnek vardır. OxiDB'ye yeni bir komut — koleksiyonları belirli depolama seçenekleriyle oluşturan bir komut — eklendiğinde, onu yalnızca işleyicide tanımlamak yetmedi; komutun, rol denetim tablosuna da, hangi rollerin onu çalıştırabileceğini belirtecek biçimde eklenmesi gerekti. Aksi halde, komut var olsa bile, hiçbir rolün ona izni olmadığı için reddedilirdi. Bu, rol geçidinin dekoratif değil, gerçek bir kapı olduğunu — her komutun ondan açıkça geçmesi gerektiğini — gösterir.

24.5 Çok veritabanı: tek sunucuda yalıtılmış dünyalar

Rol geçidini anlatırken, bir kullanıcının yetkilerinin veritabanı düzeyinde değişebildiğine değindik. Bunun altında, sunucunun tek bir veritabanına değil, birçok yalıtılmış veritabanına aynı anda ev sahipliği yapabilmesi yatar. Bir **veritabanı yöneticisi** (database manager), veri dizininin altındaki her alt dizini ayrı, kendi

içine kapalı bir motor örneği olarak açar; her veritabanının kendi koleksiyonları, indeksleri, işlemleri ve dosyaları vardır ve biri diğerinden habersizdir. İki farklı veritabanında aynı adı taşıyan iki koleksiyon, birbirine hiç dokunmayan iki ayrı şeydir.

Bu düzenin birkaç sabit noktası vardır. Adı ayrıca belirtilmeyen istekler, `oxidb` adlı **varsayılan veritabanına** gider; bu ad, alışkanlıktan gelen bir `postgres` takma adıyla da anılabilir. Sunucunun tümüne ait olan — kimlik ve denetim gibi — veriler ise, herhangi bir kullanıcı veritabanının içinde değil, ağacın tepesinde ayrı tutulur; böylece bir veritabanını düşürmek, sunucunun kimlik bilgilerini yanına almaz.

Hattaki karşılığı da yalındır. Her istek, üzerinde çalışacağı veritabanını isteğe bağlı bir alanla belirtebilir; belirtmezse oturumun varsayılanı geçerli olur ve bu varsayılan, `use_db` ile değiştirilebilir — tıpkı ilişkisel bir kabukta bir veritabanına geçmek gibi. Veritabanlarını yönetmek için `create_database`, `drop_database` ve `list_databases` komutları vardır; aynı işlemler, SQL yüzünden `CREATE DATABASE`, `DROP DATABASE`, `SHOW DATABASES` ve `USE` deyimleriyle de yapılabilir. İki yüz de aynı niyete varır. Yetkilendirme burada da işler: kullanıcılara veritabanı düzeyinde roller verilebilir ya da geri alınabilir, böylece aynı kimlik bir veritabanında yalnızca okuyabilirken bir başkasında yazabilir. Yeni bir veritabanı yaratmak ya da düşürmek ise, doğası gereği, yalnızca **yönetici** rolüne açıktır; bunlar, veriyle çalışmanın değil, sunucuyu biçimlendirmenin işlemleridir.

24.6 Aktarım şifrelemesi ve denetim

On dördüncü bölümün diğer iki katmanı da sunucuda karşımıza çıkar. **Aktarım şifrelemesi**, sunucu bağlantısının şifreli bir kanala sarılmasıyla sağlanır; böylece hattı dinleyen biri yalnızca anlamsız baytlar görür ve istek ile yanıtın içeriği ağ üzerinde korunur.

Denetim ise, on dördüncü bölümde anlattığımız "kim, ne zaman, ne yaptı" kaydını tutar. OxiDB'de bu yetenek isteğe bağlıdır; açıldığında, işlemler bir denetim günlüğüne kaydedilir. On dördüncü bölümde, denetim kayıtlarının kendisinin de yönetilmesi gerektiğini — sınırsız büyümemeleri için döndürülmeleri gerektiğini — söylemiştik. OxiDB bunu olgun biçimde yapar: denetim günlüğü, boyuta göre, geçen zamana göre ya da takvim sınırlarına göre döndürülebilir ve döndürülen eski kayıtlar isteğe bağlı olarak sıkıştırılabilir. Böylece denetim, sistemin uzun süre çalışmasını engelleyen, kontrolsüz büyüyen bir yük olmaktan çıkar; on dördüncü bölümdeki "denetim kayıtları yönetilmeli" ilkesinin somut karşılığıdır bu.

24.7 Gözlemlenebilirlik: açıklama, yavaş sorgu ve ölçümler

Bir sunucu, yalnızca istekleri doğru işlemekle kalmaz; kendi davranışını dışarıya görünür de kılmalıdır. Çalışan bir sistemde "bu sorgu neden yavaş?" ya da "sunucu şu an ne kadar yükte?" sorularını yanıtlayamıyorsanız, onu işletmek karanlıkta yürümeye benzer. OxiDB, bu görünürlüğü üç ayrı pencereden sunar.

Birincisi, bir sorgunun **nasıl** çalıştırılacağını önceden açıklayan bir komuttur. Bir bulma, sayma ya da toplama isteğini bu komutla sararsanız, motor onu çalıştırmadan önce seçtiği planı — hangi stratejiyi kullanacağını, bir indekse binip binmeyeceğini, kaç belgeyi inceleyip kaçını döndürmeyi beklediğini, indeksin elemediği hangi süzgeçlerin sonradan uygulanacağını — döndürür; üstelik gerçek bir koşunun zamanlamasını da ekler. On dokuzuncu bölümde sorgu motorunun indeks yollarını tartışırken, bu aracın verdiği türden bir bakış tam da işe yarar: bir sorgunun beklediğiniz indeksi kullanıp kullanmadığını, tahminde bulunmak yerine, doğrudan görürsünüz.

İkincisi, yavaş sorguları kendiliğinden yakalayan bir **profilleyicidir**. Bir eşik süresi tanımlarsanız — "şu kadar milisaniyeden uzun süren her komutu kaydet" gibi — motor, bu eşiği aşan hat komutlarını ayrı bir profil koleksiyonuna işler. Bu koleksiyon, kendisini de bir yaşam-süresi (TTL) indeksiyle sınırlar; yani eski kayıtlar, belirli bir süre sonra kendiliğinden düşer ve profil verisi kontrolsüz büyümez. Böylece, hangi sorguların yavaş olduğunu ve ne sıklıkla yavaşladığını, üretim yükünü hiç durdurmadan, geriye dönük olarak inceleyebilirsiniz.

Üçüncüsü, sunucunun anlık durumunu yaygın izleme araçlarının anladığı bir biçimde dışarı veren bir **ölçüm ucudur**. HTTP arayüzü üzerindeki belirli bir yol, sunucunun sayaçlarını — istek sayıları, gecikmeler ve benzeri metrikleri — endüstride standart hale gelmiş bir metin biçiminde sunar; böylece yaygın bir izleme yığını, bu ucu hiçbir uyarlama yapmadan toplayıp grafiğe dökebilir. Dikkat çekici yanı, bu yeteneğin dışarıdan hiçbir bağımlılık getirmemesidir: ölçümler, sunucunun zaten tuttuğu sayaçlardan doğrudan üretilir.

24.8 İsteğin sunucudaki yolu

Bu parçaları bir araya getirip, bir isteğin sunucudaki yolunu izleyelim; bu, on beşinci bölümdeki yaşam döngüsünün sunucuya özgü ayrıntısıdır. İstek, ağ üzerinden çerçevelenmiş bir mesaj olarak gelir ve çözülür. Önce, henüz kimlik doğrulanmamışsa, el sıkışma ve kimlik doğrulama adımları tamamlanır. Sonra, komut **rol geçidinden** geçer: rolün bu komuta izni yoksa, istek burada reddedilir. Geçidi aşan istek, artık tanıdık yola girer: motora ulaşır, hedef koleksiyon bulunur ve istek — okuma ya da yazma — önceki bölümlerde anlattığımız mekanizmalarla işlenir. Sonuç, çerçevelenip istemciye geri gönderilir; eğer denetim açıksa, bu işlem günlüğe de kaydedilir.

Burada, on beşinci bölümde değindiğimiz birleştirici noktayı yeniden görürüz: sunucu, çekirdek motorun üzerine giydirilmiş bir ağ kabuğudur. Kimlik doğrulama, yetkilendirme, protokol, denetim — bunların hepsi, isteği asıl işleyen çekirdeğe ulaşmadan önceki ve sonraki katmanlardır. Çekirdek, gömülü kipte de sunucu kipinde de aynıdır; sunucu yalnızca ona ağ üzerinden, güvenli bir biçimde erişim sağlar. İlerideki bölümde göreceğimiz gibi, OxiDB'nin küme kipi bile, istekleri aynı işleyici yolundan geçirir; yalnızca araya replikasyon ve yönlendirme ekler.

24.9 Bu bölümün bıraktığı yer

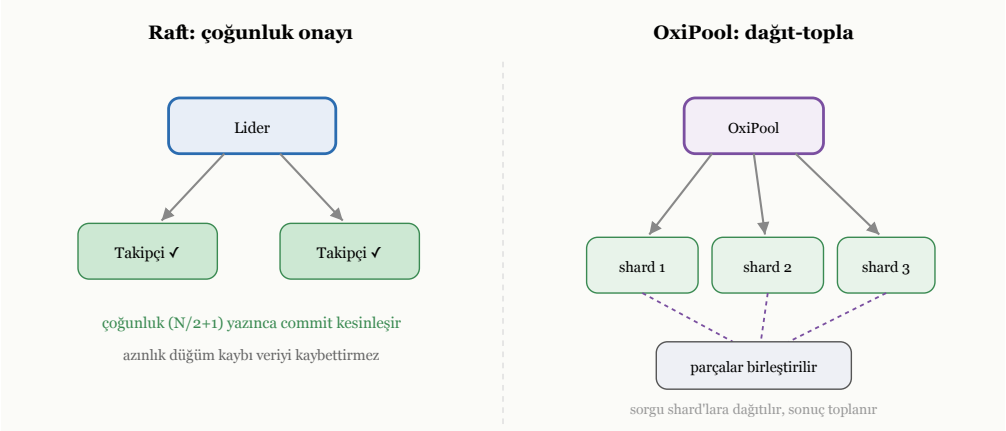
Bu bölümde, OxiDB'nin sunucu katmanını yakın plana aldık. Çerçeveleme sorununu ve OxiWire protokolünün uzunluk-önekli, JSON ya da ikili biçimli mesajlarını; bağlantı havuzunu ve el sıkışmayı; SCRAM tabanlı kimlik doğrulamayı ve parolanın hattan hiç geçmemesini; üç rollü yetkilendirme geçidini ve onun gerçek bir kapı olduğunu; tek sunucuda yalıtılmış birçok veritabanını barındıran çok-veritabanı düzenini; sorgu planını açıklayan komuttan yavaş-sorgu profilleyicisine ve ölçüm ucuna uzanan gözlemlenebilirlik pencerelerini; aktarım şifrelemesini ve döndürülebilen denetim günlüğünü; ve bir isteğin sunucudaki uçtan uca yolunu gördük. Tüm bunların, çekirdek motorun üzerine giydirilmiş bir güvenlik ve iletişim kabuğu olduğunu da gördük.

Şimdiye dek hep tek bir sunucu düğümünden söz ettik. Ama on ikinci bölümde öğrendiğimiz gibi, bir veritabanı tek makinenin sınırlarına dayandığında, birçok makineye yayılmak gerekir. Bir sonraki bölümde, OxiDB'nin ölçeklendirme katmanını — on ikinci bölümdeki konsensüsü hayata geçiren Raft tabanlı kümeyi ve sharding'i sağlayan yönlendiriciyi — somut olarak, hatta bu kitap yazılırken doğruladığımız davranışlarla birlikte ele alacağız.

Bölüm 25

OxiDB'de Ölçeklendirme: Raft Kümesi ve OxiPool ile Sharding

Şimdiye dek hep tek bir sunucu düğümünden söz ettik. Ama on ikinci bölümde öğrendiğimiz gibi, bir veritabanı tek makinenin sınırlarına dayandığında, birçok makineye yayılmak gerekir. On ikinci bölümde ölçeklendirmenin iki büyük tekniğini — aynı veriyi çoğaltan replikasyonu ve veriyi bölen sharding'i — genel ilkeler düzeyinde tanımıştık. Bu bölüm, bu iki tekniğin OxiDB'de nasıl hayata geçtiğini ele alıyor: replikasyon için Raft tabanlı kümeyi ve sharding için OxiPool adlı yönlendiriciyi. Bu bölümün ayrı bir değeri, anlatacağımız davranışların bir kısmının bu kitap yazılırken çok-düğümlü testlerle **doğrulanmış** olmasıdır; böylece on ikinci bölümün vaatlerini yalnızca tarif etmekle kalmayıp, gösterilmiş davranışlarla bağlayabiliyoruz.



Şekil 25.1: Raft çoğunluk onayı ve OxiPool dağıt-topla.

25.1 Raft kümesi: replikasyon ve konsensüs

On ikinci bölümde, güçlü tutarlı bir replikasyonun iki fikirden — tek bir otorite ve çoğunluk mutabakatı — beslendiğini görmüştük. OxiDB'nin küme kipi, tam da bunu yapan, **Raft** adlı anlaşılır olmaya özen göstererek tasarlanmış bir çoğunluk-konsensüs protokolüne dayanır.[1] Bir grup düğüm vardır; biri **lider** olur; tüm yazmalar liderden geçer ve bir yazma, düğümlerin **çoğunluğu** onu kalıcı kıldığında "tamamlanmış" sayılır.

[1] D. Ongaro ve J. Ousterhout, "In Search of an Understandable Consensus Algorithm," *Proc. USENIX ATC*, 2014.

Bir yazmanın küme içindeki yolu şöyledir. İsteği alan düğüm, onun bir yazma olduğunu tanır ve replikasyona uygun bir günlük girdisi hazırlayıp konsensüs grubuna önerir. Çoğunluk bu girdiyi kabul ettiğinde, girdi tamamlanmış sayılır ve her düğümde uygulanır. Burada zarif bir nokta vardır: girdiyi her düğümde işleyen mekanizma, yirmi dördüncü bölümdeki tek-düğüm işleyicinin **aynısıdır**. Yani motorun tüm davranışı — depolama, indeks, sorgu — kümede de birebir aynıdır; yalnızca yazmalar, uygulanmadan önce konsensüsten geçer. Küme, çekirdeğin üzerine eklenen bir replikasyon ve sıralama katmanıdır; çekirdeği değiştirmez.

Yirmi dördüncü bölümde tanıştığımız çok-veritabanlı düzen, kümeye de taşınır. Bir yazma, ait olduğu veritabanına **kapsanarak** konsensüse verilir; böylece aynı düğümler üzerinde birçok yalıtık veritabanı yaşayabilir ve her yazma yalnızca kendi veritabanında uygulanır. Veritabanı oluşturma ve düşürme gibi tanım değişiklikleri de replikasyona girer, öyle ki her düğüm aynı veritabanı kümesini görür; süre dolumu ve uyarı gibi arka plan iş parçacıkları her veritabanı için ayrı çalışır; ve bir işlem, başladığı veritabanına bağlı kalır.

Liderin çökmesi, on ikinci bölümde gördüğümüz failover sürecini tetikler: bir takipçi, çoğunluğun oyunu alarak yeni lider olur. Çoğunluğun büyüsü — herhangi iki çoğunluğun mutlaka kesişmesi — burada iki güvenceyi birden sağlar: aynı anda iki lider olamaz (split-brain önlenir) ve tamamlanmış hiçbir yazma kaybolmaz; çünkü yeni lideri seçen çoğunluk, o yazmayı bilen en az bir düğümü içerir.

25.2 Kümenin doğrulanması

On ikinci bölümün vaatleri, bu kitap yazılırken gerçek, çok-düğümlü bir test paketiyle sınandı; bu, kitabın "anlattığını gösterme" yaklaşımının iyi bir örneğidir. Çok düğümlü kümeler kurulup — senaryonun gerektirdiği düğüm sayısıyla — şu durumlar tek tek doğrulandı: tüm düğümlerin aynı lider üzerinde anlaşması; lidere yazılan belgelerin tüm düğümlere yayılması; liderin öldürülüp bir takipçinin devralması; failover sonrası verinin tutarlı kalması; düğümlerin azınlığının kaybedilmesine rağmen çalışmaya devam edilmesi; ve en kritik güvenlik özelliği — çoğunluğunu yitiren bir azınlığın yeni bir lider **seçememesi**, yani split-brain'in oluşmaması. Bu senaryoların hepsi beklendiği gibi çalıştı. Yani on ikinci bölümde soyut olarak anlattığımız konsensüs vaatleri — lider seçimi, replikasyon, otomatik failover ve çoğunluk güvenliği — OxiDB'de yalnızca tasarımda değil, gösterilmiş davranışta da geçerli.

Yirmi birinci bölümdeki işlemlerle de bir bağ vardır. Bir küme kipinde, tamamlanmış bir işlemin biriktirilmiş değişiklikleri, konsensüs katmanına **tek bir bütün** olarak verilir; böylece işlemin tüm değişiklikleri ya birlikte replikasyona girer ya da hiçbiri girmez. İşlemin "ya hep ya hiç" niteliği, tek düğümden kümeye taşındığında da korunur.

Bu doğrulama, tek bir noktada da durmadı; kümenin daha ince arıza kiplerine karşı da sınandı. Ağın simetrik olmayan biçimde bölündüğü — bir düğümün ötekini görüp ötekinin onu göremediği — durumlar zararsız kaldı; küme ya eski liderde birleşti ya da temiz bir seçimle yeni bir lider buldu. Seçim kararlılığını artırmak için kimi konsensüs uyarlamalarının başvurduğu bir ön-oylama adımının, OxiDB'nin bu kurulumunda gereksiz olduğu — onsuz da split-brain'in oluşmadığı — deneyle görüldü. Geriye dürüstçe açık kalan tek nokta, düğümlerin duvar saatlerinin birbirinden kayması durumudur; zamana dayalı kararların güvenilirliğini ilgilendiren bu konu, hâlâ dikkat isteyen bir açıktır.

Bu gücün bedeli, on ikinci bölümde söylediğimizdir: her yazma, çoğunluğa ulaşıp onların onayını beklediği için bir gidiş-dönüş gecikmesi öder. Güçlü tutarlılık, liderden geçen ve çoğunlukla mühürlenen bu yolla sağlanır.

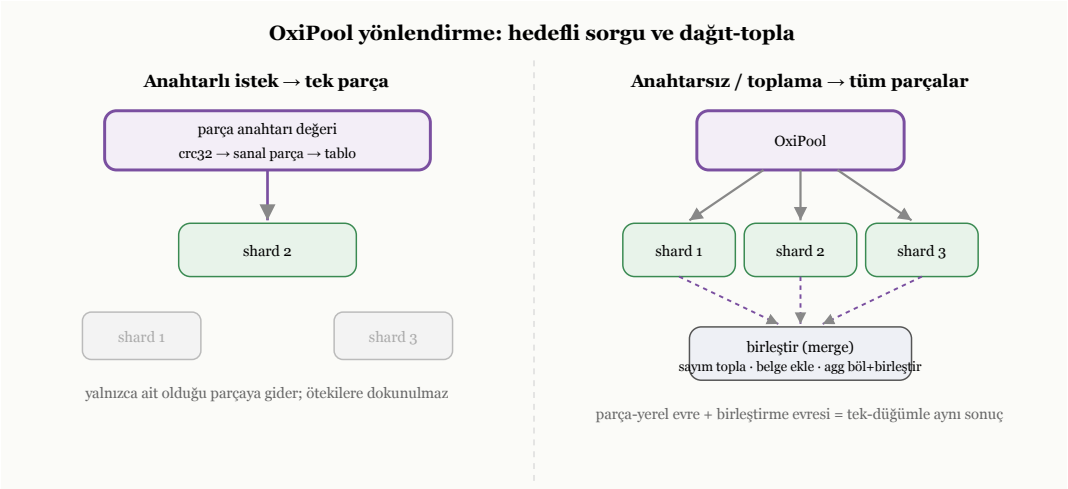
25.3 OxiPool: sharding ile veriyi bölmek

Raft kümesi, aynı veriyi çoğaltarak erişilebilirlik ve dayanıklılık sağlar; ama on ikinci bölümde gördüğümüz gibi, tek başına kapasite sorununu çözmez — her kopya yine tüm veriyi taşır. Veri tek bir makineye sığmaz hale geldiğinde, sharding gerekir. OxiDB bunu, bağımsız OxiDB parçalarının önünde duran **OxiPool** adlı

bir yönlendiriciyle sağlar. Her parça, normal bir OxiDB örneğidir — ki istenirse o parça kendi içinde bir Raft kümesi de olabilir; bu, on ikinci bölümün "shard'la, sonra her shard'ı replikasyonla çoğalt" topolojisidir.

OxiPool, on ikinci bölümdeki sharding mekaniğini doğrudan uygular. Bir **parça anahtarı** (shard key) belirlenir — örneğin "müşteri kimliği" gibi bir alan — ve anahtarlar, doğrudan parçalara değil, çok sayıda **sanal parçaya** (virtual chunk) eşlenir; sanal parçalar da parçalara dağıtılır. Eşleme şöyle işler: bir belgenin parça anahtarı alanının değeri alınır, hızlı bir sağlama toplamı işlevinden geçirilir ve çıkan sayının, sanal parça sayısına göre modülü alınarak bir sanal parça numarası elde edilir; küçük bir tablo da her sanal parçanın hangi gerçek parçaya ait olduğunu söyler. Sanal parça sayısının ikinin bir kuvveti seçilmesi, modül işleminin tek bir bit maskeleme adımına inmesini sağlar — yani çok ucuzdur. Bu iki kademeli dolaylama, on ikinci bölümde gördüğümüz yeniden dengeleme kolaylığını sağlar: bir parça eklendiğinde, anahtarları tek tek taşımak yerine, yalnızca birkaç sanal parçanın gerçek parçaya bağlanmasını değiştirmek yeterlidir; böylece taşınması gereken veri, toplam verinin yalnızca küçük bir oranıdır.

Yönlendirme şöyle işler. Parça anahtarını taşıyan bir istek — örneğin o anahtara göre bir sorgu ya da o anahtarı içeren bir belge ekleme — değer önce sanal parçaya, sonra gerçek parçaya çözülerek doğrudan o tek parçaya gönderilir; başka parçalara hiç dokunulmaz. Anahtarı içermeyen bir istek ya da bir toplama sorgusu ise, on ikinci bölümdeki **dağıt-topla** (scatter-gather) örüntüsünü tetikler: istek tüm parçalara koşut olarak gönderilir, her birinden kısmi yanıt alınır ve bunlar birleştirilir. Birleştirme, isteğin türüne göre değişir ve burada incelik vardır. Bir sayım, parçaların sayılarını toplar. Bir bulma, parçaların belge listelerini art arda ekler. Bir toplama (aggregate) ise en zorudur: gruplamayı saf biçimde her parçada ayrı yapıp sonuçları yan yana koymak yanlış olurdu — aynı grup birden çok parçada belirip iki kez sayılırdı. OxiPool bu yüzden, on ikinci bölümde anlattığımız gibi, toplama boru hattını ikiye böler: her parçada çalıştırılan bir **parça-yerel** evre ile, parçaların kısmi sonuçlarını tek bir doğru sonuçta birleştiren bir **birleştirme** evresi. Örneğin bir toplam, parçaların kısmi toplamlarının toplamına; bir ortalama, kısmi toplam ile sayının ayrı ayrı taşınıp sonda bölünmesine dönüşür. Sonuç, tek bir düğümde çalıştırılmış gibi birebir aynı çıkar; hem küresel hem de anahtar başına gruplamalar için.



Şekil 25.2: OxiPool: anahtarlı sorgunun hedefli yönü ve anahtarsız sorgunun dağıt-topla yolu.

Bu cross-shard toplama birleştirmesi de bu kitap yazılırken uçtan uca bir testle doğrulandı. Gerçek bir OxiPool yönlendiricisi, üç bağımsız parçanın önüne kuruldu; belgeler parça anahtarına göre parçalara dağıtıldı; ve bir sayımın parçalar boyunca toplandığı, bir bulmanın birleştirildiği, hem küresel hem de anahtar başına bir gruplamanın bölünüp birleştirilerek tek-düğüm beklentisiyle birebir eşleştiği, ve parça anahtarı taşıyan bir sorgunun tek bir parçaya yönlendirilip doğru sonucu verdiği doğrulandı. Yani on ikinci bölümün scatter-gather ve parçalar-arası toplama vaatleri de, gösterilmiş davranışla destekleniyor.

Burada, üçüncü ve on ikinci bölümlerde attığımız bir tohum meyve verir: kendi içinde bütün olan belgeler sharding'e doğal yatkındır. OxiPool'un belge başına yönlendirmesinin temiz çalışmasının nedeni budur —

her belge bağımsız olduğu için, hangi parçaya gideceğine kolayca karar verilir ve onu okumak için başka parçalara bakmak gerekmez.

25.4 Dürüst sınırlar

On ikinci bölümde, dağıtımın bedava olmadığını ve sharding'in olgun bir otomasyon gerektirdiğini söylemiştik. OxiDB bağlamında bu sınırları dürüstçe belirtmek gerekir. OxiDB'nin sharding'i, parçaları **otomatik dengeleyen** bir bileşene henüz sahip değildir; parçalar yapılandırılır, kendiliğinden yeniden dağıtılmaz. Cross-shard *toplama* vardır, ama parçalar arasında veriyi otomatik taşıyan bir denge mekanizması yoktur. Benzer biçimde, on birinci bölümde değindiğimiz ayarlanabilir tutarlılık düğmeleri — örneğin "en yakın takipçiden oku" gibi okuma tercihleri ya da işlem başına ince ayarlanan yazma güvenceleri — OxiDB'de henüz sınırlıdır; baskın yol, liderden geçen ve çoğunlukla mühürlenen güçlü tutarlı yoldur. Bunlar, sistemin olgunlaştıkça doldurabileceği boşluklardır; ama bir kitabın görevi, sistemin yapabildikleri kadar yapamadıklarını da dürüstçe göstermektir.

Bir dürüstlük notu da, ilerideki bölümlerde tanıtacağımız ilişkisel SQL motoru içindir. SQL yazmaları da kümede konsensüs üzerinden replikasyona girer: sunucu, gelen bir ifadeyi ayrıştırıp onun bir yazma mı yoksa salt-okuma mı olduğunu belirler; yazma ifadeleri, tıpkı belge yazmaları gibi çoğunlukla mühürlenir, salt-`SELECT` sorguları ve ayrıştırılamayan ifadeler ise düğüm-yerel çalışır. Buna karşılık SQL, sharding'e **girmez**: OxiPool bir isteği parça anahtarına göre yönlendirir, oysa bir SQL isteğinin böyle bir anahtarı yoktur; bu yüzden SQL, parçalara dağıtılmaz, tek bir arka uca gider — salt-okuma ise bir kopyaya, değilse lidere.

25.5 İkisini birleştirmek

On ikinci bölümde, replikasyon ile sharding'in birbirinin alternatifi değil, tamamlayıcısı olduğunu ve büyük sistemlerde birlikte kullanıldığını söylemiştik. OxiDB bu birleşimi destekler: veri, kapasite için OxiPool ile parçalara bölünür; ve her parça, erişilebilirlik için kendi içinde bir Raft kümesi olarak çoğaltılabilir. Böylece sistem hem tek makineye sığmayan veriyi taşır, hem de herhangi bir makinenin çökmesine dayanır. Bu, on ikinci bölümde anlattığımız standart büyük ölçek topolojisinin OxiDB'deki karşılığıdır.

Ama on ikinci bölümün kapanış dersini de tekrarlamak gerekir: dağıtım, kapasite ve erişilebilirlik kazandırır, ama koordinasyon, gecikme ve sorgu kısıtları getirir. Tek bir düğümde çalışan OxiDB, çoğu zaman bir kümeden ya da sharding'li bir kurulumdan çok daha basittir ve çok daha kolay akıl yürütülür. OxiDB'nin ölçeklendirme yeteneklerini, bir varsayılan değil, gerçekten ihtiyaç doğduğunda başvurulacak araçlar olarak görmek doğru olur.

25.6 Bu bölümün bıraktığı yer

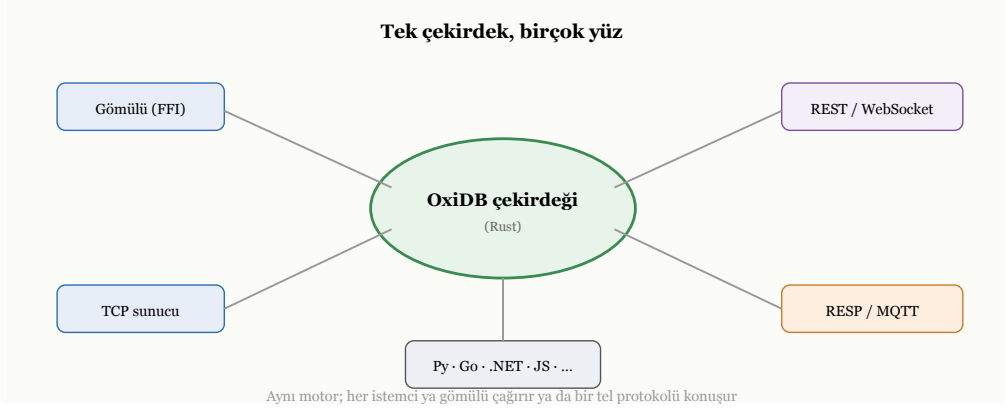
Bu bölümde, OxiDB'nin ölçeklendirme katmanını ele aldık. Raft tabanlı kümenin, on ikinci bölümdeki konsensüsü — lider seçimi, çoğunluk mutabakatı, otomatik failover ve split-brain güvenliğini — nasıl hayata geçirdiğini ve bunların çok-düğümlü testlerle nasıl doğrulandığını gördük. OxiPool'un, parça anahtarı, sanal parçalar ve dağıt-topla ile sharding'i nasıl sağladığını ve cross-shard toplamanın uçtan uca nasıl doğrulandığını izledik. Dürüst sınırları — otomatik dengeleyicinin ve ince tutarlılık ayarlarının henüz eksik olduğunu — ve iki tekniğin birleşik topolojisini gördük.

Buraya kadar OxiDB'yi hep motor ve sunucu tarafından, yani veriyi sağlayan taraftan tanıdık. Ama bir veritabanı, ona erişen uygulamalar kadar değerlidir. Bir sonraki bölümde, OxiDB'ye farklı programlama dillerinden nasıl erişildiğini — istemci kütüphanelerini, gömülü ve sunucu kiplerini ve aynı çekirdeğin nasıl birçok dilin önüne çıktığını — ele alacağız.

Bölüm 26

Uyumluluk Katmanları ve İstemciler

Önceki bölümün sonunda, bir veritabanının ona erişen uygulamalar kadar değerli olduğunu söylemiştik. Buraya kadar OxiDB'yi hep veriyi sağlayan taraftan — motor ve sunucu olarak — tanıdık. Bu bölüm, madalyonun öteki yüzüne, OxiDB'ye **nasıl erişildiğine** bakıyor: farklı programlama dillerinden kullanmayı sağlayan istemci kütüphanelerine ve OxiDB'yi başka ekosistemlerin protokolleriyle konuşturan uyumluluk katmanlarına. Bu bölüm, on beşinci bölümde tanıttığımız "tek çekirdek, çok yüz" felsefesinin en somut görünümüdür.



Şekil 26.1: Tek çekirdek, birçok istemci ve protokol yüzü.

26.1 Çekirdeğe iki kapı

Birinci bölümde, bir veritabanına iki uçtan erişilebileceğini söylemiştik: gömülü olarak ya da bir sunucu üzerinden. OxiDB'ye erişim de bu iki kapıdan birinden geçer. **Gömülü** kapıda, uygulamanız OxiDB ile aynı süreçte çalışır ve veriye doğrudan, bir ağ aradan geçmeden, işlev çağrılarıyla erişir; bu çok hızlıdır ve hiçbir sunucu kurulumu gerektirmez. **Sunucu** kapısında ise uygulamanız, yirmi dördüncü bölümdeki OxiWire protokolüyle ağ üzerinden bağlanır; bu, aynı veriye birçok uygulamanın aynı anda erişmesini sağlar.

İstemci kütüphanelerinin işi, bu iki kapı arasındaki farkı uygulamadan **gizlemektir**. Bir istemci kütüphanesi kullandığınızda, ister gömülü ister sunucu kipinde çalışın, karşınıza aynı kavramlar — koleksiyonlar, sorgular, indeksler, toplama, işlemler — aynı biçimde çıkar; çünkü hepsi, on beşinci bölümde söylediğimiz gibi, aynı çekirdek motorun farklı kapılarıdır.

26.2 Diğer dillere köprü: FFI

OxiDB'nin çekirdeği Rust dilinde yazılmıştır; peki başka dillerden nasıl kullanılır? Yanıt, **C uyumlu bir köprü** — yabancı işlev arayüzü (FFI, *foreign function interface*) — katmanındadır. Çekirdek, C dilinin çağırma kurallarına ve bellek düzenine uyan bir arayüz sunar; ve neredeyse her programlama dili, C ile konuşabildiği için, bu köprü sayesinde OxiDB'yi gömülü olarak çağırabilir. Bu köprünün neden C üzerinden kurulduğu öğreticidir: C, kırk yılı aşkın süredir işletim sistemlerinin ortak dilidir ve hemen her dil, C işlevlerini çağırmanın bir yolunu taşır — Python'da ctypes, Go'da cgo, .NET'te P/Invoke gibi. C, bir tür **evrensel ara katman** işlevi görür; her dilin her dile ayrı köprü yazması yerine, herkes ortak C zeminine konuşur.

Köprünün taşıdığı tek yük, dillerin **belleği farklı yönetmesidir**: Rust'ın sahiplik kuralları ile bir çağıran dilin çöp toplayıcısı birbirini tanımaz. Bu yüzden FFI sınırında bir disiplin gerekir — dizgeler ve tamponlar, bir tarafın ayırıp diğerinin serbest bıraktığı sızıntılara yol açmayacak biçimde, açık kurallarla devredilir. Bu disiplin bir kez doğru kurulduğunda, sonuç şudur: düşük seviyeli, performans-kritik bir çekirdek bir kez yazılır ve bu C köprüsü üzerinden birçok yüksek seviyeli dile açılır. Böylece OxiDB'nin gömülü hızı, yalnızca Rust'a değil, başka dillere de — ağ aradan geçmeden, doğrudan işlev çağrısıyla — sunulur.

26.3 İstemci kütüphaneleri

Bu köprünün ve sunucu protokolünün üzerine kurulu istemci kütüphaneleri, OxiDB'yi çeşitli programlama dillerinden — örneğin Python, Go, .NET ve JavaScript'ten — kullanılabilir kılar. Her istemci, ya gömülü köprüye doğrudan bağlanır, ya da sunucuya ağ üzerinden konuşur; bazı istemciler, tek bir arayüzün arkasında her iki kipi birden sunar, böylece uygulamanız gömülüden sunucuya geçerken kodunu değiştirmez.

Bu dağarcık, birkaç dilin ötesine geçecek kadar geniştir. Python, hem sunucuya ağ üzerinden bağlanan hafif bir istemciyle hem de az önce anlattığımız C köprüsü üzerinden gömülü bir sürümle gelir. Go ve JavaScript kendi bağımlılıksız istemcilerini taşır; JavaScript istemcisi, birazdan değineceğimiz web yüzeyinin gerçek zamanlı abonelik kanalını — bir sorguya abone olup değişiklikleri dinlemeyi — da sarmalar. .NET tarafı tek bir kütüphane değil, bir ailedir: ağ ve gömülü istemcilerin yanında, dile gömülü sorgu, standart bir veri erişim katmanı ve tam bir nesne-ilişki eşleyici sağlayıcısı. Julia, bilinçli bir tercihle yalnızca belge yüzünü sunar. Daha da ötede, PHP için bir istemci ve WordPress'in veritabanı katmanının yerine geçebilen bir eklenti; ve mobil için, aynı C köprüsü üzerinden iOS ve Swift'e uzanan bir bağ vardır. Hepsi, aynı çekirdeğe farklı dillerden açılan kapılardır.

İstemcilerin ortak özelliği, hepsinin aynı komut dağarcığını yansıtmasıdır; çünkü hepsi aynı çekirdeğe konuşur. Bu birörnekliğin somut bir örneği, bu kitap yazılırken yaşandı. OxiDB'ye yeni bir yetenek — koleksiyonları belirli depolama seçenekleriyle oluşturma — eklendiğinde, bu yetenek önce çekirdekte ve sunucu protokolünde tanımlandı; sonra istemcilerin her birine, bu komutu çağıran küçük bir sarmalayıcı eklendi. Python'da bir, Go'da bir, .NET'te bir, JavaScript'te bir. Hepsi aynı altta yatan komutu çağırıyordu; yalnızca her dilin kendi deyimine — Python'un adlandırılmış argümanlarına, Go'nun işlev-seçenek desenine — uygun bir biçim alıyordu. Bu, istemcilerin neden hepsinin aynı kavramları sunduğunu güzel gösterir: onlar bağımsız veritabanları değil, tek bir motorun farklı dillerdeki kapılarıdır.

26.4 Uyumluluk katmanları: başka protokolleri konuşmak

İstemci kütüphaneleri, uygulamaların OxiDB'nin kendi protokolüyle konuşmasını kolaylaştırır. Ama bazı uygulamalar, zaten **başka bir protokolü** konuşacak biçimde yazılmıştır ve OxiWire öğrenmek istemez. İşte burada OxiDB'nin uyumluluk katmanları devreye girer: OxiDB'yi, var olan araçların ve istemcilerin tanıdığı yüzlerle sunarak, onların az değişiklikle ya da hiç değişmeden çalışmasını sağlar.

Birinci uyumluluk yüzü, bir **bellek-içi anahtar-değer katmanıdır**. Bu katman, yaygın bir önbellek protokolüyle — RESP adıyla bilinen, satır tabanlı, basit bir istek-yanıt biçimiyle — uyumlu çalışır; böylece o

protokolü konuşan mevcut önbellek istemcileri ve komut satırı araçları, hiç değişmeden OxiDB'ye bağlanabilir. İkinci bölümde tanıdığımız anahtar-değer modelini hatırlayın; OxiDB, kendi belge motorunun üzerine, bu sade ve yaygın anahtar-değer yüzünü de giydirir. Katman yalnızca düz anahtar-değer ile sınırlı değildir: **sıralı kümeler** (her üyenin bir puanla sıralandığı yapılar) ve **yayınla-abone ol** (publish/subscribe) gibi, o ekosistemin tanıdık yeteneklerini de taşır. Burada dürüst bir nitelik gerekir: amaç, o önbellek sisteminin birebir kopyası olmak değil, en yaygın komutlarını desteklemektir; yani protokolün her inceliği değil, pratikte en çok kullanılan altkümesi karşılanır.

İkinci uyumluluk yüzü, bir **mesajlaşma protokolü** — MQTT'nin yaygın bir sürümü — desteğidir. Bu, nesnelerin interneti gibi, çok sayıda aygıtın yayınla-abone ol biçiminde, az bant genişliğiyle haberleştiği senaryolar içindir. En zarif yanı, bu mesajlaşma yüzünün, az önceki anahtar-değer katmanının yayınla-abone ol kanallarıyla **çapraz çalışmasıdır**: aynı kanal havuzunu paylaştıkları için, MQTT ile bir konuya yayınlanan bir mesaj, anahtar-değer protokolünden o konuya abone olan bir istemci tarafından dinlenebilir; tersi de geçerlidir. Böylece bir sıcaklık sensörü MQTT ile veri yayınlarken, bir gösterge paneli aynı veriyi önbellek protokolüyle dinleyebilir — ikisi arasında köprü kuran ayrı bir bileşene gerek kalmadan. Aynı mesajlaşma ailesinde, daha ağır kurumsal kuyruk sistemlerinin konuştuğu bir başka protokol — AMQP, yani RabbitMQ'nun dili — de desteklenir; böylece o ekosistem için yazılmış istemciler de, dayanıklı kuyruk güvenceleriyle, OxiDB'ye değişmeden bağlanabilir.

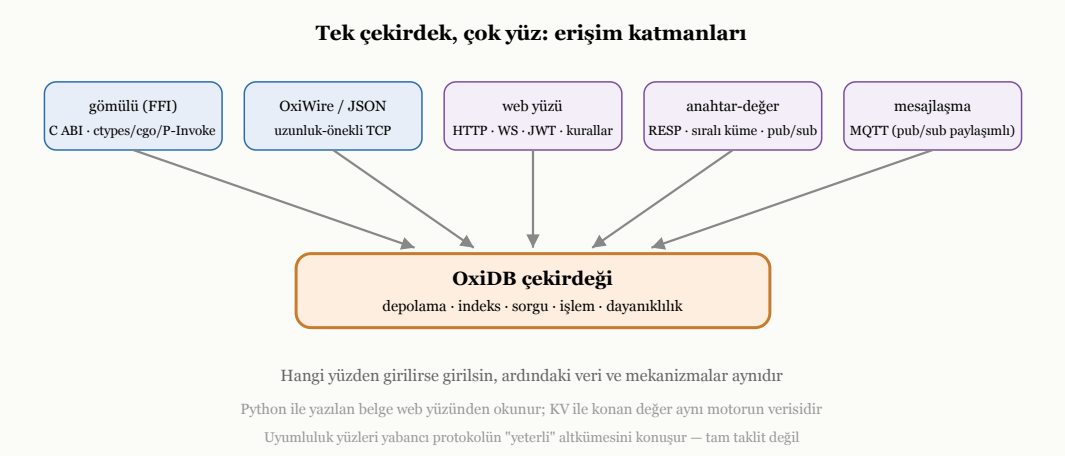
Üçüncü ve belki en geniş uyumluluk yüzü, **web yüzeyidir** ve dört parçadan oluşur. Birincisi, doğrudan bir **HTTP arayüzüdür**: belge ekleme, bulma, güncelleme, silme ve toplama işlemleri, tarayıcıların ve web araçlarının doğal dili olan istek-yanıt biçimiyle yapılabilir. İkincisi, **gerçek zamanlı bir abonelik kanalıdır** (WebSocket): bir istemci bir sorguya abone olur ve eşleşen belgeler her değiştiğinde sunucu ona bir değişiklik olayı iter; böylece istemci sürekli sormak (polling) zorunda kalmaz. Üçüncüsü, bir **kimlik sistemidir**: kayıt, giriş ve oturum doğrulama; parolalar, on dördüncü bölümün kuralına uygun olarak yavaş ve tuzlanmış bir özetle saklanır ve oturumlar, kendi içinde imzalı, durum tutmayan belirteçlerle (JWT) taşınır. Dördüncüsü, **belge düzeyinde güvenlik kurallarıdır**: "bir kullanıcı yalnızca sahibi olduğu belgeyi güncelleyebilir" gibi koşullar, ayrı bir kurallar koleksiyonunda tanımlanır ve her erişimde değerlendirilir.

Yirmi üçüncü ve on beşinci bölümlerde değindiğimiz bu Firebase benzeri yüz, web ve mobil uygulamaların, araya kendi yazdıkları bir sunucu katmanı koymadan, doğrudan OxiDB ile — gerçek zamanlı güncellemeler, kimlik doğrulama ve belge başına erişim kurallarıyla — çalışmasını mümkün kılar. JavaScript istemcisi, tam da bu web yüzeyi üzerinden konuşur ve bağımlılıksız olacak biçimde, hem tarayıcıda hem sunucu tarafı çalışma ortamında çalışır.

Bu üç yüz — anahtar-değer, mesajlaşma ve web — uyumluluğun en görünür örnekleridir; ama OxiDB'nin başka ekosistemlere uzanan yüzleri bunlarla sınırlı değildir. Belge motorunun kendisi, en yaygın belge veritabanının — MongoDB'nin — istek biçimini tanır: onun ekleme, bulma, güncelleme ve silme çağrılarını konuşan bir uyumluluk uyarlaması vardır ve bu uyum, MongoDB'nin kendi davranış testlerinin bir bölümünün OxiDB'ye karşı koşturulup geçmesiyle sınanmıştır. İlişkisel tarafta, sunucu kendini yaygın SQL istemcilerinin beklediği bir veritabanı adıyla da tanıtır; ve ilerideki bölümlerde tanıtacağımız SQL motoru, popüler bir nesne-ilişki eşleyicinin resmî uyumluluk testlerinin tamamını geçecek kadar eksiksiz konuşur. O motor ayrıca, sunucu içinde çalışan saklı yordamları iki dilde barındırır: doğrudan SQL metniyle yazılan gövdeler ve önceden derlenip küçük bir sanal makinede yürütülen bytecode yordamlar. Depolama tarafında, ilerideki bir bölümde ele alacağımız blob katmanı bir nesne-depolama (S3) HTTP arayüzü sunar; zaman-serisi motoru ise yaygın bir zaman-serisi veritabanının satır protokolünü anlar. Her biri kendi bölümünde incelenecek bu yüzlerin ortak fikri aynıdır: var olan araçları, yeniden yazmaya zorlamadan, olduğu gibi ağırlamak.

26.5 Tek çekirdek, çok yüz

Tüm bu erişim biçimlerinin — gömülü çağrı, OxiWire sunucusu, dil istemcileri, uyumluluk katmanları — ortak bir noktası vardır ve o, on beşinci bölümdeki felsefenin özüdür: hepsi aynı çekirdek motora oturur. Python istemcisiyle yazılan bir belge, web yüzeyinden okunabilir; anahtar-değer katmanıyla konan bir değer, aynı motorun verisidir. Hangi kapıdan girerseniz girin, ardındaki depolama, indeks, işlem ve dayanıklılık



Şekil 26.2: Tek çekirdeğin üstündeki erişim katmanları: gömülü FFI, OxiWire ve uyumluluk yüzleri.

mekanizmaları aynıdır. OxiDB'nin bu genişliği — tek bir motoru bu kadar çok yüzle sunması — onu klasik bir belge veritabanından ayıran belirgin özelliklerinden biridir.

Bu genişliğin bir bedeli olduğunu da dürüstçe söylemek gerekir. Her yüz, bakımı gereken bir yüzeydir; ve uyumluluk katmanları, çoğu zaman yabancı protokolün **tamamını** değil, "yeterli" bir altkümesini konuşur — yani o protokolün her inceliğini değil, en yaygın kullanımını destekler. Bu, makul bir ödünleşimdir: amaç, başka bir protokolün birebir kopyası olmak değil, o ekosistemin araçlarının çoğunun OxiDB ile çalışmasını sağlamaktır. Yine de, bir uyumluluk katmanını kullanırken, onun bir köprü olduğunu — hedef sistemin yerine geçen tam bir taklit değil — akılda tutmak gerekir.

26.6 Bu bölümün bıraktığı yer

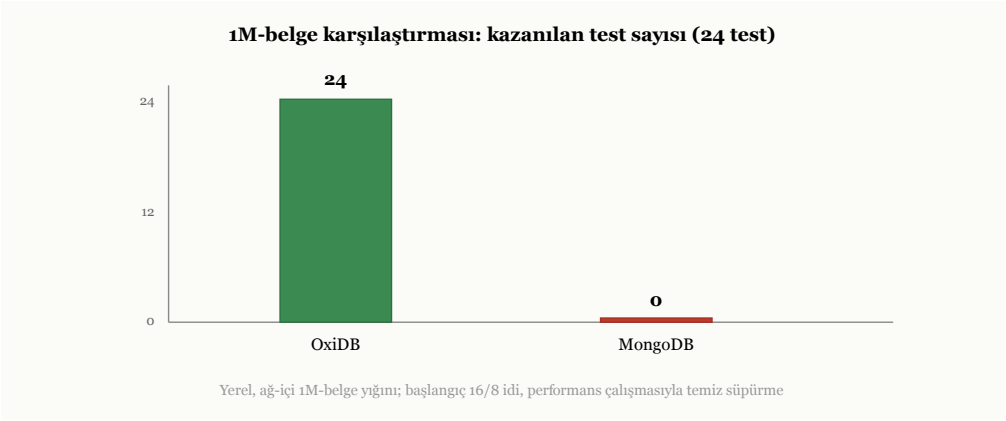
Bu bölümde, OxiDB'ye erişimin iki kapısını — gömülü ve sunucu — ve istemci kütüphanelerinin bu farkı nasıl gizlediğini gördük. Çekirdeği başka dillere açan C köprüsünü; istemcilerin neden hepsinin aynı kavramları sunduğunu ve bunun bu kitap yazılırken eklenen bir yetenekle nasıl somutlaştığını izledik. Anahtar-değer, mesajlaşma ve web olmak üzere üç uyumluluk yüzünü ve OxiDB'nin başka ekosistemlerin araçlarını nasıl ağırladığını gördük. Hepsinin aynı çekirdeğe oturduğunu — "tek çekirdek, çok yüz" — ve bu genişliğin dürüst bedelini de gördük.

Böylece, OxiDB'nin tüm katmanlarını — depolamadan dayanıklılığa, indeksten sorguya, işlemden ölçeklendirmeye, sunucudan istemcilere — dolaşmış olduk. Geriye, kitabı kapatmadan önce bir adım geri çekilip bütüne bakmak kaldı. Son bölümde, OxiDB'nin bu kitap boyunca birçok kez değindiğimiz bellek optimizasyonunu bir araya getirecek, onu MongoDB ile karşılaştıran ölçümleri dürüstçe değerlendirecek ve baştan beri izlediğimiz ilkelerin gerçek bir sistemde nasıl bir bütün oluşturduğu üzerine düşüneceğiz.

Bölüm 27

Bellek Optimizasyonu ve Karşılaştırmalı Değerlendirme

Kısım III boyunca, OxiDB'nin her katmanını tek tek dolaştık. Bu son bölüm, bir adım geri çekilip bütüne bakıyor. İki şeyi bir araya getiriyor: kitap boyunca birçok kez değindiğimiz bellek optimizasyonu hikâyesini ve OxiDB'yi olgun bir emsalle — MongoDB ile — karşılaştıran ölçümlerin dürüst bir değerlendirmesini. Amacımız bir galip ilan etmek değil; bu kitabın baştan beri savunduğu gibi, her sayının altındaki ödünleşimi görmek ve OxiDB'nin tercihlerini, onların gerçek sonuçlarıyla birlikte anlamaktır.



Şekil 27.1: 1M-belge karşılaştırmasında kazanılan test sayısı.

27.1 Belleği yerleşik yığından çıkarma yolculuğu

On üçüncü ve on altıncı bölümlerde, belleğe öncelikli bir veritabanının temel kısıtını görmüştük: yerleşik bellek, veriyle birlikte büyür. OxiDB'nin belleğe öncelikli kipi tam da böyledir ve küçük-orta ölçekte mükemmeldir; ama veri büyüdükçe bellek, hem pahalı hem de sınırlayıcı bir kısıt haline gelir. OxiDB'nin bellek optimizasyonu, bu kısıtı adım adım gevşetmenin — ve giderek disk-öncelikli kipi varsayılan yapmanın — hikâyesidir; bu kitap yazılırken izlediğimiz bir yoldur.

Yol, etki sırasıyla şu durakları içerir. Önce **önbellekler bir bütçeyle sınırlandı**: on üçüncü bölümde anlattığımız gibi, kontrolsüz büyüyen önbellekler, sabit bir bellek bütçesine bağlandı; bu, bellek kullanımını öngörülebilir kıldı. Sonra **belge gövdeleri bellekten çıkarıldı**: on altıncı bölümdeki disk-öncelikli kip, yerleşik bellekte yalnızca kompakt bir kimlik-konum dizini bırakarak, belge başına yüzlerce baytlık yükü

birkaç düzine bayta indirdi. Ardından **indeksler de çıkarıldı**: on sekizinci bölümde gördüğümüz belleğe yansıtılmış indeksler, indekslerin de yerleşik bellekten çıkmasını sağladı. Sonra **sorgu sırasındaki geçici bellek dizginlendi**: on dokuzuncu bölümdeki bayt düzeyinde süzme, büyük bir sonucun yüzlerce megabayt nesneye dönüşmesini önledi. Sıkıştırma, ölü alanı geri kazandı; ve sıkıştırmasız kip, tarama yükünü hafifletti.

Bu durakların birleşik sonucu çarpıcıdır. Disk-öncelikli kipte, beş yüz bin belgelik ve birkaç indeksli bir koleksiyonu taze açan bir süreç, yalnızca birkaç megabayt yerleşik bellekle açılır — oysa belleğe öncelikli yaklaşımda aynı veri yüzlerce megabayt tüketirdi. Hem belge gövdeleri hem de indeksler, yerleşik bellekten çıkmış, gerektiğinde diskten getirilen, geri alınabilir veriye dönüşmüştür. Bu yolculuk, on üçüncü bölümdeki "belleğe öncelikli kısıt" sorununun, bir dizi bilinçli mühendislik tercihiyle nasıl aşıldığının somut bir örneğidir.

27.2 Bellek ölçümünün dürüst yüzü

On üçüncü bölümde, "veritabanım ne kadar bellek kullanıyor" sorusunun aldatıcı derecede zor olduğunu söylemiştik; OxiDB'nin ölçümleri bunu canlı biçimde gösterir. Disk-öncelikli kipte taze açılışta bellek çok düşüktür — çünkü henüz yalnızca o küçük dizin bellektedir. Ama tüm veriye dokunan büyük bir taramadan sonra, bellek yükselir; çünkü işletim sistemi dokunulan sayfaları belleğe çeker. Bu yükselen bellek, on üçüncü bölümde anlattığımız gibi, geri alınabilir bir bellektir; baskı altında işletim sistemi onu serbest bırakır.

Bu yüzden iki sistemi yalnızca bir andaki bellek sayısıyla karşılaştırmak yanıltıcıdır. OxiDB, taze açılışta emsalinden kat kat az bellek kullanır; ama tüm veriyi baştan sona tarayan bir iş yükünden sonra, iki sistemin yerleşik belleği birbirine yakınsar — çünkü her ikisi de dokunulan veriyi belleğe çeker. Dürüst sonuç şudur: disk-öncelikli kipin bellek kazancı, çalışma kümesi tüm veriden küçük olan yükler için en büyüktür; tüm veriyi sürekli tarayan yükler için ise bu kazanç azalır. Bu, bir zayıflık değil, on üçüncü bölümde tarif ettiğimiz çalışma kümesi gerçeğinin doğal bir sonucudur.

Aynı dürüstlük, **kalıcı bellek tüketimi** için de gereklidir; çünkü buradaki tablo tek yönlü değildir. Bu kitap yazılırken alınan ölçümlerde, bir milyon belgelik bir yük işlendikten sonraki an incelendiğinde, OxiDB'nin yerleşik belleği emsalininkinden bir miktar **yüksekti**; buna karşılık diskte kapladığı yer daha **azdı**. İlk bakışta çelişkili gibi görünen bu, aslında bu kitabın anlattığı iki gerçeğin aynı anda çalışmasıdır: tüm veriye dokunan böyle bir yük, dokunulan sayfaları işletim sisteminin belleğe çekmesine yol açar — bu yüzden o anlık ölçümde bellek yüksekti, ama bu, baskı altında geri alınabilen bir bellektir — buna karşılık on altıncı bölümün kompakt, ekle-yalnız depolaması, diskte yer kazandırır. Yani "OxiDB daha az bellek kullanır" gibi tek cümlelik bir iddia, koşulları söylenmeden eksiktir: hangi kip, hangi iş yükü, ölçümün hangi anı? Bu sorular yanıtlanmadan, bir bellek sayısı tek başına bir şey söylemez.

27.3 Karşılaştırmanın felsefesi

Bir karşılaştırmalı değerlendirme, koşulları açıkça belirtilmedikçe, bir yalandan ibarettir. Bu yüzden ölçümlere geçmeden önce, onları nasıl okumak gerektiğini söyleyelim. Bu kitaptaki ölçümler, iki sistemi aynı makinede, aynı veriyle ve emsalin belleği sınırlanmış bir biçimiyle karşılaştırdı. Ama daha önemlisi, bir karşılaştırmanın amacı bir galip ilan etmek değildir; her sonucun altındaki **ödünleşimi** görmektir. Tek bir sayıyı seçip "şu sistem daha hızlı" demek kolaydır; zor ve dürüst olan, o sayının hangi tercihten doğduğunu söylemektir. Aşağıda, OxiDB'nin nerede kazandığını, nerede berabere kaldığını ve nerede kaybettiğini — ve her birinin **neden** öyle olduğunu — bu kitabın anlattığı mekanizmalara bağlayarak göreceğiz.

Sayısal sonucu da açıkça söyleyelim: bir milyon belgelik bu on sekiz senaryoluk karşılaştırmada, bu bölümün başındaki şekilde özetlendiği gibi, OxiDB — bugün varsayılan olan disk-öncelikli, sıkıştırmasız kipte koşturulduğunda — senaryoların çoğunu önde kapattı. Bu niteleme baştan gereklidir: sonuç, hangi kipte ölçüldüğüne bağlıdır. Belge gövdelerinin kayıt kayıt sıkıştırıldığı eski çerçeve, taramaları ve toplamaları belirgin biçimde yavaşlatıyordu; sıkıştırmanın varsayılan olarak kapatılması, tabloyu tam da bu yüzden değiştirdi.

Ama bu rakamı vurgulamamızın nedeni övünmek değil, tam tersine onu hemen ardından nitelemek: bu sonuç belirli koşullara — aynı makine, aynı veri, emsalin belleğinin sınırlandığı bir kurulum — bağlıdır ve asıl değeri, "kim kazandı" değil, "her bir sonucun altında hangi tercih yatıyor" sorusundadır.

Bu sonucun bir geçmişi olduğunu da söylemek, "anlattığını gösterme" sözüne sadık kalmak için gereklidir. Belleğe öncelikli varsayılanın egemen olduğu daha eski bir ölçümde, motor bu testlerin neredeyse tümünde öndeydi — ama bunu, tüm veriyi bellekte tutmanın yüksek bellek bedeliyle yapıyordu. Bu kitabın anlattığı bellek çalışmalarıyla varsayılan disk-öncelikli kipe geçildiğinde, terazi bir süre için bir miktar geri kaydı; çünkü artık veri bellekte değil, diskte, belleğe yansıtılmış haldeydi. Ardından birbirini izleyen birkaç bilinçli mühendislik adımı — disk-öncelikli kipin perdesini aralayan sıkıştırmasız seçenek, indeksli saymanın disk indekslerinde yeniden etkinleştirilmesi, bir bileşik indeksin kapsadığı toplamaların belgeye hiç dokunmadan yanıtlanması, sıralı-ilk-N yolunun neredeyse anlık hale getirilmesi, bayt düzeyinde süzme ve toplu boşaltmalı ekleme — geride kalınan testlerin çoğunu tek tek öne çevirdi. Yani bugünkü sonuç bir başlangıç hali değil, bu kitapta anlatılan ilkelerin ölçülebilir bir sonucudur; üstelik bu kez kazanç, düşük bellek ve dürüst dayanıklılık altında elde edilmiştir. Bir emsali ölçütle test etmenin asıl değeri de buradadır: nereye yatırım yapılacağını söyleyen bir pusula olması. Ölçütün ayrıca, ağ üzerinden değil, aynı ağ içinden — bağlantı maliyeti her iki taraf için de eşitlenecek biçimde — koşturulduğunu da belirtmek gerekir; çünkü tek bir belgeyi uzaktan ekleme gibi mikro işlemlerde, ölçülen sürenin büyük kısmı ağ gidiş-dönüşü olabilir ve bu, ölçülmek istenen şeyi gölgeler.

27.4 OxiDB nerede kazanır

OxiDB'nin belirgin biçimde öne geçtiği yerler, bu kitapta anlattığımız mekanizmaların doğrudan meyveleridir. İndeksli **tam eşleşme** sorgularında — belirli bir e-posta adresine sahip belgeyi bulmak gibi — OxiDB, emsalini kat kat geçer; bu, yedinci ve on sekizinci bölümlerdeki sıralı indekslerin gücüdür. **Saymada** üstünlük daha da belirgindir; çünkü on sekizinci ve dokuzuncu bölümlerde gördüğümüz gibi, OxiDB indeksli bir alana göre saymayı belgelere hiç dokunmadan, doğrudan indeksten yapar. **Sıralı ilk-N** sorgularında, sekizinci ve on dokuzuncu bölümlerdeki erken sonlanma sayesinde OxiDB öne çıkar. **İndeks kurmada**, toplu silmede ve belleğe öncelikli kipte eşzamanlı okumalarda da OxiDB güçlüdür. Bu kazançların hiçbiri sihir değildir; her biri, kitabın bir bölümünde anlattığımız somut bir tasarımın sonucudur.

27.5 OxiDB nerede berabere kalır

Bazı işlemlerde, iki sistem başa baş gider: eşitlik ve aralık sorguları gibi. Ama en öğretici beraberlik, **eklemededir**. OxiDB, toplu eklemede emsaliyle başa baş hız tutturur; ve bunu, altıncı ve on yedinci bölümlerde anlattığımız gibi, **her partiyi gerçekten diske boşaltarak** yapar. Emsali ise, varsayılan ayarında bu boşaltmayı yapmaz; yazmayı bellekten onaylar. Yani OxiDB, başa baş hıza, daha **güçlü** bir dayanıklılık güvencesiyle ulaşır. Bu, bir sonraki başlıkta döneceğimiz can alıcı dürüstlük noktasıdır.

27.6 OxiDB nerede kaybeder ve neden

Dürüst bir değerlendirme, kaybedilen yerleri ve nedenlerini de açıkça söylemek zorundadır. OxiDB'nin emsalinin gerisinde kaldığı her yer, bu kitabın anlattığı bir ödünleşime kadar izlenebilir.

Birincisi, **tek bir belgenin güncellenmesidir**. OxiDB, varsayılan katı dayanıklılık kipinde her commit'i gerçekten diske boşalttığı için, tek bir güncelleme bir tam boşaltma — milisaniyeler mertebesinde bir maliyet — öder. Emsali, varsayılan ayarında bu boşaltmayı yapmadığı için ilk bakışta daha hızlı görünür. Ama on yedinci bölümde gördüğümüz gibi, bu elma ile armuttur: OxiDB her tekil yazmayı gerçekten dayanıklı kılarken bir bedel öder, emsali o bedeli erteler. İki sistem aynı dayanıklılık modeline getirildiğinde bu fark kapanır: OxiDB'nin tek belge güncellemesi emsaliyle başa baştır. Dahası, **toplu güncellemede** OxiDB artık öne geçmiştir; çünkü on dokuzuncu bölümde anlattığımız bayt düzeyinde tekniğin bir uzantısı olarak,

toplu güncellemeler belgeyi baştan çözmek yerine ham ikili biçim üzerinde bir birinci-faz eşleştirme yapar ve bu, işi belirgin biçimde ucuzlatır. Yani buradaki "kayıp", bir performans yetersizliği değil, tekil ve katı bir yazmanın görünür maliyetidir; ölçek büyüdükçe ya da dayanıklılık eşitlendikçe silinir.

İkincisi, **disk-öncelikli kipte, büyük ölçekte eşzamanlı tekil okumalardır**. On üçüncü bölümde gördüğümüz gibi, çalışma kümesi belleği aştığında, rastgele tekil okumalar diske fault eder ve yavaşlar. Milyon belgelik bir veride, çok sayıda eşzamanlı rastgele okuma, disk-öncelikli kipte belirgin biçimde yavaştı; çünkü belge gövdeleri diskte, belleğe yansıtılmış halde duruyordu. Bu, disk-öncelikli tercihin beklenen bedelidir — bellek kazancının karşılığında, soğuk veriye erişimin gecikmesi.

Üçüncüsü, **disk-öncelikli, sıkıştırılmış kipte tüm koleksiyonu tarayan toplamalardı**. Yirminci ve on altıncı bölümlerde anlattığımız gibi, toplama tüm belgelere dokunur; sıkıştırılmış kipte her belgenin açılması gerekir ve bu maliyet birikir. Ama bu, kitap yazılırken giderildi: sıkıştırmasız kip her belgeyi açma yükünü kaldırdı; indeksli sayma kestirmesi disk indekslerinde yeniden etkinleştirildi; ve bir bileşik indeksin kapsadığı toplamalar artık belgelere hiç dokunmadan, yalnızca indeks yürüyüşüyle yanıtlanıyor — öyle ki toplama satırlarının hepsi OxiDB'nin lehine döndü. Yani bu kayıp, bir kez teşhis edilip mekanizmaya bağlandıktan sonra, doğru tercihlerle kapandı.

Bu üç kaybın ortak özelliği, hepsinin bu kitabın anlattığı bir ödünleşime dayanmasıdır: dayanıklılık-hız, bellek-gecikme, yer-işlemci. Hiçbiri açıklanamaz bir zayıflık değildir; her biri, bilinçli bir tercihin ölçülmüş sonucudur.

27.7 Karşılaştırmanın asıl dersi: dayanıklılık merceği

Bu değerlendirmenin en önemli dürüstlük noktası şudur: hız sayılarını, dayanıklılık güvencelerini karşılaştırmadan okumak yanıltıcıdır. İki veritabanını yalnızca "kaç işlem saniyede" diye karşılaştırmak, eğer biri her yazmayı diske boşaltıyor, diğeri bellekten onaylıyorsa, elma ile armuttur. OxiDB'nin varsayılanı, emsalinin varsayılanından daha güçlü bir dayanıklılık sunar; ekleme ve güncelleme sayıları, ancak bu mercekle okunduğunda anlam kazanır. Bir karşılaştırmayı dürüst kılan, sayıları vermek değil, o sayıların hangi güvenceler altında elde edildiğini söylemektir.

27.8 İlkelerin bir bütün oluşturması

Bu son bölüm, aslında tüm kitabın bir özetidir. OxiDB'nin her kazancı, her beraberliği ve her kaybı, bu kitabın ilk iki kısmında öğrendiğimiz bir ilkeye ve bir ödünleşime kadar izlenebilir. Sıralı indeksler tam eşleşmeyi ve saymayı hızlandırır; katı dayanıklılık tekil yazmaları yavaşlatır ama güvenceyi güçlendirir; disk-öncelikli kip belleği kurtarır ama soğuk okumayı yavaşlatır; sıkıştırma yer kazandırır ama tarama işlemcisini artırır. Karşılaştırmalı ölçümler, bu kitabın soyut ilkelerinin **ölçülebilir** hale gelmiş halidir.

İkinci bölümde "her model bir ödünleşimdir, gümüş kurşun yoktur" demiştik; bu son bölüm, aynı dersi sistem düzeyinde tekrarlar. OxiDB, evrensel ilkelerin somut bir cisimleşmesidir; ayırt edici tercihleri — tek çekirdek-çok yüz, belleğe öncelikli ve disk-öncelikli arasındaki bellek ayarı, belge odağı, bayt düzeyinde teknikler — sihir değil, bilinçli ödünleşimlerdir. Onun nerede parladığını ve nerede zorlandığını anlamak, bu ödünleşimleri görmekten geçer; ve bu kitap, baştan beri, o ödünleşimleri görmeyi öğretmeye çalıştı.

27.9 Bu bölümün ve ana metnin sonu

Bu bölümle birlikte, kitabın ana yolculuğunu tamamladık. Belge veritabanlarının ne olduğunu temelden kurduk; içeride nasıl çalıştıklarını katman katman açtık; ve tüm bu ilkelerin OxiDB adlı gerçek bir motorda nasıl hayata geçtiğini adım adım izledik. İkinci bölümde söylediğimiz gibi, ilk iki kısım size bu alanın sözlüğünü ve dilbilgisini öğretti; üçüncü kısım, o dille yazılmış gerçek bir metni birlikte okudu. Artık bir belge veritabanına

baktığınızda — ister OxiDB olsun ister başkası — onun verdiği her sözün altındaki mekanizmayı ve her hızın ardındaki ödünleşimi görebilecek bir bakışa sahipsiniz.

Kitabın geri kalanında, başvurmak isteyebileceğiniz iki ek bulacaksınız: kitap boyunca kullandığımız terimlerin bir sözlüğü ve daha derine inmek isteyenler için bir kaynaklar listesi. Ama asıl yolculuk burada tamamlandı. Bir veritabanının mütevazı vaadiyle — bir şeyi hatırlamak — başladık ve o vaadin altındaki zarif, çetin ve birbirine bağlı mekanizmaların tümünü dolaştık. Umarız, bir daha bir veritabanına "veriyi koy, sonra geri al" diye baktığınızda, o basit cümlenin altında ne kadar derin bir mühendisliğin yattığını hatırlarsınız.

Bölüm 28

OxiDB'nin İkinci Motoru: SQL

Buraya kadar okuduğunuz her şey tek bir motoru anlatıyordu: belgeleri saklayan, JSON tabanlı sorgularla süzen, toplama boru hattıyla dönüştüren belge motorunu. Üçüncü bölümde ilişkisel modelden belge modeline geçişin gerekçelerini tartışmış, ikisinin farklı sorulara farklı yanıtlar verdiğini söylemiştik. Şimdi o tartışmanın bıraktığı yerden devam ediyoruz: OxiDB, aynı süreç içinde, aynı portun ve aynı kimlik doğrulamanın arkasında, **ikinci bir motor** barındırır. Bu motor ilişkiseldir; ilişkisel bir katalog, ilişkisel bir satır deposu, kendi günlüğü ve kendi işlem yöneticisiyle gelir. Konuştuğu dil SQL'dir.

Bu bölüm o motoru anlatıyor: neden var olduğunu, belge motoruyla nasıl yan yana yaşadığını, hangi SQL yüzeyini desteklediğini, sorguları nasıl hızlandırdığını ve hangi istemcilerden konuşulabildiğini. İkinci kısımda kurduğumuz genel ilkeler — depolama, günlük, indeks, sorgu işleme, işlem — burada da geçerlidir; yalnızca somutlaştıkları biçim değişir.

28.1 Neden ikinci bir motor?

OxiDB'nin geçmişinde bir SQL yüzeyi vardı ve bilinçli olarak **kaldırıldı**. Nedeni öğreticidir: o yüzey gerçek bir motor değil, belge motorunun üzerine serpiştirilmiş bir **SQL lehçesiydi**. Aynı depolamayı, aynı sorgu ağacını, aynı işlem makinesini kullanıyordu. Sonuç, ikisinin de en kötü yanıydı: ilişkisel semantiği (sabit şema, tip zorlaması, birleştirme) ilişkisel olmayan bir depoya zorla giydirmek, hem SQL'i eksik hem belge motorunu karmaşık bırakıyordu.

İkinci girişimin çıkış noktası bambaşkaydı: **iki motor, tek örnek, farklı dosyalar**. SQL motoru belge veritabanının dosyalarına dokunmaz; belge motoru SQL tablolarını görmez. Paylaştıkları tek şey süreç, ağ portu, kimlik doğrulama ve dış yaşam döngüsüdür. Bir koleksiyon adı ile bir tablo adı asla çakışmaz, çünkü ikisi ayrı ad uzaylarında yaşar. Bu karar, projede ADR-0010 olarak kayıtlıdır.[1]

Bu ayrılığın pratik bir sonucu daha vardır: SQL motoru **varsayılan olarak kapalıdır**. Sunucu, `OXIDB_SQL` ortam değişkeni doğru bir değere ayarlanmadıkça motoru hiç kurmaz — ne dosya açar, ne bellek ayırır, ne iş parçacığı başlatır. Kullanmadığınız şeyin bedelini ödemezsiniz; on üçüncü bölümde savunduğumuz "isteğe bağlı yüzey" ilkesinin bir örneği daha.

```
# SQL motorunu açarak sunucuyu başlatın. Kapalıyken maliyeti sıfırdır.
export OXIDB_SQL=1
export OXIDB_DATA=./oxidb_data          # belge motorunun kökü
export OXIDB_SQL_DATA=./oxidb_data/sql  # SQL motorunun kendi dizini (varsayılan)
oxidb-server
```

[1] Mimari Karar Kaydı (Architecture Decision Record): bir tasarım kararının bağlamını, seçeneklerini ve gerekçesini kalıcılaştıran kısa belge. OxiDB'nin kararları `docs/decisions/` altında numaralandırılmıştır.

Sunucu açıldığında iki motor da aynı bağlantı üzerinden konuşulabilir. İstek üzerindeki `engine` alanı yolu belirler: alan yoksa ya da `"doc"` ise istek belge motoruna, `"sql"` ise SQL motoruna gider. Eski istemciler tek bir bayt bile değişiklik görmez.

```
{
  "engine": "sql",
  "cmd": "sql",
  "sql": "SELECT ad FROM musteriler WHERE id = $1",
  "params": [7]
}
```

Bu istek, uzunluk önekli JSON protokolünün (yirmi dördüncü bölüm) sıradan bir mesajıdır; yanıt da öyledir. Yetkilendirme tarafında `sql` komutu **ReadWrite** rolüyle korunur; salt-okunur `Read` rolüne sahip bir oturum yalnızca `SELECT` ve `SHOW` çalıştırabilir, veri değiştiren her ifade reddedilir.

Çok veritabanlı kurulumda (ADR-0012) SQL motoru da veritabanı başınadır: varsayılan veritabanının tabloları `${OXIDB_DATA}/sql` altında, adlandırılmış her veritabanınınkiler `${OXIDB_DATA}/<ad>/sql` altında yaşar. Yani iki veritabanının `siparisler` tablosu birbirinden habersizdir.

28.2 Mimari: katalog, satır deposu, günlük, denetim noktası

SQL motorunun iç yapısı, beşinci ve altıncı bölümlerde anlattığımız depolama ve dayanıklılık ilkelerinin ilişkisel bir yeniden yorumudur.

Katalog, şemanın kendisidir: tablolar, sütunlar, tipler, kısıtlar, indeks tanımları, görünümler ve saklı yordamlar. Belge motorunda şema yoktu — belgeler kendi yapılarını taşırdı. Burada şema merkezîdir, çünkü satırların ikili düzenini katalog belirler. Bir sütun tipi bilinmeden bir satırı kodlamak ya da çözmek mümkün değildir.

Satır deposu, tablo başına bir `.rdat` dosyasıdır: sabit şemalı, tipli hücrelerden oluşan satırların ikili gösterimi. Desteklenen tipler `INT`, `DOUBLE`, `TEXT`, `BOOL`, `TIMESTAMP`, `BLOB` ve `DECIMAL`'dır. Zaman damgaları, belge motoruyla aynı sözleşmeyi izler: epoch milisaniye olarak `i64`. `DECIMAL` ise onluk tabanda **kesin** sabit noktalı aritmetiktir — para hesaplarının kayan noktaya emanet edilmemesi gerektiği için vardır.

Günlük (WAL), SQL motorunun kendi yazma-öncesi günlüğüdür. Altıncı bölümde gördüğümüz sözleşme aynen geçerlidir: bir değişiklik, veri dosyasına yansımadan önce günlüğe dayanıklı biçimde yazılır; çökme sonrasında açılış, son denetim noktasındaki anlık görüntüyü yükleyip günlüğün kuyruğunu yeniden oynatarak durumu kurtarır.

Denetim noktası (checkpoint), günlüğü `.rdat` anlık görüntülerine katlayıp kısaltır. Varsayılan olarak günlük 64 MiB'ı aştığında otomatik tetiklenir; `OXIDB_SQL_CHECKPOINT_BYTES` ile ayarlanır, `0` verilirse yalnızca elle çağrılır.

Son olarak **disk-öncelikli mod** vardır. Varsayılanda motor tüm satırları bellekte tutar; `OXIDB_SQL_DISK_FIRST` açıldığında ise satırların gövdesi son denetim noktasının bellek eşlemli (`mmap`) anlık görüntüsünden okunur ve yalnızca denetim noktasından sonraki değişiklikler bellekte bir örtü olarak durur. On üçüncü bölümdeki bellek-disk ödünleşimi burada tek bir ortam değişkenine iner: hız mı, ayak izi mi. İki mod aynı dosya biçimini paylaşır, yani bir veritabanı istediğiniz modda yeniden açılabilir.

28.3 SQL yüzeyi: şemadan sorguya

En baştan başlayalım. Tablo yaratmak, sütun tipleri, birincil anahtar ve otomatik artan kimlik — hepsi tanıdık biçimdedir.

```sql
-- Şema: birincil anahtar, otomatik artan kimlik, NOT NULL, zaman damgası.
CREATE TABLE musteriler (
  id     INT PRIMARY KEY AUTO_INCREMENT,
  ad     TEXT NOT NULL,
  sehir  TEXT,
  puan   INT,
  kayit  TIMESTAMP
);

CREATE TABLE siparisler (
  id       INT PRIMARY KEY AUTO_INCREMENT,
  musteri  INT,
  tutar    DECIMAL(12, 2),   -- para: kesin onluk aritmetik
  durum    INT,
  olusma   TIMESTAMP
);
```

Katalog, sorgulanabilir. Şemayı keşfetmek için ayrı bir araca gerek yoktur; bu yüzeyi ileride EF Core'un iskele çıkarma (scaffolding) desteği de kullanacak.

```sql
SHOW TABLES;                  -- tablo adları
SHOW VIEWS;                   -- görünümler
SHOW INDEXES FROM siparisler; -- bir tablonun indeksleri
DESCRIBE musteriler;          -- sütunlar, tipler, null'lanabilirlik
```

Veri değiştirme ifadeleri de standarttır. Tek bir `INSERT` birden çok satır alabilir ve — önemlisi — **atomiktir**: satırlardan biri birincil anahtarı ihlal ederse hiçbiri yazılmaz.

```sql
-- Çok satırlı ekleme; AUTO_INCREMENT sütunu listeye yazılmayabilir.
INSERT INTO musteriler (ad, sehir, puan) VALUES
  ('ali', 'ankara', 10),
  ('ayse', 'izmir', 25),
  ('veli', 'ankara', NULL);

UPDATE musteriler SET puan = puan + 5 WHERE sehir = 'ankara';
DELETE FROM musteriler WHERE puan IS NULL;
```

Sorgu tarafında bağlama parametreleri iki biçimi de kabul eder: soru işareti (`?`) soldan sağa sırayla, dolar-numara (`$1`, `$2`) konumsal olarak. Sekizinci bölümde değindiğimiz gibi, parametre bağlama yalnızca kolaylık değil, aynı zamanda enjeksiyona karşı yapısal savunmadır: değer hiçbir zaman metne gömülmez.

```sql
-- İki parametre biçimi de geçerlidir.
SELECT ad, puan FROM musteriler WHERE sehir = ? AND puan > ?;
SELECT ad, puan FROM musteriler WHERE sehir = $1 AND puan > $2;
```

`SELECT`, beklediğiniz her şeyi yapar: yıldız ya da açık sütun listesi, ifade projeksiyonu, takma adlar, projeksiyonda olmayan bir sütuna göre sıralama, `LIMIT` ve `OFFSET` ile sayfalama.

```sql
-- İfade projeksiyonu + takma ad + çıktı takma adına göre sıralama + sayfalama.
SELECT id * 100 + puan AS skor, ad
FROM musteriler
ORDER BY skor DESC
LIMIT 10 OFFSET 20;
```

28.4 Birleştirmeler ve gruplama

Birleştirme, ilişkisel modelin ayırt edici işlemidir; üçüncü bölümde belge modelinin bu işlemi gömme ve referansla nasıl takas ettiğini tartışmıştık. SQL motoru dört birleştirme türünü de destekler: `INNER`, `LEFT`, `RIGHT` ve `FULL` — artı `CROSS JOIN`. Kendisiyle birleştirme ve çok tablolu zincirler de çalışır. Yalnızca virgülle yazılan eski usul birleştirme (`FROM a, b`) bilinçli olarak reddedilir; niyet açıkça yazılmalıdır.

```
-- Siparişi olmayan müşteriler: LEFT JOIN + sağ tarafta NULL denetimi
-- (klasik "anti-join" deseni).
SELECT m.ad
FROM musteriler m
LEFT JOIN siparisler s ON m.id = s.musteri
WHERE s.id IS NULL;
```

Motor, eşitlik koşullu birleştirmeleri karma (hash) birleştirmeyle çalıştırır; çok sütunlu anahtarlar (`ON a.k1 = b.k1 AND a.k2 = b.k2`) ve eşitlik dışı artık koşullar (`... AND b.tutar > 150`) doğru biçimde ele alınır. `NULL` anahtarlar hiçbir zaman eşleşmez — bu, `LEFT JOIN` semantiğinin sessizce bozulabildiği klasik tuzaktır ve testlerle çivilenmiştir.

Gruplama ve `HAVING` de tam anlamıyla oradadır. Toplama işlevlerinin `NULL` davranışı standarda uyar: `COUNT(*)` satırları sayar, `COUNT(sütun)` `NULL` olmayanları; `SUM`, `MIN`, `MAX`, `AVG` `NULL`'ları yok sayar; boş küme üzerinde `COUNT` sıfır, diğerleri `NULL` verir ama yine de **tek bir satır** döner.

```
-- Şehir başına ciro, yalnızca ikiden çok siparişi olan şehirler.
SELECT m.sehir,
       COUNT(*)      AS siparis_sayisi,
       SUM(s.tutar)  AS ciro
FROM musteriler m
JOIN siparisler s ON m.id = s.musteri
GROUP BY m.sehir
HAVING COUNT(*) > 2
ORDER BY ciro DESC;
```

Alt sorgular üç bağlamda çalışır: skaler alt sorgu (`WHERE v = (SELECT MAX(v) ...)` — birden çok satır dönerse hata), `IN` alt sorgusu ve `EXISTS`. `EXISTS`/`NOT EXISTS`, dış sorgunun sütunlarına **korelasyonlu** olabilir; motor tek eşitlikli korelasyonları bir yarı-birleştirme kümesine dönüştürerek (yani dekorele ederek) her dış satır için alt sorguyu yeniden çalıştırmaktan kurtulur.

```
-- 100 birimden büyük siparişi olan müşteriler (korelasyonlu EXISTS).
SELECT m.ad
FROM musteriler m
WHERE EXISTS (
  SELECT 1 FROM siparisler s
  WHERE s.musteri = m.id AND s.tutar > 100
);
```

Korelasyon, motorun en çok emek verilen köşelerinden biridir: dış referanslar yalnızca bir seviye değil, iç içe geçmiş alt sorguların, türetilmiş tabloların ve `VALUES` listelerinin **herhangi bir derinliğinden** dışarıyı görebilir; ara kapsamlar adları doğru biçimde gölgeler.

28.5 Analitik yüzey

Basit sorgular her motorun yaptığı iştir; bir SQL motorunu ayıran şey analitik yüzeyidir. OxiDB'nin SQL motoru burada şaşırtıcı ölçüde geniştir.

Pencere işlevleri ile sıralama ve bölüm-içi toplama yapılır. Bir pencere, `PARTITION BY` ile bölünür ve `ORDER BY` ile çalışan (kümülatif) hale gelir.

```
-- Bölüm içi sıralama: eşitler RANK'te aynı, ROW_NUMBER'da farklı numara alır.
SELECT departman, puan,
       ROW_NUMBER() OVER (PARTITION BY departman ORDER BY puan) AS sira,
       RANK()       OVER (PARTITION BY departman ORDER BY puan) AS rutbe,
       DENSE_RANK() OVER (PARTITION BY departman ORDER BY puan) AS yogun_rutbe
FROM personel
ORDER BY departman, puan;

-- Bölümün tamamı üzerinde toplam ve kümülatif (çalışan) toplam:
SELECT g, v,
       SUM(v) OVER (PARTITION BY g)          AS bolum_toplami,
       SUM(v) OVER (PARTITION BY g ORDER BY v) AS kumulatif
FROM olcumler;
```

İki sınırı açıkça söylemek gerekir: pencere işlevleri yalnızca **projeksiyonda** kullanılabilir (`WHERE` içinde ya da toplama işlevleriyle aynı `SELECT`'te karıştırılarak değil) ve açık çerçeve tanımı (`ROWS BETWEEN ...`) desteklenmez. Pencere sonucunu süzmek istiyorsanız klasik çözüm işe yarar: bir görünüm ya da türetilmiş tablo üzerinden filtreleyin.

```
-- Pencere sonucunu süzmenin yolu: görünüm (ya da türetilmiş tablo).
CREATE VIEW siralanan AS
  SELECT v, ROW_NUMBER() OVER (ORDER BY v) AS rn FROM olcumler;

SELECT v FROM siralanan WHERE rn = 1;
```

`DISTINCT ON`, PostgreSQL'den tanıdık "argmax" kestirmesidir: bir anahtarın her değeri için, `ORDER BY`'ın ilk satırını tutar. Gruplamayla birleştiğinde, "her müşterinin en çok harcadığı kategori" gibi soruları tek ifadeye indirir.

```
-- Her müşteri için baskın kategori: önce (müşteri, kategori) toplamları,
-- sonra müşteri başına en yüksek toplamı taşıyan satır.
SELECT DISTINCT ON (musteri) musteri, kategori, SUM(harcama) AS toplam
FROM satislar
GROUP BY musteri, kategori
ORDER BY musteri, SUM(harcama) DESC;

-- Sıralı-küme toplaması: grubun en sık görülen değeri (eşitlikte küçük olan).
SELECT g, mode() WITHIN GROUP (ORDER BY kategori) AS en_sik
FROM olaylar GROUP BY g;
```

Küme işlemleri tam takımdır: `UNION`, `EXCEPT`, `INTERSECT` — her biri `ALL` çeşidiyle. `ALL` olmayan biçim tekilleştirir; `ALL` çanta (bag) semantiğini korur, yani kopyalar sayılır. Öncelik standarttır: `INTERSECT` daha sıkı bağlar.

```
-- Çanta semantiği: soldaki her kopyayı sağdaki bir kopya götürür.
SELECT x FROM a EXCEPT ALL SELECT x FROM b ORDER BY x;

-- INTERSECT, UNION'dan sıkı bağlar: a UNION (b INTERSECT c).
SELECT x FROM a UNION SELECT x FROM b INTERSECT SELECT x FROM c;
```

`WITH` (ortak tablo ifadeleri) sorguyu adlandırılmış parçalara böler. Özyinelemesiz `WITH`, ayrıştırma anında türetilmiş tabloya açılır; sütun adı listeleri (`t(a, b)`) desteklenir. Asıl ilginç olan **`WITH RECURSIVE`**'dir: bir hiyerarşiyi ya da grafı, sabit noktaya ulaşana dek yinelemeyle dolaşır. Yapı zorunludur: `çapa UNION [ALL] adım`, ve adım kolu kendi adına referans verir.

```
-- Sayı üreteci: çapa 1'i verir, adım kolu bir öncekinin üstüne ekler.
WITH RECURSIVE t(n) AS (
  SELECT 1
  UNION ALL
  SELECT n + 1 FROM t WHERE n < 10
)
SELECT count(*), sum(n) FROM t;   -- 10, 55
```

Aynı makine, gerçek veriyi de dolaşır. Bir organizasyon şemasında bir yöneticinin altındaki herkesi derinlikleriyle çıkarmak, tek bir ifadeye sığar.

```
-- Geçişli kapanış: CTO'nun altındaki herkes, derinlik bilgisiyle.
WITH RECURSIVE alt(id, ad, derinlik) AS (
  SELECT id, ad, 0 FROM personel WHERE id = 2          -- çapa: CTO'nun kendisi
  UNION ALL
  SELECT p.id, p.ad, a.derinlik + 1                    -- adım: bir kat aşağısı
  FROM personel p JOIN alt a ON p.yonetici = a.id
)
SELECT ad, derinlik FROM alt ORDER BY derinlik, ad;
```

Döngülü graflarda `UNION` (yani `ALL` olmayan biçim) sonlanmayı sağlar: daha önce üretilmiş satırlar elenir, dolayısıyla bir çevrim sonsuza dek dönmez. Buna karşın `UNION ALL` ile yazılmış, sonlanma koşulu olmayan bir özyineleme sonsuza kadar çalışmaz — motorun 1 milyon yineleme ve 10 milyon satır korkulukları devreye girip anlaşılır bir hata döndürür.[2]

```
-- Döngülü graf: UNION (tekilleştiren) yinelemeyi sonlandırır.
WITH RECURSIVE erisim(dugum) AS (
  SELECT 1
  UNION
  SELECT k.hedef FROM kenar k JOIN erisim e ON k.kaynak = e.dugum
)
SELECT dugum FROM erisim ORDER BY dugum;
```

`LATERAL` birleştirme, sağdaki türetilmiş tablonun soldaki satıra başvurmasına izin verir; yani "her grup için ilk N" sorusunu doğrudan yanıtlar. Bu, EF Core'un koleksiyon projeksiyonlarını çevirdiği şekildir de.

```
-- Her blog için en yüksek puanlı 2 yazı (grup başına top-N).
SELECT b.id, y.baslik
FROM bloglar b
```

[2] Sonsuza dek asılı kalan bir sorgu, hata veren bir sorgudan çok daha kötüdür: birincisi kaynağı yer ve teşhis edilmesi zordur; ikincisi kendini söyler.

```
JOIN LATERAL (
  SELECT baslik FROM yazilar
  WHERE yazilar.blog_id = b.id
  ORDER BY puan DESC LIMIT 2
) y ON TRUE
ORDER BY b.id, y.baslik;
```

`LEFT JOIN LATERAL` boş dönen gövdeleri `NULL` ile doldurur; `CROSS JOIN LATERAL`, `ON TRUE` ile aynı şeydir. `RIGHT`/`FULL` LATERAL ise reddedilir — anlamı belirsiz olduğu için.

Tarih/zaman yüzeyi de EF Core ihtiyaçlarının itmesiyle olgunlaştı: `NOW()`/`CURRENT_TIMESTAMP`, `EXTRACT(part FROM ts)` ve eşdeğeri `date_part('part', ts)`, `date_trunc('part', ts)` (UTC takvim aritmetiği, ISO haftaları, PostgreSQL'in hafta-günü numaralandırması), `INTERVAL` sabitleri ve zaman damgası aritmetiği.

```
-- Aylık kova + hafta günü; INTERVAL sabitleri milisaniyeye katlanır.
SELECT date_trunc('month', olusma) AS ay,
       COUNT(*)                    AS adet
FROM siparisler
WHERE olusma >= NOW() - INTERVAL '30 days'
GROUP BY date_trunc('month', olusma)
ORDER BY ay;
```

Bir sınırı burada da dürüstçe söyleyelim: `INTERVAL` yalnızca **sabit** birimler için (gün, saat, dakika, saniye, milisaniye) tanımlıdır; ay ve yıl gibi takvimsel birimler reddedilir, çünkü sabit bir milisaniye karşılıkları yoktur. Ay eklemek için takvim-doğru `add_months(ts, n)` işlevi vardır (ayın son günü taşarsa kırpar). Skaler işlev dağarcığı bunun ötesinde geniştir: `FLOOR`/`CEILING` (`DECIMAL` üzerinde kesin), `POWER`, `SQRT`, `%`/`MOD`, `POSITION`/`STRPOS`, `LPAD`/`RPAD`, `LEAST`/`GREATEST`, tam kayan nokta matematiği ve düzenli ifade eşleştirmesi (`regexp_like`).

28.6 İndeksler ve sorgu hızlandırma

Yedinci bölümde gördüğümüz indeks ödünleşimi — okumayı hızlandır, yazmayı biraz yavaşlat, yer harca — burada da aynen geçerlidir; yalnızca indeksin üzerinde durduğu şey artık bir belge alanı değil, bir tablo sütunudur.

En ucuz erişim yolu **birincil anahtar nokta aramasıdır**: motor, `PRIMARY KEY` için değerden satır kimliğine bir eşleme tutar, dolayısıyla `WHERE id = ?` taramaya değil doğrudan bir aramaya iner. **İkincil indeksler** açıkça yaratılır ve `INSERT`/`UPDATE`/`DELETE` boyunca otomatik bakılır: güncellenen bir satır indeksin bir kovasından diğerine taşınır, silinen satır düşürülür. İndeksler kataloğa yazıldığı için yeniden açılışta hayatta kalır — denetim noktası alınmamış olsa bile, günlük yeniden oynatılarak yeniden kurulurlar.

```
-- İkincil indeks: seçici eşitlik aramalarını taramadan kurtarır.
CREATE INDEX ix_siparis_musteri ON siparisler(musteri);
CREATE INDEX IF NOT EXISTS ix_musteri_sehir ON musteriler(sehir);

-- Artık bu sorgu tabloyu taramaz, indeksten adayları alır:
SELECT id, tutar FROM siparisler WHERE musteri = 4211;
```

Motorun sorgu hızlandırma tarafında dört ayrı mekanizma birlikte çalışır.

Birincisi, **işlem içi indeks kullanımıdır**. Uzun süre, bir işlemin (ya da saklı yordamın) içindeki okumalar indeksleri hiç danışmıyor, her seferinde tam tarama yapıyordu. Artık işlem, temel tablonun ikincil indeksini danışıp sonucu kendi tamponlanmış yazma örtüsüyle birleştirir. Etkisi çarpıcıdır: yaklaşık 1500 nokta araması yapan, 500 bin satırlık bir sahtekârlık taraması saklı yordamı **55 saniyeden 0,15 saniyeye** indi.

İkincisi, **indeks-iç-içe-döngü birleştirmesidir** (index-nested-loop join). Sol tarafı küçük olan bir birleştirme, sağdaki büyük tabloyu taramak yerine onun indeksini yoklar. Klasik "küçük ⋈ büyük" deseni — 759 satırlık bir seçim ile 500 bin satırlık işlem tablosu — böylece bildirimsel SQL'de saklı yordam hızına ulaşır.

Üçüncüsü, **akışlı tarama** (streamed scan) ve beraberindeki iki eniyileştirme: yüklem itmesi (predicate pushdown) — `WHERE` koşulu satırlar somutlaştırılmadan, ödünç alınmış satırlar üzerinde değerlendirilir — ve tarama-içi top-N: `ORDER BY ... LIMIT n` yalnızca n satırlık bir yığın tutarak taramayı gezer, tüm tabloyu sıralamaz. Bu eniyileştirmelerin altın kuralı **görünmez olmalarıdır**: sonuçlar, somutlaştırılmış tam bir taramanın vereceğiyle bayt bayt aynı olmalıdır. Örneğin `DISTINCT ... LIMIT 3`, "ilk 3 satırı al sonra tekilleştir" diye kısaltılamaz — tekilleştirme `LIMIT`'ten **önce** gelir. Dış birleştirmelerde de yüklem, `NULL` ile doldurulabilen tarafa itilemez; itilirse yukarıdaki anti-join deseni sessizce yanlış cevap verir.

Dördüncüsü, **birleştirme yeniden sıralamasıdır**: yazılış sırası ne olursa olsun, planlayıcı küçük tabloları öne alabilir. Dış birleştirmeler ve `LATERAL` söz konusu olduğunda yeniden sıralama yapılmaz, çünkü sıra semantiğin parçasıdır.

Bir uyarıyı da yedinci bölümden hatırlatalım: indeks her sorgunun devası değildir. Tablonun tamamını gezen bir toplama sorgusu indeksten yararlanmaz — PostgreSQL'de olduğu gibi burada da tüm satırlar okunur. İndeksler seçici erişimi ve küçük-büyük birleştirmeleri kurtarır; tam tarama işini ortadan kaldırmaz.

28.7 İşlemler

Onuncu bölümde işlemlerin ne vaat ettiğini tanımlamıştık: ya hep ya hiç, tutarlılık, yalıtım, dayanıklılık. SQL motoru kendi işlem yöneticisini taşır ve işlemler **oturum boyunca** açık kalabilir; yani `BEGIN` bir çağrıda, `COMMIT` başka bir çağrıda gelebilir. Sunucu, açık işlemin kimliğini oturumla birlikte park eder; bağlantı düşerse işlem güvenle geri alınır.

```
BEGIN;
UPDATE hesaplar SET bakiye = bakiye - 100 WHERE id = 1;
UPDATE hesaplar SET bakiye = bakiye + 100 WHERE id = 2;
-- Bu noktada, işlemin içinden okuyan biri yeni bakiyeleri görür
-- (kendi yazdığını okuma); dışarıdan otomatik-commit ile okuyan
-- biri hâlâ eski bakiyeleri görür (yalıtım).
COMMIT;
```

İşlem içinde **kayıt noktaları** (savepoint) vardır: bir alt bölümü, işlemin tamamını feda etmeden geri almanın yolu. Kayıt noktası bir işlem bağlamı gerektirir — açık işlem yokken `SAVEPOINT` çağırmak hata verir.

```
BEGIN;
INSERT INTO t VALUES (5, 'p');
SAVEPOINT a;
INSERT INTO t VALUES (6, 'q');
ROLLBACK TO SAVEPOINT a;   -- yalnızca 6 geri alınır
INSERT INTO t VALUES (7, 'r');
COMMIT;                    -- 5 ve 7 kalıcı olur
```

Birincil anahtar tekliği işlemlerin içinde de zorlanır: hem taahhüt edilmiş bir satırla çakışma, hem aynı işlemin iki taahhüt edilmemiş satırı arasındaki çakışma yakalanır. Silip aynı anahtarı yeniden eklemek ise geçerlidir.

28.8 Saklı yordamlar: iki dil

Saklı yordamlar (ADR-0014) iki dilde yazılabilir ve ikisi de aynı `CALL` ile çağrılır. Bir `CALL` **atomiktir**: üst seviyede örtük bir işlem açar, açık bir işlem varsa ona katılır. Gövdedeki herhangi bir ifade patlarsa, o ana dek yapılan her şey geri alınır.

Birinci dil **SQL metnidir**: gövde, adlandırılmış parametreleri olan bir DML/ SELECT toplu işidir. Hiçbir araç zinciri gerektirmez; her çağrıda yeniden ayrıştırılır. `CALL`'ın sonucu, gövdedeki **son** ifadenin sonucudur.

```
-- SQL gövdeli saklı yordam: para yatır ve yeni bakiyeyi döndür.
CREATE PROCEDURE yatir(kime TEXT, tutar DOUBLE) AS BEGIN
  UPDATE hesap SET bakiye = bakiye + tutar WHERE ad = kime;
  SELECT bakiye FROM hesap WHERE ad = kime;
END;

CALL yatir('ali', 25);          -- 125.0
CALL yatir($1, $2);             -- çağıranın parametreleri de kullanılabilir
```

Bir ayrıntı bilinçlidir ve belgelenmiştir: ifade konumunda, bir parametre adıyla aynı olan niteliksiz bir ad **parametreyi** ifade eder, sütunu değil. Sütuna ulaşmak isterseniz tablo adıyla nitelendirin (`hesap.ad`).

İkinci dil **Cobra**'dır: derlenmiş bir bayt kodu, motorun içindeki bir sanal makinede çalışır. Yordam gövdesi `.cobrac` dosyasının base64'üdür; dosya bir `run(db, ...)` işlevi tanımlar, `db.query`/`db.execute` çağrıları `CALL`'ın işlemine katılır, `print` çıktıları istemciye **bildirim** (notice) olarak döner, ve yürütme 100 milyon komutluk bir **yakıt** sınırıyla korunur — sonsuz döngü bir sunucuyu esir alamaz. Determinizm `CREATE` anında doğrulanır (eşzamansız çağrılar, dış içe aktarma, giriş/çıkış reddedilir), böylece bir `CALL` küme genelinde güvenle çoğaltılabilir.

```
# Cobra kaynağı (derlenip base64 olarak CREATE PROCEDURE'e gömülür).
# Sorgular, döngüde toplama, bir bildirim ve sözlük dönüşü.
def run(db)
    let rows = db.query("SELECT name, age FROM people ORDER BY age")
    let total = 0
    let n = 0
    let oldest = ""
    for r in rows
        total = total + r["age"]
        n = n + 1
        oldest = r["name"]
    end
    print("stats over", n, "rows")          # -> istemciye bildirim
    return {"count": n, "total": total, "oldest": oldest}
end
```

Cobra'nın `try`/`catch`'i, kısıt ihlali gibi hataları yakalanabilir kılar; yordam patlamak yerine bir yedek değer döndürebilir. Yordamlar birbirini çağırabilir (aynı işlemi paylaşırlar) ve özyineleme bir derinlik korkuluğuyla sınırlıdır.

```
-- Derlenmiş Cobra bayt kodu, base64 olarak gömülür.
CREATE PROCEDURE stats() LANGUAGE COBRA AS '<base64 .cobrac>';
CALL stats();
SHOW PROCEDURES;
```

28.9 İstemciler

En alt katman, gördüğümüz JSON isteğidir; her istemci kütüphanesi onun üzerine oturur. Yanıt, ifade başına bir sonuç taşır: `SELECT` için `{"columns": [...], "rows": [[...]]}`, DML için `{"affected": N}`, DDL için `{"ddl": true}`, işlem ifadeleri için `{"transaction": true}`.

Python istemcisinde bu tek bir metottur.

```python
from oxidb import OxiDb

db = OxiDb(host="127.0.0.1", port=4444, username="admin", password="...")

db.sql("CREATE TABLE users (id INT PRIMARY KEY, name TEXT)")
db.sql("INSERT INTO users VALUES (?, ?)", [1, "ada"])

[sonuc] = db.sql("SELECT name FROM users WHERE id = $1", [1])
print(sonuc["columns"])   # ['name']
print(sonuc["rows"])      # [['ada']]
```

JavaScript SDK'sında aynı yüzey `sql(sql, params)` olarak durur ve söz (promise) döndürür.

```javascript
import { OxiDb } from "oxidb";

const db = new OxiDb({ host: "127.0.0.1", port: 4444 });
await db.connect();

await db.sql("INSERT INTO users VALUES (?, ?)", [2, "grace"]);
const [r] = await db.sql("SELECT name FROM users ORDER BY id");
console.log(r.rows); // [['ada'], ['grace']]
```

.NET tarafında iş daha derindir. `OxiDb.Data`, gerçek bir **ADO.NET sağlayıcısıdır**: `OxiDbConnection`, `OxiDbCommand`, `DbDataReader`. Bu, Dapper gibi ADO.NET üstüne kurulu her şeyin doğrudan çalışması demektir.

```csharp
using Dapper;
using OxiDb.Data;

using var conn = new OxiDbConnection("Host=127.0.0.1;Port=4444");
conn.Open();

conn.Execute("""
    CREATE TABLE musteriler (
      id    INT PRIMARY KEY AUTO_INCREMENT,
      ad    TEXT NOT NULL,
      puan  INT
    )
    """);

// Dapper'ın adlandırılmış parametreleri, sağlayıcı tarafından bağlanır.
conn.Execute("INSERT INTO musteriler (ad, puan) VALUES (@Ad, @Puan)",
             new { Ad = "ali", Puan = 10 });

var iyiler = conn.Query<string>(
    "SELECT ad FROM musteriler WHERE puan > @Alt ORDER BY puan DESC",
    new { Alt = 5 });
```

28.10 EF Core: sağlayıcı, göç, iskele

Bir adım yukarısı, Entity Framework Core sağlayıcısıdır (ADR-0013). LINQ sorguları SQL'e çevrilir; `DateTime` üyeleri ve `AddX` metotları tarih işlevlerine, `Math.*` çağrıları skaler işlevlere, `Contains`/`StartsWith` metin işlevlerine, EF'in `CROSS`/`OUTER APPLY` yapıları `[LEFT] JOIN LATERAL`'a düşer.

```
public sealed class ShopContext(string cs) : DbContext
{
    public DbSet<Musteri> Musteriler => Set<Musteri>();
    public DbSet<Siparis> Siparisler => Set<Siparis>();

    protected override void OnConfiguring(DbContextOptionsBuilder options)
        => options.UseOxiDb(cs);   // OxiDB sağlayıcısı
}

// Sıradan LINQ; motorun tarafında gruplu bir SQL sorgusuna çevrilir.
var enIyiSehirler = db.Siparisler
    .Join(db.Musteriler, s => s.MusteriId, m => m.Id, (s, m) => new { m.Sehir, s.Tutar })
    .GroupBy(x => x.Sehir)
    .Select(g => new { Sehir = g.Key, Toplam = g.Sum(x => x.Tutar) })
    .OrderByDescending(x => x.Toplam)
    .Take(5)
    .ToList();
```

Sağlayıcı yalnızca sorgu çevirmez. **Göçler** (migrations) çalışır: gerçek bir `__EFMigrationsHistory` tablosuyla `Database.Migrate()`, sütun ekleme/silme/yeniden adlandırma, indeks yaratma/düşürme. **İskele çıkarma** (`dotnet ef dbcontext scaffold`) da çalışır; var olan bir şemadan sınıf üretmek için yukarıda gördüğümüz `SHOW TABLES`/`DESCRIBE`/`SHOW INDEXES` yüzeyini kullanır.

Bu iddiaların ölçüsü nedir? En sert ölçü, Microsoft'un kendi yayımladığı **resmî EF Core ilişkisel şartname testleridir**. OxiDB sağlayıcısı bu paketin on iki Northwind takımının tamamını — `Where`, `Select`, `Functions`, `Join`, `GroupBy`, `Include`, `Navigations`, `SetOperations` ve diğerleri — **3832 testin 3832'sinde** geçer. Geçmeyen bir avuç senaryo, testlerde açıkça belirtilmiş sınırlardır (örneğin `DECIMAL`'in çift duyarlıklı hassasiyeti) ve çoğu SQLite sağlayıcısının da atladığı testlerle örtüşür. Bu tür bir uyum, bir SQL motorunun "gerçek" olup olmadığının en dürüst sınavıdır; çünkü testleri siz yazmazsınız.

28.11 Performans

Sayılar bağlamsız anlamsızdır, o yüzden iki somut ölçümle bitirelim.

Birincisi, aynı EF Core sağlayıcısının PostgreSQL karşısındaki durumu. Aynı model, aynı LINQ sorguları, aynı makine: birincil anahtar nokta araması, indeksli yabancı anahtar süzmesi, `ORDER BY ... LIMIT 20`, yüklemli `COUNT`, süzülmüş `SUM`, projeksiyonlu süzme, metin `Contains`, tekil ekleme, 100'lük toplu ekleme, nokta güncelleme, nokta silme. Bu basit sorgu şekillerinin tamamında OxiDB öndedir. Bu sürpriz değildir: gömülü bir motorun ağ ve süreç sınırı maliyeti düşüktür ve satırların çoğu zaten bellektedir.

İkincisi, daha zorlu bir sınav. Üretimde çalışan, PostgreSQL için yazılmış bir müşteri segmentasyonu işlevi — `WITH`, `DISTINCT ON`, `mode() WITHIN GROUP`, `LEAST` içeren, 300 bin satır üzerinde çalışan bir sorgu — OxiDB'de **metni harfi harfine aynı kalarak** çalıştırılabildi; çıktı PostgreSQL 15'inkiyle birebir aynı, süre ise biraz daha kısaydı (~144 ms'ye karşı ~149 ms). Buradaki asıl haber hız değil, **taşınabilirliktir**: sorguyu yeniden yazmak gerekmedi.

Buna karşılık, dürüst olalım: analitik iş yükünün ağır ucunda — tam tablo taramalı büyük toplamalarda, karmaşık çok tablolu planlarda — olgun bir ilişkisel motorun on yıllarca biriktirdiği planlayıcı zekâsı hâlâ

öndedir. OxiDB'nin SQL motoru bir veri ambarı motoru değildir; gömülü ve sunucu iş yüklerinde, uygulama sorgularının büyük çoğunluğunu hızlı ve doğru biçimde karşılamak için tasarlandı.

28.12 Sınırlar ve kapanış

Kapanışta, bu motorun bugün nerede durduğunu açıkça yazalım. Takvimsel `INTERVAL` birimleri (ay, yıl) desteklenmez; pencere işlevlerinin açık çerçeveleri yoktur; küme bazlı yabancı anahtar zorlaması, tetikleyiciler ve maddileşmiş görünümler yoktur. SQL motoru v1'de düğüm-yereldir; belge motorunun küme yetenekleriyle aynı olgunlukta değildir. Ve iki motor **henüz ortak bir işlem paylaşmaz**: bir belgeyi ve bir tabloyu tek bir atomik işlemde güncellemek, ADR-0011 olarak önerilmiş ama henüz gerçeklenmemiş bir yoldur.

Ama tabloyu bütün olarak görmek gerekir. Aynı ikili, aynı port, aynı kimlik doğrulama; bir tarafta şemasız belgeler ve toplama boru hattı, öbür tarafta sabit şemalı tablolar, birleştirmeler, pencere işlevleri, özyinelemeli sorgular, saklı yordamlar ve resmî şartname testlerini geçen bir EF Core sağlayıcısı. Üçüncü bölümde "ilişkisel mi, belge mi?" diye sormuştuk. OxiDB'nin yanıtı, bu sorunun bir seçim olmak zorunda olmadığıdır — yeter ki iki motor birbirinin dosyalarına karışmasın.

Bölüm 29

Zaman Serisi Motoru: oxidb-tsdb

OxiDB'nin hikâyesini buraya kadar iki motorla anlattık: belgeleri saklayan çekirdek motor ve onun yanına, kendi dosyalarıyla, kendi tablolarıyla oturan ilişkisel SQL motoru. Bu bölüm üçüncü bir motoru tanıtıyor: **oxidb-tsdb**, yani zaman serisi motoru. Diğer ikisi gibi o da ayrı bir sandıkta yaşar — kendi dizini, kendi dosya biçimi, kendi sorgu yüzeyi vardır — ve varsayılan olarak **kapalıdır**; açmadığınız sürece tek bayt maliyeti yoktur.

Ama neden üçüncü bir motor? Bir belge veritabanı, sonuçta her şeyi saklayabilir; bir sıcaklık ölçümünü de bir JSON belgesi olarak yazabilirsiniz. Bu bölümün asıl savı şudur: **zaman serisi verisi, belge modelinin iyi olduğu her şeyde kötüdür** ve tam da bu yüzden onun için ayrı bir depolama biçimi kurmak, bir milyon noktayı yüz megabayt yerine üç yüz kilobayta indirir. Bu, mühendislikte nadiren karşılaştığımız türden bir kazanç — bir yüzde otuz değil, elli altı kat. Bölümün yıldızı da bu kazancı sağlayan şey olacak: **Gorilla sıkıştırması**.

29.1 Zaman serisi verisinin doğası

Bir zaman serisi noktası şuna benzer: "cpu ölçümü, host=a etiketiyle, usage=0.93, zaman damgası 1700000000000". Bir saniye sonra neredeyse aynısı gelir: "cpu, host=a, usage=0.94, 1700000001000". Bir saniye sonra yine. Bu akış günlerce, aylarca sürer.

Bu verinin üç ayırt edici özelliği vardır. Birincisi, **şekli hep aynıdır**: aynı ölçüm adı, aynı etiketler, aynı alan adları, sonsuza kadar. İkincisi, **zaman damgaları düzenlidir**: bir saniyelik bir toplayıcı, bir saniyelik aralıklarla yazar; aralar milisaniye düzeyinde sapsa da yapı bozulmaz. Üçüncüsü, **değerler yavaş değişir**: bir CPU kullanımı, bir sıcaklık, bir kuyruk uzunluğu, ardışık iki ölçümde tamamen farklı olmaz; çoğu zaman ya aynıdır ya da ondalık basamağın sonunda oynar. Üstelik veri neredeyse hep **ekleme-sadece**dir: geçmişteki bir noktayı güncellemek istisnadır; yenisi hep sonuna gelir. Ve sorgular tekil noktaları değil, **aralıkları ve özetleri** sorar: "son bir saatin dakikalık ortalaması", "bugünün p95 gecikmesi".

Şimdi bu veriyi belge modeline koyalım ve dördüncü bölümde tanıdığımız o esnek, şemasız yapının bedelini sayalım. Bir nokta, belge olarak yaklaşık şöyle durur:

```
{
  "_id": "0f3c...",
  "measurement": "cpu",
  "tags": { "host": "a", "region": "eu" },
  "fields": { "usage": 0.93 },
  "ts": 1700000000000
}
```

Bu belge, JSON olarak yüz bayt civarındadır. Beşinci ve on altıncı bölümlerde gördüğümüz depolama katmanı bunu bir kaydın içine yazar; her kaydın başında durum baytı ve uzunluk alanı vardır, WAL'a bir kopyası daha düşer, indeksler kimliği bir kez daha tutar. Bir milyon nokta, kabaca **yüz megabayt** eder. Oysa aynı milyon noktanın taşıdığı gerçek bilgi, iki sayıdır: bir zaman damgası (`i64`) ve bir değer (`f64`) — yani **16 bayt**, toplamda 16 MB. Geri kalan her şey — "measurement", "tags", "host", alan adları, süslü parantezler — **bir milyon kez tekrarlanan aynı sabittir**. Belge modeli, şeması değişebilen veriyi taşımak için her belgeye kendi şemasını iliştirir; zaman serisinde şema hiç değişmediği için bu, saf israftır.

Sütunsal depolama tam olarak bu israfı ortadan kaldırır. Beşinci bölümde satır (kayıt) yönelimli depolamayı anlatırken, bir kaydın tüm alanlarının yan yana durduğunu söylemiştik; sütunsal depolamada ise **aynı alanın ardışık değerleri** yan yana durur. Alan adı bir kez, akışın kimliğinde yazılır; değerler ise adsız, tek tip bir dizi olur. Aynı tipteki, birbirine benzeyen değerlerin yan yana gelmesi ise sıkıştırmanın rüyasıdır.

29.2 Seri kimliği: ölçüm × etiket kümesi × alan

oxidb-tsdb'nin veri modeli InfluxDB'nin modelidir. Bir **nokta** dört parçadan oluşur: bir **ölçüm** (measurement) adı, sıfır ya da daha çok **etiket** (tag — hepsi metin), bir ya da daha çok **alan** (field — asıl ölçülen değer) ve bir milisaniye **zaman damgası**.

Etiketlerle alanlar arasındaki fark, motorun tüm iç yapısını belirler. **Etiketler kimliktir**: hangi makinenin, hangi bölgenin, hangi sembolün verisi olduğunu söylerler; sorguda onlara göre filtreleyip gruplarsınız. **Alanlar veridir**: ölçülen sayının kendisi. Motor bu ayrımı radikal biçimde uygular:

> Bir seri = ölçüm × (sıralı) etiket kümesi × alan adı.

Yani `cpu,host=a,region=eu usage=0.9,temp=51` biçimindeki tek bir nokta, iki alan taşıdığı için **iki ayrı seriye** dağılır: `cpu/host=a,region=eu/usage` ve `cpu/host=a,region=eu/temp`. Her seri, kendi başına bir sütunsal akıştır: yalnızca `(zaman, değer)` ikilileri. Etiketler sıralı tutulur ki aynı etiket kümesi, hangi sırayla yazılırsa yazılsın, aynı seriye düşsün.

Bu ayrıştırmanın bedeli de vardır ve dürüst olalım: etiketlerden birine yüksek kardinaliteli bir değer (örneğin kullanıcı kimliği) koyarsanız, milyonlarca seri yaratırsınız; her seri kendi tamponunu ve bloklarını taşıdığı için bu, belleği şişirir. Zaman serisi dünyasının klasik günahıdır bu; etiketler **sınırlı** bir değer kümesinden gelmelidir.

29.3 Gorilla: bir noktayı iki bite indirmek

Sıkıştırmanın kalbi `gorilla.rs`'te durur ve adını, Facebook'un 2015'teki bellek içi zaman serisi veritabanından alır.[1] Fikir iki bacaklıdır: zaman damgalarını **delta'nın delta'sıyla**, değerleri ise **XOR** ile kodlamak.

29.3.1 Zaman damgaları: delta-of-delta

Bir saniyelik toplayıcının zaman damgalarını düşünün:

```
t:    1700000000000, 1700000001000, 1700000002000, 1700000003000
delta:                       1000,          1000,          1000
delta-of-delta:                 -,             0,             0
```

[1] T. Pelkonen ve ark., "Gorilla: A Fast, Scalable, In-Memory Time Series Database," *Proceedings of the VLDB Endowment* 8(12), 2015.

Ardışık farklar (delta) hep 1000'dir; farkların farkı (delta-of-delta) ise **sıfır**. Sıfırı kodlamak için tek bir bit yeter. Motor tam da bunu yapar: `dod == 0` ise akışa tek bir `0` biti yazılır. Sıfır değilse, değişken genişlikte kovalara düşer — küçük sapmalar `10` öneki + 7 bit, biraz büyükleri `110` + 9 bit, daha da büyükleri `1110` + 12 bit, geri kalan her şey `1111` + tam 64 bit. Yani düzenli bir akış noktayı **1 bite**, ufak seğirmeleri (bir toplayıcının 3 ms geç kalması gibi) 9 bite indirir; yalnızca gerçekten düzensiz sıçramalar tam bedeli öder.

29.3.2 Değerler: XOR

Kayan noktalı sayılar için hile daha zariftir. İki ardışık `f64` değerinin bit desenlerini XOR'larsınız:

- Değer **hiç değişmediyse** XOR sıfırdır → akışa tek bir `0` biti yazılır. Bir milyon kez aynı sıcaklığı yazarsanız, milyon nokta bir megabit bile etmez.
- Değiştiyse, XOR sonucunun **anlamlı penceresi** (baştaki sıfırlardan sonrası, sondaki sıfırlardan öncesi) saklanır. Benzer büyüklükteki sayılar üsteli paylaştığı için XOR'un yalnızca son birkaç biti yanar; pencere dardır.
- Üstelik ardışık değerlerin pencereleri de birbirine benzediği için, motor önceki pencereyi yeniden kullanabiliyorsa (`lead` ve `trail` en az öncekiler kadarsa) pencere tanımını **hiç yazmaz**: `1` (değişti) + `0` (aynı pencere) + anlamlı bitler.

Bir blok kendi kendini betimler: `[u32 sayı][i64 t0][f64 v0][bit akışı]` — yani 20 baytlık bir başlık ve ardından bit düzeyinde paketlenmiş gövde. `bits.rs` içindeki `BitWriter`/`BitReader` de zaten bunun için vardır: bayt sınırlarını umursamadan tek tek bit yazıp okumak.

29.3.3 Somut bit hesabı

Gerçekçi bir gösterge (gauge) metriğini alalım: saniyede bir örnek, değer yuvarlanmış olduğu için bir süre sabit kalıyor, sonra bir basamak oynuyor. Bir noktanın maliyeti:

Bileşen	En iyi hâl	Tipik hâl
Zaman damgası (dod = 0)	1 bit	1 bit
Değer değişmemiş (XOR = 0)	1 bit	—
Değer değişmiş, pencere yeniden kullanılıyor	—	1 + 1 + ~10 bit
Nokta başına	**2 bit**	**~13 bit**

Değerlerin dörtte üçünün sabit kaldığı gerçekçi bir akışta ortalama, nokta başına 2–3 bit civarında kalır; buna 1024 noktalık blok başına 20 baytlık başlığı eklerseniz (nokta başına 0,16 bit) tablo değişmez. Ölçülen sonuç: **~0,28 bayt/nokta**. Karşılaştırma tablosu, bir milyon nokta için:

Biçim	Nokta başına	1M nokta
Belge (JSON, etiketler her belgede)	~100 bayt	~100 MB
Ham ikili (`i64` + `f64`)	16 bayt	16 MB
Gorilla (oxidb-tsdb)	~0,28 bayt	**~280 KB**

Ham ikiliye göre yaklaşık **56 kat**, belge biçimine göre iki kat daha fazla. On üçüncü bölümde bellek/disk ödünleşimini anlatırken "çalışma kümesini belleğe sığdırmak" demiştik; sıkıştırma, o kümeyi büyütmenin en ucuz yoludur. Bir yıllık saniyelik metrik (31,5 milyon nokta), Gorilla ile bir seri başına ~9 MB'tır — belleğe rahatça sığar.

Motorun testleri bu iddiayı doğrudan sınar: `oxidb-tsdb/tests/e2e.rs` içindeki `compression_ratio_and_retention`, 100.000 noktalık gerçekçi bir gösterge yazar ve sıkıştırılmış boyutun ham boyutun sekizde birinden küçük olmasını (yani en az 8×) şart koşar; pratikte gelen oran çok daha yüksektir.

29.4 Bloklar, aktif tampon ve saklama

Bir seri iki parçadan oluşur: **mühürlenmiş bloklar** ve bir **aktif tampon**. Gelen nokta önce tampona düşer (basit bir `(i64, f64)` vektörü). Tampon eşiğe (varsayılan 1024 nokta) ulaşınca **mühürlenir**: içindekiler zamana göre sıralanır, Gorilla ile kodlanır ve `min_ts`, `max_ts`, `count` bilgileriyle birlikte bir bloğa dönüşür. Bloklar bir daha değişmez.

Bu tasarımın iki güzel sonucu vardır. Birincisi **sorgu budaması**: bir aralık sorgusu, `[start, end)` ile kesişmeyen blokları hiç açmaz — `max_ts < start` ya da `min_ts >= end` ise blok es geçilir, tek bir bit bile çözülmez. İkincisi **saklamanın (retention) bedavaya gelmesi**: süresi dolan veriyi silmek, nokta nokta bir yeniden yazma değil, **bütün bir bloğu listeden düşürmektir**. `max_ts` değeri kesim noktasından eskiyse blok tümüyle atılır; hiçbir şey çözülmez, hiçbir şey yeniden kodlanmaz. Zaman serisi motorlarının bloklu depolamayı sevmesinin asıl nedeni budur.

Bu bloklu yapı sıkıştırma oranıyla saklama inceliği arasında bir ödünleşim kurar: blok ne kadar büyükse sıkıştırma o kadar iyi, ama saklamanın ve budamanın çözünürlüğü o kadar kaba olur. `with_block_points` bu ayarı açar; varsayılan 1024 makul bir orta noktadır.

29.5 Kalıcılık: MANIFEST tek commit noktasıdır

Altıncı ve on yedinci bölümlerde WAL'ın temel sözleşmesini kurmuştuk: veriyi kalıcı yapıya işlemeden önce niyeti günlüğe yaz. TSDB motoru aynı sözleşmeyi, ama bloklu depolamaya uygun bir biçimde uygular. `persist.rs`, üç dosya tipiyle çalışır:

```
<dir>/MANIFEST            # {"generation": N} — yetkili nesil numarası
<dir>/blocks.<N>.tsb      # N. kontrol noktasındaki tam blok anlık görüntüsü
<dir>/wal.<N>.log         # N'den sonra yazılan noktalar
```

Her nokta önce WAL'a eklenir; sonra bellekteki seriye girer. **Kontrol noktası (checkpoint)** şu sırayla işler: aktif tamponlar mühürlenir → `blocks.<N+1>.tsb` yazılır ve fsync edilir → MANIFEST, geçici dosya + `rename` ile atomik olarak N+1'e çevrilir → yeni `wal.<N+1>` açılır ve N nesline ait dosyalar silinir.

Buradaki tüm dayanıklılık argümanı **tek bir rename**'in iki yanında saklıdır. Çökme rename'den **önce** olursa, MANIFEST hâlâ N der; kurtarma eski anlık görüntüyü ve onun WAL'ını okur — o WAL, kontrol noktasından sonraki her noktayı hâlâ içerdiği için hiçbir şey kaybolmaz. Çökme rename'den **sonra** olursa, MANIFEST N+1 der; kurtarma yeni anlık görüntüyü okur ve eski WAL'a hiç bakmaz — dolayısıyla hiçbir nokta **iki kez sayılmaz**. Arada bir üçüncü durum yoktur; çünkü rename atomiktir. Saklama da bedavaya kalıcı olur: düşürülen bloklar, bir sonraki anlık görüntüde zaten yoktur.

WAL 8 MiB'ı geçtiğinde motor kendiliğinden bir kontrol noktası alır; `checkpoint` komutuyla elle de tetikleyebilirsiniz.

29.6 Tipli alanlar

Alanlar tiplidir: **float**, **integer**, **boolean** ve **string**. Sayısal üçlü aynı Gorilla yolundan geçer — hepsi `f64` olarak saklanır (tamsayılar 2^53'e kadar tam, mantıksal değerler 0/1) — ama tip seri bazında hatırlanır ve sorgu sonucunda `"type"` alanıyla geri bildirilir. Metin alanları ise ayrı bir yolda, `(zaman, metin)` çiftleri olarak yaşar; onlarda anlamlı toplamalar `first`, `last`, `count` ve `distinct`tır.

29.7 Motoru açmak

Motor varsayılan olarak kapalıdır. İki ortam değişkeni yeter:

```
# TSDB motorunu aç; veri ${OXIDB_DATA}/tsdb altına yazılır.
export OXIDB_TSDB=1
export OXIDB_DATA=./oxidb_data
# İsteğe bağlı: varsayılan veritabanının TSDB dizinini başka yere al.
export OXIDB_TSDB_DATA=/var/lib/oxidb/tsdb
oxidb-server
```

SQL motoru gibi TSDB de **veritabanı başınadır**: `oxidb` varsayılan veritabanı `${OXIDB_DATA}/tsdb`, adlandırılmış `metrics` veritabanı ise `${OXIDB_DATA}/metrics/tsdb` altında yaşar.

Tel protokolü, yirmi dördüncü bölümdeki OxiWire'ın aynısıdır — uzunluk önekli JSON — yalnızca isteğe bir `engine` alanı eklenir. Tüm TSDB istekleri `cmd: "tsdb"` altında toplanır ve bir `op` alanı eylemi seçer: `write`, `write_lp`, `query`, `stats`, `retention`, `checkpoint`, `rollup_add`, `rollup_refresh`, `rollups`. RBAC açısından `tsdb` komutu ReadWrite rolüyle korunur; Read rolü yalnızca sorgu çalıştırabilir.

29.8 Yazmak

En basit hâl — tek bir nokta:

```
{
  "engine": "tsdb", "cmd": "tsdb", "op": "write",
  "points": [
    { "measurement": "cpu",
      "tags":   { "host": "a", "region": "eu" },
      "fields": { "usage": 0.93 },
      "ts": 1700000000000 }
  ]
}
```

Yanıt: `{"ok": true, "data": {"written": 1}}`.

Python istemcisinin henüz özel bir TSDB yardımcısı yok; ama alttaki istek işlevini kullanmak yeterlidir. Küçük bir sarmalayıcı, bölümün geri kalanındaki tüm örnekleri okunur kılar:

```
from oxidb import OxiDbClient

db = OxiDbClient(host="127.0.0.1", port=4444)

def tsdb(op, **kw):
    """TSDB motoruna bir istek gönder; hata olursa istisna fırlatır."""
    return db._checked({"engine": "tsdb", "cmd": "tsdb", "op": op, **kw})

tsdb("write", points=[{
    "measurement": "cpu",
    "tags": {"host": "a", "region": "eu"},
    "fields": {"usage": 0.93},
    "ts": 1700000000000,
}])
```

Toplu yazma, aynı isteğe birden çok nokta koymaktır — bir noktada birden çok alan da olabilir; her alan ayrı bir seriye gider. Alanların JSON tipi, saklama tipini belirler: mantıksal değer → boolean, tam sayı → integer, metin → string, geri kalan → float.

```
base = 1_700_000_000_000
points = []
for i in range(600):                      # 10 dakikalık 1 sn'lik örnekler
    points.append({
        "measurement": "cpu",
        "tags": {"host": "a", "region": "eu"},
        "fields": {
            "usage":  round(0.4 + 0.1 * (i % 7), 2),  # float
            "cores":  8,                              # integer
            "up":     True,                           # boolean
            "state":  "healthy",                      # string (metin yolu)
        },
        "ts": base + i * 1000,
    })
print(tsdb("write", points=points))       # {'written': 600}
```

29.8.1 InfluxDB satır protokolü

Metrik dünyasının ortak dili, InfluxDB'nin **satır protokolüdür** (line protocol). Telegraf'tan collectd'ye kadar pek çok toplayıcı bu biçimde konuşur; motor onu doğrudan yutar. Biçim:

```
ölçüm[,etiket=değer,...] alan=değer[,alan=değer,...] [zaman_damgası]
```

Alan değerlerinin tipi son ekle belirlenir: `1.5` float, `10i` integer, `t/true/f/false` boolean, `"..."` metin. Zaman damgası **milisaniyedir**; yoksa sunucunun o anki saati kullanılır. Gerçek bir satır kümesi:

```
cpu,host=a,region=eu usage=0.9,cores=8i,up=true 1700000000000
cpu,host=b,region=us usage=0.4,cores=16i,up=true 1700000000000
mem,host=a free=1024i 1700000000000
weather,location=us\ midwest temp=82
```

(Son satırda hem kaçışlı boşluk hem de zaman damgasının yokluğu var: "şimdi" kabul edilir.) Bunu tek bir istekle göndermek:

```
{
  "engine": "tsdb", "cmd": "tsdb", "op": "write_lp",
  "lp": "cpu,host=a usage=0.9 1700000000000\ncpu,host=b usage=0.4 1700000000000"
}
```

Python'dan, bir toplayıcının çıktısını olduğu gibi aktarmak:

```
lines = "\n".join(
    f"cpu,host={h} usage={u} {base + i*1000}"
    for i, (h, u) in enumerate([("a", 0.91), ("b", 0.42), ("a", 0.93)])
)
tsdb("write_lp", lp=lines)   # {'written': 3}
```

Aynı isteği ham TCP üzerinden, hiçbir istemci kütüphanesi olmadan da gönderebilirsiniz; protokol, küçük-endian 4 baytlık uzunluk öneki + JSON'dur:

```
import net from "node:net";

const sock = net.connect(4444, "127.0.0.1", () => {
  const body = Buffer.from(JSON.stringify({
    engine: "tsdb", cmd: "tsdb", op: "write_lp",
    lp: "cpu,host=a usage=0.97 1700000600000",
  }));
  const len = Buffer.alloc(4);
  len.writeUInt32LE(body.length);        // uzunluk öneki: küçük-endian u32
  sock.write(Buffer.concat([len, body]));
});
sock.on("data", (buf) => {
  console.log(buf.subarray(4).toString()); // {"ok":true,"data":{"written":1}}
  sock.end();
});
```

29.9 Sorgulamak

Bir sorgu şunları söyler: hangi ölçüm, hangi alan, hangi etiket filtreleri, hangi yarı-açık zaman aralığı `[start, end)`, isteğe bağlı olarak hangi genişlikte zaman kovaları (`interval`), hangi etiketlere göre gruplama (`group_by`) ve hangi toplama (`agg`).

En yalın hâli — bir saatlik pencerede tek bir ortalama:

```
{
  "engine": "tsdb", "cmd": "tsdb", "op": "query",
  "measurement": "cpu", "field": "usage",
  "tags": { "host": "a" },
  "start": 1700000000000, "end": 1700003600000,
  "agg": "mean"
}
```

Yanıtın gövdesi her zaman bir **grup listesidir**; her grubun etiketleri, alanın tipi ve noktaları vardır:

```
[
  { "tags": {}, "type": "float",
    "points": [ { "ts": 1700000000000, "value": 0.72 } ] }
]
```

`interval` verilmediğinde tüm aralık tek bir kovadır ve kovanın zaman damgası `start` olur. Verildiğinde ise **downsample** devreye girer: kovalar epoch'a hizalıdır (InfluxDB'deki gibi), yani 60.000 ms'lik kovalar tam dakika sınırlarına oturur.

```
{
  "engine": "tsdb", "cmd": "tsdb", "op": "query",
  "measurement": "cpu", "field": "usage",
  "start": 1700000000000, "end": 1700003600000,
  "interval": 600000,
  "agg": "mean"
}
```

Bu, bir saati **on dakikalık altı ortalamaya** indirir — bir grafiğe çizilecek şey tam olarak budur. Motorun `e2e.rs` testi de bu davranışı sabitler: bir saatlik saniyelik veri, `interval = 600000` ile tam altı nokta döndürür.

Etiketlere göre gruplamak, tek sorguda birden çok çizgi üretir:

```
res = tsdb("query",
           measurement="cpu", field="usage",
           start=base, end=base + 3_600_000,
           interval=60_000,          # dakikalık kovalar
           group_by=["host"],        # her host için ayrı çizgi
           agg="mean")
for series in res:
    host = series["tags"]["host"]
    print(host, series["type"], len(series["points"]), "kova")
```

Filtre (`tags`) ile gruplamayı (`group_by`) karıştırmamak gerekir: birincisi serileri **eler**, ikincisi çıktıyı **böler**. İkisi birlikte de kullanılabilir — "yalnızca eu bölgesi, ama host'a göre ayrı çizgiler" gibi.

29.9.1 Toplama türleri

Toplama	Anlamı
`mean` / `avg`	Kovadaki değerlerin ortalaması (varsayılan)
`sum`	Toplam
`min` / `max`	En küçük / en büyük
`count`	Nokta sayısı
`first` / `last`	Kovanın en eski / en yeni değeri
`distinct`	Kovadaki farklı değer sayısı
`rate`	Saniyedeki değişim: `(last − first) / süre_sn` — sayaçlar için
`percentile` (+ `p`)	`p`. yüzdelik; doğrusal aradeğerleme
`p95`, `p99`, `p50`…	`percentile` için kısayol

Yüzdelikler, gecikme ölçümlerinin dilidir; ortalama gecikme yalan söyler, p95 söylemez:

```
{
  "engine": "tsdb", "cmd": "tsdb", "op": "query",
  "measurement": "http", "field": "latency_ms",
  "start": 1700000000000, "end": 1700003600000,
  "interval": 60000,
  "agg": "p95"
}
```

Aynı sorgu, uzun yazımıyla `"agg": "percentile", "p": 99.9` biçiminde de yazılabilir. `rate` ise **sayaç** (monoton artan) serileri içindir: bir kovadaki ilk ve son değerin farkını, aradaki süreye böler — "saniyede kaç istek" sorusunun doğru cevabı budur:

```
# İstek sayacı: saniyede kaç istek (dakikalık kovalarla)
rps = tsdb("query",
           measurement="http", field="requests_total",
           start=base, end=base + 3_600_000,
           interval=60_000, agg="rate")
```

Metin alanları da aynı sorgu yüzeyinden geçer; motor alanın metin olduğunu kendi anlar ve `first/last/count/distinct` toplamalarını uygular:

```
# Bir saatte kaç farklı durum görüldü? ("healthy", "degraded", ...)
tsdb("query", measurement="cpu", field="state",
     start=base, end=base + 3_600_000, agg="distinct")
# -> [{"tags": {}, "type": "string", "points": [{"ts": ..., "value": 3.0}]}]
```

29.10 Saklama ve kontrol noktası

Saklama, tek bir kesim zamanıyla çalışır: bu andan eski **bloklar** düşürülür.

```
import time
cutoff = int(time.time() * 1000) - 7 * 86_400_000   # 7 günden eskisi gitsin
print(tsdb("retention", cutoff=cutoff))              # {'removed': 6_048_000}

print(tsdb("stats"))
# {'series': 42, 'points': 3_601_234, 'bytes': 1_012_448}
```

`stats` üç sayı verir: seri sayısı, nokta sayısı ve sıkıştırılmış bayt. Bu üçünün oranı, bölümün başındaki iddianın canlı kanıtıdır — noktaları bayta bölün, 0,3'e yakın bir sayı çıkacaktır (aktif tampondaki, henüz mühürlenmemiş noktalar ham 16 bayt sayıldığı için, taze yazılmış bir veritabanında oran biraz daha yüksek görünür; bir `checkpoint` sonrası gerçek değerine oturur).

```
tsdb("checkpoint")   # aktif tamponları mühürle, anlık görüntü al, WAL'ı döndür
```

29.11 Sürekli toplama: rollup

Bir yıllık saniyelik veriyi saklamak ucuzdur, ama her grafik çizişinde otuz bir milyon noktayı taramak değildir. Zaman serisi dünyasının standart çözümü **sürekli toplamadır** (continuous aggregate): ham veriyi bir kez tarayıp, kapanmış zaman kovalarının özetini kalıcı olarak yazmak; grafikler artık özeti okur.

oxidb-tsdb'de bir rollup kuralı şunu söyler: "`cpu` ölçümünün her sayısal serisini, 60.000 ms'lik kovalarda `mean`, `max` ve `count` olarak topla." Sonuç, türetilmiş bir ölçümdür: adı `<ölçüm>@<etiket>`, alanları `<alan>_<toplama>`.

```
{
  "engine": "tsdb", "cmd": "tsdb", "op": "rollup_add",
  "measurement": "cpu", "label": "1m", "interval": 60000,
  "aggs": ["mean", "max", "count"]
}
```

Bu kural, `cpu` ölçümündeki `usage` alanından `cpu@1m` ölçümünde `usage_mean`, `usage_max`, `usage_count` alanlarını üretir — kaynak serinin etiketlerini (host, region…) aynen taşıyarak. Kural `rollups.json`'a yazılır, yani yeniden başlatmayı atlatır.

Kuralı çalıştırmak ayrı bir adımdır:

```
tsdb("rollup_add", measurement="cpu", label="1m",
     interval=60_000, aggs=["mean", "max", "count"])

written = tsdb("rollup_refresh")     # "now" verilmezse sunucu saati
print(written)                       # {'written': 60}

print(tsdb("rollups"))
# [{'measurement': 'cpu', 'label': '1m', 'interval': 60000}]
```

`rollup_refresh` yalnızca **tümüyle kapanmış** kovaları işler; içinde bulunulan dakikanın yarısı henüz gelmemiş olabilir, onu yazmak yanlış olurdu. Artımlılığı ise bir **su işareti** (watermark) sağlar: her (seri, aralık) çifti için son materyalize edilen kova başlangıcı kalıcı olarak tutulur. Yeniden başlatmadan sonra yenileme çağırırsanız, motor kaldığı yerden devam eder — ve `rollup.rs` testinin sabitlediği gibi, aynı kovayı **iki kez yazmaz**. Bu, sürekli toplamanın en sinsi hatasıdır (çöküş sonrası çift sayım) ve su işareti tam olarak ona karşı vardır.

Özet artık sıradan bir ölçümdür; onu da normal bir sorguyla okursunuz:

```
tsdb("query", measurement="cpu@1m", field="usage_mean",
     start=base, end=base + 86_400_000,
     interval=3_600_000, group_by=["host"], agg="mean")
```

Buradaki incelik şu: `cpu@1m` üzerinde saatlik ortalama almak, dakikalık ortalamaların ortalamasıdır — kovalar eşit doluysa doğru, değilse hafifçe kaymıştır. Kesinlik gerekiyorsa `usage_sum` ve `usage_count` alanlarını da materyalize edip oranı kendiniz kurun. Zincirleme (1s → 1m → 1h) motorda kendi kendine olmaz; kuralları arka arkaya tanımlayıp sırayla yenilemek çağıranın işidir.

29.12 Gerçek kullanım: tick'ten muma

Kitabın yirminci bölümünde, belge motorunun toplama hattındaki `$ohlcv` aşamasıyla tick verisini muma çevirmiştik. TSDB motoru aynı işi depolama tarafından çözer: tick'ler ham seri, mumlar rollup'tır.

```
# 1) Borsa akışını satır protokolüyle akıt (fiyat + hacim).
lp = "\n".join(
    f"trades,symbol=BTCUSDT price={p},qty={q} {ts}"
    for ts, p, q in stream_of_trades()      # (epoch_ms, fiyat, miktar)
)
tsdb("write_lp", lp=lp)

# 2) Dakikalık mum kuralı: açılış/en yüksek/en düşük/kapanış = first/max/min/last
tsdb("rollup_add", measurement="trades", label="1m", interval=60_000,
     aggs=["first", "max", "min", "last", "count"])
tsdb("rollup_refresh")

# 3) Mumları oku: trades@1m ölçümünde price_first = açılış, price_last = kapanış
candles = tsdb("query", measurement="trades@1m", field="price_last",
               tags={"symbol": "BTCUSDT"},
               start=base, end=base + 86_400_000,
               interval=60_000, agg="last")
```

Hacim (`qty_count` yerine gerçek toplam) istiyorsanız kural listesine `sum` eklemeniz yeter: `qty_sum` alanı doğrudan mumun hacmi olur.

Sunucu metrikleri tarafında da kalıp aynıdır: Telegraf'ın satır protokolü akışını `write_lp` ile içeri alın, panoların çizdiği çözünürlükte (`1m`, `5m`) bir rollup tanımlayın, ham veriyi yedi günlük bir `retention` ile budayın, özeti aylarca tutun. Ham veri pahalı olan değil — pahalı olan onu **her sorguda yeniden taramaktır**.

29.13 Sınırlar ve motorun yeri

Dürüst bir kapanış için sınırları da yazalım. TSDB motoru **düğüm-yereldir**: Raft ile çoğaltılmaz. Etiket filtreleri yalnızca eşitliktir (regex ya da `OR` yoktur). Aralıklar milisaniye cinsindendir ve takvim birimleri (ay, yıl) yoktur — tıpkı belge motorunun `$densify` aşamasında olduğu gibi, sabit süreli birimlerle çalışılır. Metin alanları henüz sıkıştırılmaz; ham `(zaman, metin)` çiftleri olarak yaşarlar. Ve kardinalite disiplini çağıranın sorumluluğundadır.

Yine de, bu bölümün asıl dersi mimaridir: OxiDB, "her veri için tek bir depolama" demek yerine, verinin şekli radikal biçimde farklılaştığında **onun için ayrı bir motor** açtı. Belge motoru esnek şemalı veriyi, SQL motoru ilişkisel veriyi, TSDB motoru da düzenli, ekleme-sadece, sonsuz akan ölçümleri saklıyor — ve her biri, diğerlerinin dosyalarına dokunmadan, aynı sunucunun aynı bağlantısı üzerinden konuşuyor. On beşinci bölümde "tek çekirdek, çok yüz" demiştik; bu bölümden sonra söylemesi gereken bir cümle daha var: **tek sunucu, çok motor.**

Bölüm 30

OxiMem: Redis'in Telini Konuşan Bellek-İçi Anahtar-Değer Katmanı

Bir veritabanı motorunun içinde neden bir anahtar-değer katmanı bulunsun? On üçüncü bölümde, belleğin diskten binlerce kat hızlı olduğunu ve her ciddi veritabanının, aslında, "hangi veriyi bellekte tutayım" sorusuna verilmiş bir cevap olduğunu görmüştük. OxiMem, bu sorunun bir başka biçimine verilmiş cevaptır: **bazı veriler diske hiç inmemeli.**

Bir uygulamanın veri kümesi tek tip değildir. Bir tarafta, kaybolmaması gereken veri vardır: siparişler, işlemler, kullanıcı kayıtları. Bunlar belge motorunun işidir; yazma-öncesi günlüğe yazılır, indekslenir, işlemler içinde korunur, ve sunucu çökse bile geri gelirler. Ama aynı uygulamanın öteki tarafında, bambaşka karakterde bir veri kümesi vardır: bir oturum belirteci, on beş dakika sonra zaten geçersiz olacaktır. Bir sayfa görüntülenme sayacı, saniyede yüzlerce kez artar ve kaybolursa kimse ölmez. Bir hız sınırlama penceresi, tanımı gereği altmış saniye yaşar. Bir iş kuyruğu, birkaç saniyeliğine dolar ve boşalır. Bir emir defterinin en iyi alış fiyatı, milisaniyede bir değişir.

Bu ikinci kümeye belge motorunun tüm makinesini uygulamak — JSON'a serileştirmek, WAL'a yazmak, fsync beklemek, indeksleri güncellemek — israftır. Dayanıklılığın bedelini, dayanıklılığa ihtiyacı olmayan veri için ödemek anlamına gelir. OxiMem, tam olarak bu israfı ortadan kaldırmak için vardır: **JSON yok, WAL yok, indeks yok**; yalnızca hash tabloları, çift uçlu kuyruklar ve dengeli ağaçlar üzerinde çalışan, RESP telini konuşan bir bellek-içi katman.[1]

30.1 Neden Redis'in telini konuşmak?

Yeni bir bellek-içi depo yazmanın en kolay kısmı depodur; en zor kısmı, onu dünyaya bağlamaktır. İstemci kütüphaneleri, bağlantı havuzları, izleme araçları, komut satırı istemcileri, çerçeve entegrasyonları — bunların hepsi yıllar süren ekosistem emeğidir. Kendi protokolünüzü icat ederseniz, bu emeği baştan harcamanız gerekir.

OxiMem bu yolu seçmez. Bunun yerine, sektörde fiilen standart hale gelmiş bir teli — **RESP**'i (Redis Serialization Protocol) — konuşur. Bunun sonucu şudur: `redis-cli` ile OxiMem'e bağlanabilirsiniz; Python'da `redis-py`, Node'da `node-redis`, Go'da `go-redis` — hiçbirinde tek satır değişiklik gerekmez. Var olan bütün ekosistem, bedavaya gelir. Protokolü uyarlamak, ekosistemi kopyalamaktan kat kat ucuzdur.

RESP'in kendisi şaşırtıcı derecede basittir; sekiz yıllık bir protokolün bu kadar sade kalabilmesi, iyi tasarımın kanıtıdır. Her mesaj, ilk baytıyla türünü bildirir: `+` basit dize, `-` hata, `:` tamsayı, `$` toplu dize (uzunluk önekli),

[1] Kaynakta bu, tek cümleyle şöyle özetlenir: "commands operate on raw `HashMap`/`VecDeque`/`HashSet` structures for maximum throughput (no JSON, no WAL, no indexes)".

`*` dizi. Bir istemci komutu, her zaman toplu dizelerden oluşan bir dizidir. `GET oturum:42` komutunun hattaki tam karşılığı şudur:

```
*2\r\n$3\r\nGET\r\n$8\r\noturum:42\r\n
```

Yani: "iki elemanlı bir dizi geliyor; birincisi üç baytlık bir dize (`GET`), ikincisi sekiz baytlık bir dize (`oturum:42`)". Sunucunun cevabı da aynı dilbilgisiyle kodlanır:

```
$5\r\nahmet\r\n        (değer bulundu: "ahmet")
$-1\r\n                (anahtar yok: null toplu dize)
:42\r\n                (bir sayaç sonucu)
+OK\r\n                (başarı)
-ERR unknown command   (hata)
*-1\r\n                (null dizi — birazdan göreceğiz: iptal edilmiş işlem)
```

Uzunluk önekli olması, çerçeveleme sorununu — yirmi dördüncü bölümde gördüğümüz o "bir mesaj nerede biter" sorusunu — çözer. OxiMem'in çözümleyicisi, kendi ikili protokolünde olduğu gibi bir üst sınır uygular: on altı mebibaytı aşan bir uzunluk bildirimi, tek bayt bellek ayrılmadan reddedilir. Gerçek Redis burada daha cömerttir, ama cömertlik, kimliği doğrulanmamış bir istemcinin sunucuyu bellek ayırmaya zorlaması demektir.[2]

30.2 Açmak: bir port

OxiMem varsayılan olarak **kapalıdır**; kullanmadığınız şeyin bedelini ödemezsiniz. Açmak için tek bir ortam değişkeni yeter:

```
# RESP dinleyicisini 6380 portunda aç
export OXIDB_OXIMEM_PORT=6380
./oxidb-server

# Artık standart redis-cli ile bağlanabilirsiniz
redis-cli -p 6380 PING
# PONG
```

Buradan sonrası, alışkın olduğunuz her şeydir.

30.3 Komut ailesi

OxiMem, Redis'in komut yüzeyinin bir alt kümesini uygular — ama pratikte kullanılan çekirdeğin neredeyse tamamını. Aşağıdaki tablo, motorun fiilen tanıdığı komutları aileleriyle listeler; bu listede olmayan bir komut, `unknown command` hatasıyla döner.

Aile	Komutlar
Bağlantı	`PING` `ECHO` `QUIT` `SELECT` `AUTH` `HELLO` `CLIENT` `COMMAND`

[2] Bu sınır, bir bulanık test (fuzzing) çalışmasının bulgusudur: uzunluk alanı saldırgan denetimindedir, ve `vec![0u8; len]` gibi masum bir satır, hizmet engelleme açığına dönüşür.

Aile	Komutlar
Dize	`SET GET GETSET SETNX SETEX PSETEX MSET MGET APPEND STRLEN GETRANGE SETRANGE GETDEL GETEX`
Sayaç	`INCR DECR INCRBY DECRBY INCRBYFLOAT DECRBYFLOATGE`
Anahtar / TTL	`DEL EXISTS TYPE KEYS SCAN RANDOMKEY RENAME COPY DBSIZE EXPIRE PEXPIRE EXPIREAT PEXPIREAT PERSIST TTL PTTL FLUSHDB FLUSHALL`
Hash	`HSET HMSET HSETNX HGET HMGET HGETALL HDEL HEXISTS HKEYS HVALS HLEN HINCRBY HRANDFIELD HSCAN`
Liste	`LPUSH RPUSH LPOP RPOP LLEN LRANGE LINDEX LSET LREM LTRIM LMOVE RPOPLPUSH LMPOP`
Küme	`SADD SREM SMEMBERS SISMEMBER SMISMEMBER SCARD SPOP SRANDMEMBER SINTER SUNION SDIFF SINTERSTORE SUNIONSTORE SDIFFSTORE SSCAN`
Sıralı küme	`ZADD ZREM ZSCORE ZCARD ZCOUNT ZINCRBY ZRANK ZREVRANK ZRANGE ZREVRANGE ZRANGEBYSCORE ZREVRANGEBYSCORE ZRANGEBYLEX ZPOPMIN ZPOPMAX ZREMRANGEBYRANK ZREMRANGEBYSCORE ZUNIONSTORE ZINTERSTORE ZMPOP ZSCAN`
Bit	`SETBIT GETBIT BITCOUNT`
Engelleyen	`BLPOP BRPOP BZPOPMIN BLMOVE BRPOPLPUSH`
Yayın/abonelik	`PUBLISH SUBSCRIBE PSUBSCRIBE PUNSUBSCRIBE`
İşlem	`MULTI EXEC DISCARD WATCH UNWATCH`
Betik	`EVAL EVALSHA SCRIPT`
Sunucu	`INFO CONFIG`

Bu tablodaki tek yabancı isim `DECRBYFLOATGE`'dir; o, Redis'te bulunmayan, OxiMem'e özgü bir eklemedir ve biraz sonra sırası gelecek.

En basitinden başlayalım. Dize komutları, anahtar-değer fikrinin en saf halidir:

```
redis-cli -p 6380

SET kullanici:42:ad "Ayşe"     # OK
GET kullanici:42:ad            # "Ayşe"
EXISTS kullanici:42:ad         # (integer) 1
STRLEN kullanici:42:ad         # (integer) 5
APPEND kullanici:42:ad " Y."   # (integer) 8
SETNX kullanici:42:ad "Başka"  # (integer) 0  — zaten var, yazmaz
DEL kullanici:42:ad            # (integer) 1
GET kullanici:42:ad            # (nil)
```

Sayaçlar, anahtar-değerin en çok işe yarayan yüzüdür; çünkü tek bir komutta oku-değiştir-yaz üçlüsünü **atomik** biçimde yaparlar. Bir web isteği içinde "oku, bire ekle, geri yaz" yazarsanız, iki eşzamanlı istek birbirinin artışını yutar; `INCR` ise böyle bir yarışa yer bırakmaz:

```
INCR sayfa:anasayfa:goruntulenme      # (integer) 1
INCR sayfa:anasayfa:goruntulenme      # (integer) 2
INCRBY sayfa:anasayfa:goruntulenme 10 # (integer) 12
DECR sayfa:anasayfa:goruntulenme      # (integer) 11
```

```
# Hash alanları da atomik olarak artırılabilir
HINCRBY istatistik:2026-07 istek 1    # (integer) 1
HINCRBY istatistik:2026-07 hata 1     # (integer) 1
HGETALL istatistik:2026-07            # 1) "istek" 2) "1" 3) "hata" 4) "1"
```

Listeler bir kuyruğa, kümeler bir üyelik testine, sıralı kümeler bir puan tablosuna dönüşür:

```
# Liste: üretici-tüketici kuyruğu (sağdan it, soldan çek = FIFO)
RPUSH isler "is-1" "is-2" "is-3"   # (integer) 3
LLEN isler                          # (integer) 3
LRANGE isler 0 -1                   # "is-1" "is-2" "is-3"
LPOP isler                          # "is-1"

# Engelleyen çekme: kuyruk boşsa 5 saniyeye kadar bekler
BLPOP isler 5                       # 1) "isler" 2) "is-2"

# Küme: üyelik
SADD cevrimici:kullanicilar 42 77 91  # (integer) 3
SISMEMBER cevrimici:kullanicilar 77   # (integer) 1
SCARD cevrimici:kullanicilar          # (integer) 3

# Sıralı küme: puan tablosu (ve emir defteri!)
ZADD skorlar 1500 "ayse" 1800 "mehmet" 1200 "zeynep"
ZINCRBY skorlar 400 "zeynep"          # "1600"
ZREVRANGE skorlar 0 2 WITHSCORES      # mehmet 1800, zeynep 1600, ayse 1500
ZRANK skorlar "ayse"                  # (integer) 0  — en düşük puanlı
```

Sıralı kümenin altında, on yedinci bölümün B-ağacı fikrinin küçük bir yankısı vardır: puanlar hem bir hash tablosunda (üye → puan, O(1) arama) hem de dengeli bir ağaçta (puan, üye) tutulur; böylece "en iyi alış fiyatı" sorusu — bir borsa uygulamasının en sıcak sorusu — `ZREVRANGE anahtar 0 0` ile logaritmik zamanda cevaplanır.

30.4 TTL ve süre dolumu

Anahtar-değer katmanının belge motorundan ayrıldığı en karakteristik nokta, verinin **kendiliğinden ölebilmesidir**. Bir oturum belirtecini silmek için bir temizlik işi yazmazsınız; ona bir ömür verirsiniz ve unutursunuz.

```
# Ömürle birlikte yaz — üç eşdeğer yol
SETEX oturum:abc 900 "kullanici=42"    # 900 saniye
SET   oturum:abc "kullanici=42" EX 900 # aynısı, SET seçeneğiyle
SET   oturum:abc "kullanici=42" PX 500 # milisaniye cinsinden

TTL oturum:abc      # (integer) 899   — kalan saniye
PTTL oturum:abc     # kalan milisaniye
PERSIST oturum:abc  # (integer) 1     — ömrü kaldır, kalıcı yap
TTL oturum:abc      # (integer) -1    — ömrü yok
TTL yok:boyle:bir   # (integer) -2    — anahtar hiç yok
```

Mekanizma iki katmanlıdır ve her ikisi de gereklidir. Birincisi **tembel süre dolumudur**: bir anahtar okunduğunda, önce son kullanma zamanı denetlenir; geçmişse anahtar o anda silinir ve okuma "yok" cevabı döner. Bu, doğruluğu garanti eder — süresi dolmuş bir değeri asla göremezsiniz. Ama tek başına yetmez: hiç

okunmayan bir anahtar, süresi dolduğu halde sonsuza dek bellekte kalırdı. Bu yüzden ikinci katman vardır: saniyede bir uyanan bir **süpürücü iş parçacığı**, süresi dolmuş anahtarları toplu olarak siler. Süpürücü ayrıca, silinen her anahtar için — açıksa — bir `expired` bildirimi yayımlar ve birazdan göreceğimiz sürüm sayacını artırır; yani bir anahtarın sessizce ölmesi bile, onu izleyen bir işlem için bir **değişikliktir**.

Burada dürüst olmak gereken bir sınır var: OxiMem'de ömür yalnızca **dize anahtarlarına** uygulanır. Bir hash'e, listeye ya da kümeye `EXPIRE` verirseniz `0` alırsınız — yani "böyle bir şey yapmadım". Bunun pratik sonucu, oturum önbelleği gibi ömürlü verileri hash olarak değil, tek bir serileştirilmiş dize olarak tutmanız gerektiğidir.[3] Aşağıdaki oturum önbelleği deseni, bu kısıtı tam olarak böyle karşılar:

```
import json
import redis

r = redis.Redis(host="localhost", port=6380, decode_responses=True)

OTURUM_OMRU = 15 * 60  # 15 dakika

def oturum_yaz(belirtec: str, kullanici: dict) -> None:
    # Hash'lere TTL uygulanmadığı için oturumu TEK bir dizeye serileştiriyoruz.
    r.setex(f"oturum:{belirtec}", OTURUM_OMRU, json.dumps(kullanici))

def oturum_oku(belirtec: str) -> dict | None:
    ham = r.get(f"oturum:{belirtec}")
    if ham is None:
        return None                      # süresi dolmuş ya da hiç yok
    # Her erişimde ömrü tazele — "kayan pencere" oturumu
    r.expire(f"oturum:{belirtec}", OTURUM_OMRU)
    return json.loads(ham)

oturum_yaz("abc123", {"id": 42, "ad": "Ayşe", "rol": "admin"})
print(oturum_oku("abc123"))   # {'id': 42, 'ad': 'Ayşe', 'rol': 'admin'}
print(oturum_oku("yokboyle")) # None
```

30.5 MULTI, EXEC ve WATCH: aynı fikir, küçük ölçekte

Onuncu ve on birinci bölümlerde, işlemleri ve eşzamanlılık denetimini uzun uzun konuşmuştuk. Orada anlattığımız **iyimser eşzamanlılık denetiminin** (OCC) özü şuydu: kilitleme, oku ve çalış; yazma anında, okuduğun şeylerin değişmediğini doğrula; değiştiyse iptal et ve yeniden dene. Çatışmanın nadir olduğu bir dünyada bu, kilitlemekten çok daha ucuzdur.

OxiMem'in işlem modeli, tam olarak bu fikrin küçük ölçekte tekrarıdır — ve bu bir tesadüf değil, yakınsamadır: aynı problem, aynı çözümü doğurur.

`MULTI`, bir kuyruk açar. Ondan sonra gönderdiğiniz her komut çalıştırılmaz; `QUEUED` cevabıyla biriktirilir. `EXEC` geldiğinde, biriken komutların hepsi **arka arkaya, araya başka bir işlem girmeden** çalıştırılır:

```
MULTI                   # OK
SET sayac 5             # QUEUED  — çalışmadı, kuyruğa girdi
INCRBY sayac 10         # QUEUED
EXEC                    # 1) OK  2) (integer) 15
GET sayac               # "15"
```

[3] TTL, saniye çözünürlüğünde tutulur; `PX` ve `PSETEX` milisaniye değerlerini yukarı yuvarlar.

Redis bu yalıtımı bedavaya alır: tek iş parçacıklı olduğu için, iki `EXEC` bloğu zaten iç içe geçemez. OxiMem çok iş parçacıklıdır; bu yüzden yalıtımı açıkça satın alır: `EXEC`, mağaza düzeyinde bir işlem kilidi alır ve yalnızca kendi kuyruğunu boşaltırken elinde tutar. İki `EXEC` asla komutlarını birbirine karıştıramaz.

Ama `MULTI/EXEC` tek başına, onuncu bölümdeki anlamıyla bir işlem değildir: atomikliği verir, **yalıtımın "okuduğumu doğrula" yarısını vermez**. "Bakiyeyi oku, yeterliyse düş" gibi bir mantık, okumayla yazma arasında birinin araya girmesine açıktır — klasik kayıp güncelleme anomalisi.

İşte `WATCH` tam burada devreye girer, ve OCC'nin **doğrulama fazının** ta kendisidir. `WATCH anahtar` dediğinizde, OxiMem o anahtarın o andaki durumunun bir **parmak izini** alır. `EXEC` geldiğinde, işlem kilidinin altında bu parmak izleri yeniden hesaplanır; biri bile değişmişse işlem **çalıştırılmaz** ve `EXEC`, RESP'in null dizisini (`*-1`) döner. İstemci bunu "yeniden dene" diye okur.

Parmak izinin nasıl hesaplandığı, on birinci bölümdeki sürüm sayacı fikrinin doğrudan uygulamasıdır. Değerin bir kopyası alınmaz — bir emir defteri sıralı kümesini `WATCH` etmek, o defteri kopyalamak anlamına gelseydi, bu O(n) maliyetiyle her şeyi öldürürdü. Bunun yerine parmak izi üç küçük parçadan oluşur: anahtarın **mutasyon sayacı** (her yazma komutu, dokunduğu anahtarın sayacını bir artırır), mağazanın **çağ sayacı** (`FLUSHALL` her şeyi geçersiz kılar) ve anahtarın **var olup olmadığı** (böylece iki okuma arasında sessizce süresi dolan bir anahtar da bir değişiklik sayılır). Üçü de tamsayıdır; anlık görüntü almak O(1)'dir.

Bu tasarımın bilinçli bir asimetrisi vardır: sayaç, yazma komutu hiçbir şeyi değiştirmemiş olsa bile artar (aynı değeri tekrar yazmak gibi). Yani **yanlış alarm mümkündür** — işlem gereksiz yere iptal edilip yeniden denenebilir. Ama **kaçırılmış değişiklik imkânsızdır**. Bu, doğruluğun performansa tercih edildiği yerlerden biridir; yanlış alarmın bedeli bir yeniden deneme, kaçırmanın bedeli bozuk veridir.

Klasik yarışı görelim. İki uçbirim; A cüzdanı izlerken B araya giriyor:

```
# --- Uçbirim A ---
SET cuzdan:42 100
WATCH cuzdan:42          # OK — parmak izi alındı
GET cuzdan:42            # "100" — kararımızı bu değere göre vereceğiz

# --- Uçbirim B (araya giriyor) ---
SET cuzdan:42 999        # OK

# --- Uçbirim A (devam) ---
MULTI                    # OK
SET cuzdan:42 50         # QUEUED
EXEC                     # (nil)  — İPTAL: izlenen anahtar değişti
GET cuzdan:42            # "999"  — A'nın yazması hiç gerçekleşmedi
```

`EXEC`'in hattaki cevabı tam olarak şudur — ve bu, "boş dize" ile karıştırılmaması gereken ayrı bir RESP değeridir:

```
*-1\r\n      (null dizi = işlem iptal edildi; boş dizi *0\r\n değildir!)
```

Gerçek bir uygulamada bu döngüyü elle yazmazsınız; istemci kütüphanesi yazar. `redis-py`'nin deseni, OCC'nin ders kitabı biçimidir — oku, karar ver, doğrula, çatışırsa yeniden dene:

```python
import redis
from redis import WatchError

r = redis.Redis(host="localhost", port=6380, decode_responses=True)
r.set("cuzdan:42", 100)
```

```python
def para_dus(cuzdan: str, tutar: int, deneme: int = 5) -> bool:
    """Bakiyeyi oku, yeterliyse düş — yarış olursa yeniden dene (OCC)."""
    with r.pipeline() as boru:
        for _ in range(deneme):
            try:
                boru.watch(cuzdan)                  # parmak izini al
                bakiye = int(boru.get(cuzdan) or 0) # okuma (henüz kuyruk yok)
                if bakiye < tutar:
                    boru.unwatch()
                    return False                    # yetersiz bakiye
                boru.multi()                        # buradan sonrası kuyruğa girer
                boru.set(cuzdan, bakiye - tutar)
                boru.execute()                      # EXEC: doğrula + çalıştır
                return True
            except WatchError:
                continue                            # biri araya girdi — tekrar dene
    raise RuntimeError("çekişme çok yüksek, işlem tamamlanamadı")

print(para_dus("cuzdan:42", 30))   # True
print(r.get("cuzdan:42"))          # "70"
print(para_dus("cuzdan:42", 500))  # False — bakiye yetmiyor
```

Bu döngüde OCC'nin üç fazını da görebilirsiniz: `watch` + `get` **okuma fazı**, `multi` ile biriken komutlar **yazma tamponu**, `execute` ise **doğrulama ve işleme** fazıdır. Onuncu bölümde belge motoru için çizdiğimiz şemanın, birkaç mikrosaniyeye sığdırılmış hali.

30.6 EXECABORT: hataları erken yakalamak

Bir işlemin ortasında bir komutun çalışmayacağını, `EXEC` sırasında öğrenmek kötüdür: komutların yarısı çalışmış, yarısı çalışmamış olurdu. OxiMem bu yüzden **kuyruğa alma zamanında** doğrulama yapar: kuyruğa giren komut tanınmıyorsa ya da argüman sayısı yetersizse, o anda hata döner **ve işlem zehirlenir**. Sonraki `EXEC`, hiçbir şey çalıştırmadan `EXECABORT` ile reddedilir:

```
MULTI                  # OK
SET z 1                # QUEUED
NOPE z                 # (error) ERR unknown command 'NOPE'   ← zehirlendi
GET z                  # QUEUED  (kuyruk hâlâ alıyor, ama artık boşuna)
EXEC                   # (error) EXECABORT Transaction discarded because of
                       #         previous errors.
GET z                  # (nil)  — SET z 1 HİÇ çalışmadı
```

Bu, "ya hep ya hiç"in protokol düzeyindeki savunmasıdır: yazım hatası içeren bir işlem, veriye hiç dokunmaz.

30.7 Sunucu tarafı betikler ve atomik borç düşme

`WATCH` döngüsü, çatışma seyrekken mükemmeldir; ama çekişme yüksek olduğunda istemci ile sunucu arasında birkaç gidiş-geliş yapar ve yeniden denemeler artar. Bu yüzden OxiMem, Redis gibi, **sunucu tarafında betik** çalıştırabilir: gömülü bir Lua yorumlayıcısı,[4] tüm oku-karar-ver-yaz mantığını tek bir atomik adımda, sunucunun içinde çalıştırır. Gidiş-geliş sayısı: bir.

[4] Lua 5.4, `mlua` üzerinden gömülüdür. Betikler `redis.call` / `redis.pcall` ile komut çalıştırır; `cjson` ve `redis.sha1hex` mevcuttur. Sonsuz döngüye giren bir betik, `SCRIPT KILL` ile durdurulabilir.

```
# Yeterli bakiye varsa düş, yoksa dokunma — tek turda, atomik
EVAL "local b = tonumber(redis.call('GET', KEYS[1])) \
      if b >= tonumber(ARGV[1]) then \
        redis.call('INCRBYFLOAT', KEYS[1], '-' .. ARGV[1]) \
        return 1 \
      end \
      return 0" 1 cuzdan:42 30

# (integer) 1  — düşüldü
GET cuzdan:42
# "40"
```

Betikleri her seferinde göndermek yerine `SCRIPT LOAD` ile bir kez yükleyip `EVALSHA` ile SHA-1 özetinden çağırabilirsiniz — hat üzerinde yüz baytlık bir betik yerine kırk baytlık bir özet gider.

Bu "kontrol et ve düş" deseni o kadar yaygındır ki, OxiMem onu Lua'ya bile ihtiyaç bırakmayan yerel bir komut olarak sunar. `DECRBYFLOATGE` — "greater-or-equal ise düş" — Redis'te bulunmayan, OxiMem'e özgü tek komuttur: yeterli bakiye varsa atomik olarak düşer ve yeni bakiyeyi döner, yoksa hiç dokunmadan `nil` döner. Kredi limitini aşan bir borçlandırmanın yarış koşuluyla bile gerçekleşmesi imkânsızdır:

```
SET bakiye:42 100
DECRBYFLOATGE bakiye:42 30    # "70"   — düşüldü
DECRBYFLOATGE bakiye:42 200   # (nil)  — yetersiz; bakiyeye DOKUNULMADI
GET bakiye:42                 # "70"
```

30.8 Gerçek bir desen: hız sınırlama

Şimdiye kadarki parçaları birleştiren, üretimde en çok kullanılan desenlerden birine bakalım: **sabit pencereli hız sınırlama**. Fikir basittir: her kullanıcı için, pencerenin adını taşıyan bir sayaç anahtarı tutarsınız; her istekte artırır, ilk artışta ona bir ömür verirsiniz. Pencere dolduğunda anahtar kendiliğinden ölür — temizlik işi yoktur. Sayma `INCR` ile atomiktir, ve `INCR` ile `EXPIRE`'ı bir `MULTI` bloğuna koyarak ikisini tek turda gönderiyoruz:

```
import { createClient } from "redis";

const istemci = createClient({ url: "redis://localhost:6380" });
await istemci.connect();

const LIMIT = 100;        // pencere başına izin verilen istek
const PENCERE = 60;       // saniye

async function izinVarMi(kullaniciId) {
  // Pencere numarasını anahtara gömüyoruz: her dakika yeni anahtar doğar,
  // eskisi TTL ile kendiliğinden ölür.
  const pencere = Math.floor(Date.now() / 1000 / PENCERE);
  const anahtar = `hiz:${kullaniciId}:${pencere}`;

  // INCR + EXPIRE'ı tek bir MULTI/EXEC turunda gönder (atomik ve tek gidiş-geliş)
  const [sayi] = await istemci
    .multi()
    .incr(anahtar)
    .expire(anahtar, PENCERE)
```

```
    .exec();

  return { izin: sayi <= LIMIT, kalan: Math.max(0, LIMIT - sayi) };
}

console.log(await izinVarMi(42));  // { izin: true, kalan: 99 }
```

Bu on satırın altında, bu bölümün bütün fikirleri var: atomik sayaç, kendiliğinden ölen anahtar, tek turda çalışan bir işlem. Aynı işi belge motorunda yapmak — her istekte bir belge okuyup güncellemek, WAL'a yazmak, fsync beklemek — hem yüz kat pahalı olurdu, hem de gereksizdi: bu sayaç kaybolsa, kullanıcı birkaç istek fazla yapardı, o kadar.

Yayın-abonelik de aynı sadelikte çalışır ve gerçek zamanlı bildirimler için yeterlidir:

```
const abone = istemci.duplicate();
await abone.connect();

// Kalıp aboneliği: "fiyat." ile başlayan tüm kanallar
await abone.pSubscribe("fiyat.*", (mesaj, kanal) => {
  console.log(`${kanal} → ${mesaj}`);   // fiyat.BTC → 68500
});

await istemci.publish("fiyat.BTC", "68500");
```

30.9 Tek bir keyspace — bilinçli bir tercih

OxiDB, çok veritabanlı bir motordur: her veritabanının kendi koleksiyonları, kendi SQL motoru, kendi işlemleri vardır. OxiMem ise bunun dışındadır: **keyspace küreseldir.** Hangi veritabanına bağlı olursanız olun, `SET a 1` aynı tek keyspace'e yazar. `SELECT` komutu kabul edilir ve `OK` döner — ama hiçbir şey yapmaz.

Bu bir eksiklik değil, bir tercihtir. OxiMem'in varlık nedeni, uygulamanın **sıcak kenarıdır**: oturumlar, sayaçlar, kuyruklar, hız sınırları. Bunlar tipik olarak veritabanı sınırlarını umursamaz; bir oturum belirteci, hangi mantıksal veritabanına ait olduğu sorusuna anlamlı bir cevap vermez. Keyspace'i veritabanlarına bölmek, kazanç sağlamadan bir dolaylılık katmanı eklerdi. Ayrım isteyen, anahtar öneki kullanır — `musteri:7:oturum:abc` — ki bu, Redis dünyasının da yerleşik pratiğidir.

30.10 Kalıcılık: iki seçenek, ikisi de isteğe bağlı

"Bellek-içi" demek, "veri kaybolabilir" demektir; ve bu, çoğu kullanım için doğru karardır. Ama "yeniden başlatmayı atlatsın yeter" diyen bir orta yol da vardır ve OxiMem iki biçimde sunar.

Birincisi, **anlık görüntüdür**. `OXIDB_OXIMEM_SNAPSHOT_SECS` verilirse, mağazanın tamamı belirtilen aralıklarla bir dosyaya yazılır; sunucu açılışta bu dosyayı okuyup belleği yeniden kurar. Yazma, geçici dosyaya yaz–fsync–yeniden adlandır üçlüsüyle yapılır; altıncı bölümde gördüğümüz **atomik değiştirme** desenidir bu: yazmanın ortasında çöken bir sunucu, önceki anlık görüntüyü asla bozamaz.

İkincisi, **belge motoruna aynalamadır**. `OXIDB_OXIMEM_SQL` açıksa, her yazma aynı zamanda belge motorunun `_kv`, `_hash`, `_list`, `_set`, `_zset` koleksiyonlarına yansıtılır; açılışta bellek, bu koleksiyonlardan yeniden kurulur — ömürler dahil, çünkü ayna mutlak son kullanma zamanını da saklar. Bunun ikinci bir faydası vardır: sıcak veri, birden bire **sorgulanabilir** hale gelir. Ama bedeli açıktır ve gizlenmemelidir: her yazma artık belge motoruna da uğrar; OxiMem'in hız avantajının büyük kısmı gider. Bu yüzden mod, adında dürüstçe ayrılır — biri "hızlı kip", diğeri "aynalama kipi".

Redis'in AOF'una — her komutu diske ekleyen, saniyede bir fsync'leyen günlüğe — karşılık gelen bir şey **yoktur**. Anlık görüntü kipinde, son görüntüden sonraki pencere, çökmede kaybolur. Bunu bilerek kullanın: OxiMem'in kalıcılığı, "yeniden başlatmayı atlatmak" içindir, "hiçbir yazmayı kaybetmemek" için değil. İkincisini istiyorsanız, doğru yer belge motorudur.

30.11 Performans: Redis'e ne kadar yakın?

Bir Redis uyumlu katmanın tek anlamlı ölçütü, Redis'in kendisidir. `redis-benchmark` ile (100 bin istek, 50 istemci, 64 baytlık değerler) alınan tek-komut sonuçları şöyledir:

Test	Redis	OxiMem	Oran
PING	250K op/s	224K op/s	%90
GET	251K op/s	240K op/s	%96
SET	249K op/s	231K op/s	%93
INCR	249K op/s	187K op/s	%75
LPUSH	254K op/s	242K op/s	%95
RPUSH	238K op/s	242K op/s	%101
SADD	253K op/s	239K op/s	%94
HSET	253K op/s	193K op/s	%76
MSET (10)	164K op/s	232K op/s	%141

Yani tek komut modunda OxiMem, Redis'in yüzde 75 ile 101'i arasındadır; birkaç komutta onu geçer. Bu, yirmi yıllık, elle optimize edilmiş C veri yapılarına karşı, standart Rust koleksiyonlarıyla alınmış saygıdeğer bir sonuçtur. Asıl kazanç ise hızın kendisi değil, bu hızın **aynı süreç içinde**, belge motorunun yanı başında bulunmasıdır: ayrı bir Redis kurmak, işletmek ve sürümlemek gerekmez.

Dürüst olmak gerekirse boru hattı (pipelining) tarafında Redis hâlâ öndedir; tek iş parçacıklı olay döngüsü ve özel veri yapıları, yüzlerce komutun tek turda işlendiği senaryolarda ona avantaj sağlar. OxiMem, ardışık aynı-komut yığınlarını tek kilit altında birleştiren bir hızlı yol ekleyerek arayı büyük ölçüde kapatmıştır — ama kapatmıştır, geçmemiştir.

30.12 Eksikler, sınırlar ve ne zaman hangisi

Bu bölümü, olmayanları açıkça söyleyerek bitirmek gerekir; çünkü bir aracın sınırlarını bilmemek, onu yanlış yerde kullanmanın en kısa yoludur.

Kimlik doğrulama yoktur. `AUTH` komutu kabul edilir ve `OK` döner — ama hiçbir şey doğrulamaz. OxiMem dinleyicisi, açıldığı ağda kime açıksa, ona tam yetki verir. Bu yüzden onu asla halka açık bir arayüze bağlamayın; yerel arayüzde ya da güvenilir bir ağ bölmesinde tutun. Belge motorunun SCRAM kimlik doğrulaması ve rol tabanlı erişim denetimi, OxiMem portunu **kapsamaz**.

Kümeleme ve çoğaltma yoktur. OxiMem düğüm yereldir; Raft ile çoğaltılmaz. Bir kümede her düğümün kendi keyspace'i vardır. Sıcak veri için bu genellikle kabul edilebilir; paylaşılan durum için değildir.

RESP3 yoktur. `HELLO 3` diyen bir istemci `NOPROTO` cevabı alır ve RESP2'ye düşer — ki bütün büyük istemciler bunu sorunsuzca yapar.

Anahtar ömrü yalnızca dizelere uygulanır; hash, liste, küme ve sıralı kümeler kalıcıdır ve elle silinmelidir.

Kayıpsız kalıcılık yoktur: AOF benzeri bir komut günlüğü bulunmaz.

Peki karar? Basit bir soruyla verilir: **bu veriyi kaybetsem ne olur?**

Cevap "hiçbir şey, yeniden hesaplanır ya da zaten kısa ömürlü" ise — oturumlar, önbellekler, sayaçlar, hız sınırları, kısa ömürlü kuyruklar, canlı fiyatlar, emir defterinin anlık hali — yer OxiMem'dir. Bu veriye dayanıklılık uygulamak, hiçbir şey kazandırmadan onu yüz kat yavaşlatır.

Cevap "para kaybederim, denetim kaydım eksilir, müşteri kızar" ise — siparişler, gerçekleşmiş işlemler, kullanıcı kayıtları, defterler — yer belge motorudur. Orada yazma-öncesi günlük, işlemler, indeksler ve kurtarma vardır; ve bunların bedelini seve seve ödersiniz.

En güçlü kullanım ise, ikisini birlikte kullanmaktır ve bu kitabın örnek uygulamasında tam olarak böyle yapılır: bir borsanın emir defteri OxiMem'in sıralı kümelerinde yaşar, bakiyeler `WATCH`'lı işlemlerle atomik olarak devredilir — ama **gerçekleşen her işlem**, kalıcı bir defter kaydı olarak belge motoruna yazılır. Sıcak yol hızlı, soğuk yol dayanıklıdır. On üçüncü bölümün bellek-disk hiyerarşisi, burada iki motorlu bir mimariye dönüşmüştür; ve her verinin ait olduğu katmana yerleştirilmesi, bu kitabın baştan beri savunduğu tek ilkenin — **her tasarım bir ödünleşimdir, doğru olan, ödünleşimi bilerek seçmektir** — belki de en somut uygulamasıdır.

Bölüm 31

S3 Uyumlu Nesne Depolama: Büyük Baytların Yeri

Yirmi üçüncü bölümde, OxiDB'nin çekirdeğin etrafındaki ek yüzeylerinden söz ederken blob depolamaya kısaca değinmiştik: büyük ikili nesneler belgelerin içine gömülmez, kovalar halinde düzenlenmiş ayrı bir nesne deposunda tutulur; her nesne bir veri dosyası ile onu betimleyen bir üst veri dosyasından oluşur. Orada kavramı kurduk. Bu bölümde onu sonuna kadar açıyoruz: nesne deposunun disk üzerindeki düzeni nedir, hangi S3 operasyonları gerçekten uygulanmıştır, kimlik doğrulama nasıl çalışır, `aws-cli` ve `boto3` gibi hazır araçlar neden hiçbir uyarlama olmadan çalışır, MinIO ile karşılaştırıldığında ne kadar hızlıdır ve — en önemlisi — hangi veriyi belgeye, hangisini nesne deposuna koymalısınız.

31.1 Bir veritabanı neden nesne depolar?

Dördüncü ve beşinci bölümlerin ortak dersini hatırlayalım. Belge modeli, birlikte okunan veriyi birlikte saklamak üzerine kuruludur; bir belge, tek bir okumada diske bir kez gidilerek getirilen, kendi içinde tutarlı bir birimdir. Bu yapının verimliliği, belgelerin **küçük ve yapılı** kalmasına bağlıdır. Beşinci bölümde gördüğümüz ekle-yalnızca depolama, bir belgeyi güncellediğinizde onun yeni bir kopyasını dosyanın sonuna yazar; eski kopya ölü alan olarak kalır ve yirmi ikinci bölümdeki sıkıştırma (compaction) onu ancak sonradan temizler.

Şimdi bir megabaytlık bir profil fotoğrafını, o kullanıcının belgesine base64 olarak gömdüğünüzü düşünün. Üç şey aynı anda bozulur. Birincisi, **okuma maliyeti**: kullanıcının yalnızca adını ve e-postasını isteyen bir sorgu bile, o bir megabaytı diskten çekip JSON olarak ayrıştırmak zorunda kalır — üstelik base64, baytları yaklaşık üçte bir oranında şişirir. İkincisi, **yazma büyütmesi**: kullanıcının son giriş zamanını güncelleyen küçücük bir `$set`, ekle-yalnızca düzende belgenin **tamamının** — fotoğraf dahil — yeniden yazılmasına yol açar; bir zaman damgası için bir megabayt. Üçüncüsü, **parçalanma**: her güncelleme bir megabaytlık ölü alan bırakır, sıkıştırma işi katlanarak artar, önbellek büyük ve opak baytlarla dolar ve gerçekten sorgulanan sıcak veri önbellekten atılır.

Buradaki asıl gözlem şudur: bir görüntü, bir video ya da bir fatura PDF'i, veritabanının **hiçbir işine yaramayan** baytlardan oluşur. Veritabanı onları indeksleyemez, üzerinde karşılaştırma yapamaz, bir toplama işlemine sokamaz. Onlar için tek yaptığı şey taşımaktır. O halde onları, taşımayı ucuzlatan bir yapıya koymak gerekir: içeriğe göre sorgulanmayan, anahtarla adreslenen, akış halinde okunup yazılan, sabit boyutlu bir depo. Bu, tam olarak bir **nesne deposudur**.

OxiDB'nin cevabı, bu ihtiyacı ne dışarıya havale etmek ne de belge motorunun içine tıkıştırmaktır. Aynı sunucu süreci, ayrı bir dinleyici üzerinde S3 uyumlu bir nesne depolama API'si sunar. Böylece uygulamanız iki ayrı altyapı bileşeni işletmek zorunda kalmadan, doğru veriyi doğru motora koyabilir.

31.2 Kova ve nesne modeli

Model, S3'ten devralınmıştır ve sadeliği kasıtlıdır. **Kova** (bucket), bir isim uzayıdır; adı bir DNS etiketinin kurallarına uyar (küçük harf, rakam, tire; nokta ve büyük harf yok). **Nesne** (object), bir kova içinde bir **anahtarla** (key) adreslenen bayt dizisidir. Anahtar, düz bir dizedir; içindeki eğik çizgiler klasör görüntüsü verir ama gerçek bir dizin ağacı yoktur — `faturalar/2026/07/inv-991.pdf`, içinde eğik çizgi bulunan tek bir anahtardır. Listeleme sırasında `delimiter` parametresiyle bu düz isim uzayına bir klasör yanılsaması giydirebilirsiniz; motor ortak önekleri (`CommonPrefixes`) hesaplayıp döndürür.

Nesnenin kendisi opaktır, ama etrafındaki üst veri zengindir: boyut, içerik türü (MIME), oluşturulma zamanı, bütünlük damgası (ETag) ve **kullanıcı tanımlı üst veri** — S3'te `x-amz-meta-*` başlıklarıyla taşınan serbest anahtar/değer çiftleri.

31.3 Disk üzerindeki düzen: `.data` ve `.meta`

Her nesne, veri dizininin altındaki `_blobs/<kova>/` klasöründe **iki dosyaya** ayrılır: baytları taşıyan `<id>.data` ve onu betimleyen `<id>.meta`. Buradaki `id`, kova içinde artan bir sayaçtan gelir; anahtar → id eşlemesi bellekte tutulur. Bu ayrımın nedeni, üst verinin küçük, içeriğin ise büyük olmasıdır: `HEAD` ve `LIST` gibi yalnızca üst veri isteyen çağrılar, koca `.data` dosyasına hiç dokunmaz — hatta üst veri bellekteki önbellekten karşılanır ve diske hiç gidilmez.

`.meta` dosyası bir JSON belgesidir ve şuna benzer:

```
{
  "key": "faturalar/2026/07/inv-991.pdf",
  "bucket": "belgeler",
  "size": 184320,
  "content_type": "application/pdf",
  "etag": "3f2a91c0d4e5b6a7889900aabbccddee",
  "created_at": "2026-07-14T09:12:44Z",
  "metadata": { "musteri": "acme-ltd", "donem": "2026-07" },
  "storage_compression": "zstd",
  "stored_size": 121804,
  "format_version": 1
}
```

Üç alan özel dikkat ister. `etag`, içeriğin SHA-256 özetinin ilk 16 baytının onaltılık gösterimidir — yani 32 karakterlik bir bütünlük damgası. Nesneyi indirdikten sonra baytların özetini yeniden hesaplayıp bu damgayla karşılaştırarak, verinin yolda ya da diskte sessizce bozulmadığını doğrulayabilirsiniz.[1] `storage_compression` ve `stored_size`, yirmi üçüncü bölümde değindiğimiz seçici sıkıştırmanın izidir: metin, JSON ya da HTML gibi sıkışabilir türler zstd ile sıkıştırılıp saklanır, JPEG/MP4/ZIP gibi zaten doygun-entropili türler olduğu gibi yazılır. `format_version` ise ileriye dönük emniyet supabıdır: eski bir motor, tanımadığı daha yeni bir üst veri sürümünü okumayı reddeder.

Yazma yolu, altıncı bölümün dayanıklılık disiplinini izler. Baytlar önce geçici dosyalara yazılır, sonra yerlerine `rename` edilir — ve sıra önemlidir: **önce `.data`, sonra `.meta`**. Çünkü kurtarmada gerçeğin kaynağı `.meta` dosyasıdır; sahipsiz bir `.data` zararsızdır (bir sonraki açılışta süpürülür), ama karşılığı olmayan bir `.meta` hayalet okumaya yol açardı. `OXIDB_BLOB_SYNC=1` ile bu iki `rename` arasına dizin `fsync`'i girer ve başarılı bir dönüş, "baytlar diskte" anlamına gelir.

[1] S3'ün kendi ETag'i tek parçalı yüklemelerde içeriğin MD5'idir; OxiDB SHA-256 kesmesi kullanır. Damganın anlamı aynıdır — aynı baytlar aynı damgayı üretir — ama istemci tarafında "ETag == MD5" varsayan kod, doğal olarak, çalışmaz.

31.4 Sunucuyu S3 için açmak

Nesne depolama HTTP yüzeyi, isteğe bağlı bir derleme özelliğidir ve varsayılan olarak kapalıdır; açmadığınızda hiçbir bedeli yoktur.

```
# S3 özelliğiyle derle ve S3 dinleyicisini 9000 portunda aç.
# Kimlik bilgileri verilmezse sunucu uyarır ve API herkese açık kalır!
OXIDB_DATA=/var/lib/oxidb \
OXIDB_S3_PORT=9000 \
OXIDB_S3_ACCESS_KEY=AKIAIOSFODNN7EXAMPLE \
OXIDB_S3_SECRET_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY \
OXIDB_BLOB_SYNC=1 \
OXIDB_BLOB_COMPRESS=1 \
cargo run --release -p oxidb-server --features s3
```

Birden çok kullanıcıya ayrı anahtar çiftleri vermek isterseniz tek bir değişken yeter; dururken şifreleme de aynı şekilde açılır:

```
# Çok kullanıcılı kimlik bilgileri: "erisim:gizli" çiftleri, virgülle ayrılmış
export OXIDB_S3_CREDENTIALS="app1:s3cret-one,app2:s3cret-two"

# Sunucu tarafı şifreleme (SSE-S3) için 32 baytlık anahtar (onaltılık)
export OXIDB_S3_ENCRYPTION_KEY="$(openssl rand -hex 32)"
# Her nesne, istemci istemese bile şifrelensin:
export OXIDB_S3_DEFAULT_ENCRYPTION=true

# Tarayıcıdan doğrudan erişim için CORS kaynağı (varsayılan: *)
export OXIDB_S3_CORS_ORIGIN="https://uygulamam.example.com"
```

Dinleyici, sabit bir iş parçacığı havuzuyla (256 çalışan, 1024 derinlikte kuyruk) çalışır, HTTP/1.1 kalıcı bağlantılarını (keep-alive) destekler ve tek bir `PUT` gövdesinde 5 GiB'a kadar kabul eder.

31.5 Kimlik doğrulama: AWS Signature V4

Uyumluluğun bel kemiği imza şemasıdır. OxiDB, AWS'in **Signature Version 4** protokolünü doğrular; hem başlık tabanlı imzayı (`Authorization: AWS4-HMAC-SHA256 ...`) hem de **önceden imzalanmış URL'leri** (`X-Amz-Signature` sorgu parametresiyle, `X-Amz-Expires` süre sınırı denetlenerek) tanır. Sunucu, istemcinin kanonik isteğini yeniden kurar, kendi tarafındaki gizli anahtarla imzalama anahtarını türetir ve iki imzayı **sabit zamanlı** karşılaştırır — zamanlama saldırısına kapı bırakmadan.

İki pratik sonuç: imza kapsamındaki bölge (region) istemcinin bildirdiği kapsamdan okunur, dolayısıyla istemci hangi bölgeyi seçerse seçsin, kendi imzasıyla tutarlı olduğu sürece çalışır. Ve OxiDB **yalnızca yol tarzı** (path-style) adreslemeyi destekler: `http://sunucu:9000/kova/anahtar`. Sanal konak tarzı (`kova.sunucu`) yoktur; istemcinizde `force_path_style` benzeri seçeneği açmanız gerekir.

31.6 Hangi S3 operasyonları uygulanmış?

İşte kaynaktan doğrulanmış tam liste:

Alan	Operasyonlar
Servis	`ListBuckets`
Kova	`CreateBucket`, `DeleteBucket`, `HeadBucket`, `ListObjectsV2` (`prefix`, `max-keys`, `delimiter` + `CommonPrefixes`, `continuation-token`)
Nesne	`PutObject`, `GetObject`, `HeadObject`, `DeleteObject`, `CopyObject` (kovalar arası; `COPY`/`REPLACE` üst veri yönergesi)
Kısmi/koşullu	`Range` (`bytes=a-b`, açık uçlu, sonek), `If-Match` (412), `If-None-Match` (304), kopyada `x-amz-copy-source-if-[none-]match`
Çok parçalı	`CreateMultipartUpload`, `UploadPart`, `CompleteMultipartUpload`, `AbortMultipartUpload` (≤10.000 parça, ≤5 GiB toplam, 24 saat sonra terk edilenler toplanır)
Toplu	`DeleteObjects` (`POST /kova?delete`)
Etiketleme	`PutObjectTagging`, `GetObjectTagging`, `DeleteObjectTagging`
Yaşam döngüsü	`PutBucketLifecycleConfiguration` / `Get` / `Delete` (yalnızca `Expiration` → `Days`; 5 dakikada bir süpürücü)
Şifreleme	SSE-S3 (`x-amz-server-side-encryption: AES256`), SSE-C (istemci anahtarı)
Erişim	SigV4 (başlık + önceden imzalı URL), CORS ön-uçuş (`OPTIONS`)

Ve dürüst olmak gerekirse, **olmayanlar**: nesne sürümleme (versioning), ACL ve kova politikaları, `ListMultipartUploads`/`ListParts`, çapraz-bölge replikasyon, `S3 Select`, sanal konak tarzı adresleme. Bu listeye güvenin; bir istemcinin "desteklenmiyor" hatası alması, çoğu zaman buradaki bir satırdır.

31.7 Hazır ekosistem: uyumluluğun asıl getirisi

S3 API'sini uygulamanın ödülü, yazılmayan koddur. `aws-cli`, `boto3`, AWS SDK'ları, MinIO istemcisi, `s3fs`, yedekleme araçları — hepsi hiçbir uyarlama olmadan çalışır. Önce komut satırı:

```
# Kimlik bilgilerini bir profile yaz
aws configure set aws_access_key_id     AKIAIOSFODNN7EXAMPLE      --profile oxidb
aws configure set aws_secret_access_key wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY --profile oxidb
aws configure set region                us-east-1                 --profile oxidb

E=http://127.0.0.1:9000   # OxiDB S3 uç noktası

aws --endpoint-url $E --profile oxidb s3 mb s3://belgeler          # kova oluştur
aws --endpoint-url $E --profile oxidb s3 ls                         # kovaları listele
```

Yükleme, listeleme, indirme ve silme — yani günlük iş:

```
# Yükle (içerik türü otomatik tahmin edilir)
aws --endpoint-url $E --profile oxidb s3 cp fatura.pdf s3://belgeler/faturalar/2026/07/inv-991.pdf
```

```
# Önekle listele (anahtarlar düz bir isim uzayıdır; / yalnızca bir karakterdir)
aws --endpoint-url $E --profile oxidb s3 ls s3://belgeler/faturalar/2026/07/

# Bir dizini özyinelemeli yükle (arka planda çok sayıda PutObject)
aws --endpoint-url $E --profile oxidb s3 cp ./raporlar s3://belgeler/raporlar/ --recursive

# İndir ve sil
aws --endpoint-url $E --profile oxidb s3 cp s3://belgeler/faturalar/2026/07/inv-991.pdf ./indirilen.pd
aws --endpoint-url $E --profile oxidb s3 rm s3://belgeler/faturalar/2026/07/inv-991.pdf
```

Alt seviye `s3api` komutları da geçerlidir; etiketleme ve yaşam döngüsü kuralı gibi işler buradan yapılır:

```
# Nesneye etiket bas (kova genelinde ucuz sınıflandırma)
aws --endpoint-url $E --profile oxidb s3api put-object-tagging \
    --bucket belgeler --key faturalar/2026/07/inv-991.pdf \
    --tagging 'TagSet=[{Key=ortam,Value=uretim},{Key=gizlilik,Value=yuksek}]'

# 90 gün sonra otomatik sil: yaşam döngüsü kuralı (yalnızca Expiration/Days desteklenir)
aws --endpoint-url $E --profile oxidb s3api put-bucket-lifecycle-configuration \
    --bucket gecici --lifecycle-configuration \
    '{"Rules":[{"ID":"tmp","Status":"Enabled","Expiration":{"Days":90}}]}'
```

Ham HTTP de bir seçenektir; `curl` ile imzalayarak konuşmak, protokolün gerçekten düz bir REST yüzeyi olduğunu görmenin en kısa yoludur:

```
# aws-cli'nin imzalayıcısını ödünç al (--request-signer), gövdeyi curl ile gönder
curl -X PUT --data-binary @avatar.png \
     -H "Content-Type: image/png" \
     -H "x-amz-meta-yukleyen: u-42" \
     --aws-sigv4 "aws:amz:us-east-1:s3" \
     --user "AKIAIOSFODNN7EXAMPLE:wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY" \
     "$E/avatarlar/u-42/asil.png"

# Yanıt başlığındaki ETag, içeriğin bütünlük damgasıdır:
#   ETag: "3f2a91c0d4e5b6a7889900aabbccddee"

# Yalnızca ilk 512 baytı iste (kısmi okuma → 206 Partial Content)
curl -r 0-511 --aws-sigv4 "aws:amz:us-east-1:s3" \
     --user "AKIAIOSFODNN7EXAMPLE:wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY" \
     "$E/avatarlar/u-42/asil.png" -o ilk-parca.bin
```

31.8 boto3 ile: istemcinin kurulumu ve tam bir tur

Python tarafında tek dikkat edilecek nokta, yol tarzı adresleme ve SigV4'tür:

```
import boto3, hashlib
from botocore.config import Config

s3 = boto3.client(
    "s3",
    endpoint_url="http://127.0.0.1:9000",          # OxiDB S3 uç noktası
```

```
    aws_access_key_id="AKIAIOSFODNN7EXAMPLE",
    aws_secret_access_key="wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY",
    region_name="us-east-1",
    config=Config(
        signature_version="s3v4",                    # OxiDB yalnızca SigV4 doğrular
        s3={"addressing_style": "path"},             # sanal konak tarzı YOK
    ),
)
```

Kova oluştur, içerik türü ve kullanıcı üst verisiyle yükle, damgayı doğrula:

```
s3.create_bucket(Bucket="belgeler")

with open("fatura.pdf", "rb") as f:
    icerik = f.read()

s3.put_object(
    Bucket="belgeler",
    Key="faturalar/2026/07/inv-991.pdf",
    Body=icerik,
    ContentType="application/pdf",
    Metadata={"musteri": "acme-ltd", "donem": "2026-07"},  # → x-amz-meta-*
)

# Üst veriyi indirmeden oku (HEAD: .data dosyasına hiç dokunulmaz)
head = s3.head_object(Bucket="belgeler", Key="faturalar/2026/07/inv-991.pdf")
print(head["ContentLength"], head["ContentType"], head["Metadata"])

# Bütünlük denetimi: indirilen baytların özeti sunucunun ETag'ini vermeli
obj = s3.get_object(Bucket="belgeler", Key="faturalar/2026/07/inv-991.pdf")
baytlar = obj["Body"].read()
beklenen = hashlib.sha256(baytlar).hexdigest()[:32]     # SHA-256'nın ilk 16 baytı
assert beklenen == head["ETag"].strip('"'), "nesne bozulmuş!"
```

Listeleme, önek ve ayraçla; ayraç, düz anahtar uzayına klasör görüntüsü verir:

```
# Öneke göre listele, sayfa başına en fazla 100 anahtar
r = s3.list_objects_v2(Bucket="belgeler", Prefix="faturalar/2026/", MaxKeys=100)
for o in r.get("Contents", []):
    print(o["Key"], o["Size"], o["ETag"])

# Ayraçla "klasör" görünümü: faturalar/2026/07/, faturalar/2026/08/ ...
r = s3.list_objects_v2(Bucket="belgeler", Prefix="faturalar/2026/", Delimiter="/")
for p in r.get("CommonPrefixes", []):
    print("klasör:", p["Prefix"])

# Sayfalama: NextContinuationToken'ı bir sonraki çağrıya geri ver
token = r.get("NextContinuationToken")
if token:
    r2 = s3.list_objects_v2(Bucket="belgeler", Prefix="faturalar/2026/",
                            ContinuationToken=token)
```

Kısmi ve koşullu okuma — büyük dosyalarda bant genişliği tasarrufunun tamamı buradadır:

```
# Yalnızca ilk 1 KiB'i çek: 206 Partial Content
onizleme = s3.get_object(Bucket="belgeler",
                         Key="faturalar/2026/07/inv-991.pdf",
                         Range="bytes=0-1023")["Body"].read()

# Sondan 512 bayt (sonek aralığı da desteklenir)
kuyruk = s3.get_object(Bucket="belgeler",
                       Key="faturalar/2026/07/inv-991.pdf",
                       Range="bytes=-512")["Body"].read()

# Önbellek doğrulama: elimizdeki sürüm hâlâ güncel mi?
etag = head["ETag"]
try:
    s3.get_object(Bucket="belgeler", Key="faturalar/2026/07/inv-991.pdf",
                  IfNoneMatch=etag)      # değişmediyse 304 → gövde indirilmez
except s3.exceptions.ClientError as e:
    if e.response["ResponseMetadata"]["HTTPStatusCode"] == 304:
        print("değişmemiş, yerel kopya geçerli")
```

Büyük dosyalar için çok parçalı yükleme; parçalar bağımsız yüklenir, tamamlanma anında birleştirilir:

```
mpu = s3.create_multipart_upload(Bucket="belgeler", Key="videolar/tanitim.mp4",
                                 ContentType="video/mp4",
                                 Metadata={"kaynak": "kamera-3"})
upload_id, parcalar = mpu["UploadId"], []

with open("tanitim.mp4", "rb") as f:
    n = 1
    while (chunk := f.read(8 * 1024 * 1024)):        # 8 MiB'lik parçalar
        r = s3.upload_part(Bucket="belgeler", Key="videolar/tanitim.mp4",
                           UploadId=upload_id, PartNumber=n, Body=chunk)
        parcalar.append({"ETag": r["ETag"], "PartNumber": n})
        n += 1

s3.complete_multipart_upload(Bucket="belgeler", Key="videolar/tanitim.mp4",
                             UploadId=upload_id,
                             MultipartUpload={"Parts": parcalar})
# Vazgeçilirse: s3.abort_multipart_upload(...) — yarım parçalar bellekten silinir.
# Terk edilen yüklemeler 24 saat sonra arka planda zaten toplanır.
```

Ve önceden imzalanmış URL: sunucunun gizli anahtarını istemciye vermeden, süreli bir indirme (ya da yükleme) bağlantısı üretmek:

```
# 15 dakika geçerli, doğrudan tarayıcıya verilebilecek indirme bağlantısı
url = s3.generate_presigned_url(
    "get_object",
    Params={"Bucket": "belgeler", "Key": "faturalar/2026/07/inv-991.pdf"},
    ExpiresIn=900,
)
# URL'nin tek bir karakteri bile oynatılırsa sunucu imzayı reddeder → 403.
```

31.9 Diğer diller: .NET, JavaScript, Go

.NET tarafında `OxiDb.Client.S3` paketi, AWS SDK'sının üzerine doğru varsayılanları (yol tarzı, SigV4, bölge sabitleme) geçiren ince bir sarmalayıcıdır; geri döndürdüğü şey standart bir `IAmazonS3`'tür:

```
using OxiDb.Client.S3;
using Amazon.S3.Model;

var s3 = OxiDbS3ClientFactory.Create(new OxiDbS3Options
{
    Endpoint  = "http://127.0.0.1:9000",
    AccessKey = "AKIAIOSFODNN7EXAMPLE",
    SecretKey = "wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY",
    Region    = "us-east-1",
});

await s3.PutBucketAsync("avatarlar");

await s3.PutObjectAsync(new PutObjectRequest
{
    BucketName  = "avatarlar",
    Key         = "u-42/asil.png",
    FilePath    = "avatar.png",
    ContentType = "image/png",
    // Sunucu tarafı şifreleme (OXIDB_S3_ENCRYPTION_KEY ile yönetilen anahtar)
    ServerSideEncryptionMethod = ServerSideEncryptionMethod.AES256,
});

var resp = await s3.GetObjectAsync("avatarlar", "u-42/asil.png");
await using var fs = File.Create("indirilen.png");
await resp.ResponseStream.CopyToAsync(fs);   // akış halinde indir
```

JavaScript tarafında ayrı bir paket gerekmez; resmî AWS SDK v3 doğrudan çalışır:

```
import { S3Client, PutObjectCommand, GetObjectCommand } from "@aws-sdk/client-s3";

const s3 = new S3Client({
  endpoint: "http://127.0.0.1:9000",
  region: "us-east-1",
  forcePathStyle: true,                    // OxiDB yalnızca yol tarzını bilir
  credentials: { accessKeyId: "AKIAIOSFODNN7EXAMPLE",
                 secretAccessKey: "wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY" },
});

await s3.send(new PutObjectCommand({
  Bucket: "avatarlar",
  Key: "u-42/asil.png",
  Body: dosyaBaytlari,
  ContentType: "image/png",
  Metadata: { yukleyen: "u-42" },          // → x-amz-meta-yukleyen
}));

const obj = await s3.send(new GetObjectCommand({ Bucket: "avatarlar",
```

```
                                                Key: "u-42/asil.png" }));
const baytlar = await obj.Body.transformToByteArray();
```

Go tarafında da aynı hikâye — kıyaslama koşumumuz zaten AWS SDK for Go v2 ile yazılmıştır:

```
cfg, _ := config.LoadDefaultConfig(ctx,
    config.WithRegion("us-east-1"),
    config.WithCredentialsProvider(
        credentials.NewStaticCredentialsProvider(ak, sk, "")),
)
// Yol tarzı adresleme + özel uç nokta: OxiDB ve MinIO için aynı istemci kodu
cli := s3.NewFromConfig(cfg, func(o *s3.Options) {
    o.BaseEndpoint = aws.String("http://127.0.0.1:9000")
    o.UsePathStyle = true
})

_, err := cli.PutObject(ctx, &s3.PutObjectInput{
    Bucket: aws.String("bench"),
    Key:    aws.String("dosya-0001.bin"),
    Body:   bytes.NewReader(veri),
})
```

31.10 Yerel yüzey: wire komutları

HTTP yüzeyi tek yol değildir. Belge motoruyla konuştuğunuz aynı TCP protokolü üzerinden de blob deposuna erişebilirsiniz; bu, aynı bağlantı ve aynı oturum içinde hem belge hem nesne işlemi yapmak istediğinizde kullanışlıdır. İçerik base64 olarak taşınır:

```
{"cmd": "create_bucket", "bucket": "avatarlar"}

{"cmd": "put_object", "bucket": "avatarlar", "key": "u-42/asil.png",
 "data": "iVBORw0KGgoAAAANSUhEUgAA...",
 "content_type": "image/png",
 "metadata": {"yukleyen": "u-42", "kaynak": "mobil"}}

{"cmd": "head_object", "bucket": "avatarlar", "key": "u-42/asil.png"}
{"cmd": "list_objects", "bucket": "avatarlar", "prefix": "u-42/", "limit": 50}
{"cmd": "delete_object", "bucket": "avatarlar", "key": "u-42/asil.png"}
```

Python istemcisi bu komutları yöntem olarak sarar — ve buradan, iki motorun birlikte çalıştığı asıl desene geçebiliriz.

31.11 Asıl desen: belgede üst veri, nesne deposunda baytlar

Şu kuralı bir cümlede söyleyelim: **sorguladığınız her şey belgede, sorgulamadığınız her şey nesne deposunda.** Bir fatura sisteminde belge motoru “hangi müşterinin, hangi dönemde, ne kadarlık, hangi durumda faturası var” sorusunu yanıtlar; PDF’in baytları ise nesne deposunda, belgeden yalnızca kova ve anahtarla işaret edilerek durur.

```python
import oxidb, boto3
from botocore.config import Config

db = oxidb.Client("127.0.0.1", 4444, username="app", password="...")
s3 = boto3.client("s3", endpoint_url="http://127.0.0.1:9000",
                  aws_access_key_id="app1", aws_secret_access_key="s3cret-one",
                  config=Config(signature_version="s3v4",
                                s3={"addressing_style": "path"}))

def fatura_kaydet(musteri_id, donem, tutar, pdf_baytlari):
    anahtar = f"faturalar/{donem}/{musteri_id}.pdf"

    # 1) Opak baytlar → nesne deposu. Damgayı sunucu üretir.
    meta = s3.put_object(Bucket="belgeler", Key=anahtar, Body=pdf_baytlari,
                         ContentType="application/pdf",
                         Metadata={"musteri": musteri_id, "donem": donem})

    # 2) Sorgulanabilir üst veri → belge motoru. Belge KÜÇÜK kalır.
    db.insert("faturalar", {
        "musteri_id": musteri_id,
        "donem":      donem,
        "tutar":      tutar,
        "durum":      "odenmedi",
        "dosya": {                          # baytlara REFERANS, baytların kendisi değil
            "bucket": "belgeler",
            "key":    anahtar,
            "etag":   meta["ETag"].strip('"'),
            "boyut":  len(pdf_baytlari),
        },
    })

# Sorgu tamamen belge motorunda döner; tek bir PDF baytı diskten okunmaz.
borclular = db.find("faturalar", {"durum": "odenmedi", "tutar": {"$gt": 10000}})

# Kullanıcı bir faturayı gerçekten indirmek istediğinde — ve yalnızca o zaman —
# süreli bir bağlantı üretilir; baytlar sunucu belleğinden hiç geçmez.
def indirme_baglantisi(fatura):
    d = fatura["dosya"]
    return s3.generate_presigned_url("get_object",
                                     Params={"Bucket": d["bucket"], "Key": d["key"]},
                                     ExpiresIn=900)
```

Desenin kazancı ölçülebilir: `faturalar` koleksiyonundaki belge birkaç yüz bayttır, tümüyle bellekte durur, indeksleri küçüktür ve "ödenmemiş, 10.000'den büyük" sorgusu tek bir PDF'e dokunmadan yanıtlanır. Aynı veri belgelere gömülseydi, koleksiyon gigabaytlara çıkar, her `$set` bir PDF'i yeniden yazar ve önbellek sorgulanamayan baytlarla dolardı. Avatar örneği de aynıdır: kullanıcı belgesinde `{"avatar": {"bucket": "avatarlar", "key": "u-42/asil.png", "etag": "..."}}`, gerçek PNG ise nesne deposunda.

Bir bonus daha var: yirmi üçüncü bölümdeki tam metin arama motoru, nesne deposundaki dosyaların **içinden** metin çıkarabilir. Yani PDF baytları belge motorunda değildir, ama içeriği yine de aranabilir:

```python
# Nesne deposundaki bir PDF/DOCX/HTML'den metin çıkar (aynı wire protokolü)
metin = db.extract_text("belgeler", "faturalar/2026-07/acme-ltd.pdf")
```

```
# Kova genelinde tam metin arama: TF-IDF puanıyla sıralı sonuçlar
sonuclar = db.search("gecikme faizi", bucket="belgeler", limit=10)
for s in sonuclar:
    print(s["bucket"], s["key"], s["score"])
```

31.12 MinIO ile karşılaştırma

Uyumluluk bir şeydir, performans başka. `tests/s3-benchmark-go` koşumu, aynı Go programını (AWS SDK v2) hem OxiDB'ye hem MinIO'ya karşı çalıştırır: 5.000 dosya, her biri 10–250 KB, 100 eşzamanlı goroutine, beş faz. Sonuçlar:

Faz	OxiDB	MinIO	Oran
Yükleme	1101 ms, 577 MB/s	3730 ms, 170 MB/s	3,39× OxiDB
İndirme (+MD5 doğrulama)	316 ms, 2010 MB/s	2489 ms, 255 MB/s	7,88× OxiDB
HEAD	69 ms, 73K op/s	208 ms, 24K op/s	3,03× OxiDB
Silme	329 ms, 15K op/s	321 ms, 16K op/s	0,98× (berabere)
Listeleme	6,0 ms	19,5 ms	3,25× OxiDB
Toplam	**1814 ms**	**6748 ms**	**3,72× OxiDB**

Farkın nereden geldiği tesadüf değildir ve tam da bu bölümde anlattığımız tasarım kararlarının doğrudan sonucudur. İndirmedeki büyük fark, üst verinin bellekte önbelleklenmesi ve verinin tek bir `.data` dosyasından doğrudan okunmasıdır. HEAD ve listelemedeki fark daha da nettir: bu çağrılar diske hiç gitmez, bellekteki üst veri haritasından karşılanır — `.data`/`.meta` ayrımının ödülü budur. Silmedeki beraberlik ise `.meta`'nın eşzamanlı, `.data`'nın arka planda silinmesinden gelir: istemci `.meta` kaldırılır kaldırılmaz yanıt alır. Yüklemedeki üstünlük, kilit tutulan sürenin kısalığındandır — özet alma, sıkıştırma, şifreleme ve geçici dosya yazımı kova kilidinin **dışında** yapılır; kilit yalnızca kimlik tahsisi ve nihai kayıt için, milisaniyenin altında tutulur.

Bunlar tek makinede, `tmpfs` üzerinde ölçülmüş sayılardır; MinIO'nun kümelenme, erasure coding ve sürümleme gibi yetenekleri bu ölçüme hiç girmez. Karşılaştırmanın söylediği şey şudur: uygulamanızın tipik nesne yükünü — yüz kilobaytlık dosyalar, yüksek eşzamanlılık — OxiDB'nin nesne deposu, ayrı bir nesne depolama altyapısı işletmek zorunda kalmadan, en az onun kadar iyi taşır.

31.13 Karar rehberi: ne zaman blob, ne zaman belge?

Durum	Nereye
Üzerinde sorgu, indeks ya da toplama yapılacak alanlar	Belge
Birkaç kilobayttan küçük, her okumada zaten gereken veri	Belge
Sık güncellenen alanlar (ekle-yalnızca yazma büyütmesi!)	Belge
Görüntü, video, ses, PDF, ofis dosyası, yedek arşivi	Nesne deposu
İçeriği veritabanı için opak olan her şey	Nesne deposu
Akış halinde ya da kısmi (Range) okunacak büyük veri	Nesne deposu

Durum	Nereye
Tarayıcıya doğrudan, süreli bağlantıyla verilecek dosya	Nesne deposu
Hem sorgulanacak hem büyük olan veri	İkisi: üst veri belgede, baytlar nesnede

Pratik eşik basittir: belge, birkaç yüz kilobaytı geçmeye başlıyorsa ve büyümenin kaynağı tek bir opak alansa, o alan bir nesnedir. Ve tersi de doğrudur — üç yüz baytlık bir küçük resmi (thumbnail) nesne deposuna koymak, her okumada bir HTTP gidiş-dönüşü eklemekten başka bir işe yaramaz; onu belgede tutun.

31.14 Özet

Bu bölümde, büyük ikili verinin belge motorunda neden yıkıcı olduğunu — okuma maliyeti, yazma büyütmesi ve parçalanma üçlüsünü — kurduk ve OxiDB'nin buna verdiği cevabı ayrıntılandırdık: aynı süreç içinde, ayrı bir portta yaşayan, S3 uyumlu bir nesne deposu. Nesnelerin diskte `.data` ve `.meta` olarak ayrıldığını, üst verinin bellekte önbelleklendiğini ve bunun HEAD/LIST çağrılarını diske hiç gitmeden karşıladığını; ETag'in SHA-256 tabanlı bir bütünlük damgası olduğunu; yazmanın önce `.data` sonra `.meta` sırasıyla dayanıklı hale getirildiğini gördük. S3 uyumluluğunun asıl getirisinin yazılmayan kod olduğunu — `aws-cli`, `boto3`, AWS SDK'ları ve önceden imzalı URL'lerin hiçbir uyarlama olmadan çalıştığını — örnekledik; SigV4 doğrulamasının hem başlık hem sorgu-parametresi biçimini tanıdığını, adreslemenin yalnızca yol tarzı olduğunu belirttik. MinIO'ya karşı ölçümde toplam 3,72× üstünlüğün, anlattığımız tasarım kararlarının doğrudan sonucu olduğunu gösterdik. Ve hepsinden önemlisi, iki motoru birlikte kullanma desenini kurduk: sorguladığınız her şey belgede, sorgulamadığınız her şey nesne deposunda — belgede yalnızca kova, anahtar ve damgadan oluşan bir referans.

Bölüm 32

Motorları Birlikte Kullanmak: Tek Sunucu, Beş Yüzey, Tek Uygulama

Buraya kadar OxiDB'nin yüzeylerini teker teker tanıdık. Belge motorunu Kısım III boyunca söktük taktık; ardından gelen dört bölümde ilişkisel SQL motorunu, zaman serisi motorunu, anahtar-değer katmanı OxiMem'i ve nesne depolama katmanını ayrı ayrı gördük. Her biri kendi başına anlamlıydı. Ama gerçek bir uygulama hiçbir zaman tek bir veri şeklinden ibaret değildir; kullanıcı profilleri, sipariş defterleri, fiyat akışları, fatura dosyaları ve oturum belirteçleri aynı sistemde, aynı gün içinde yan yana yaşar.

Bu bölüm, o teker teker öğrendiklerimizi tek bir çalışan sistemde buluşturur. Sorusu şudur: **hangi veri hangi motora gider ve neden?** Cevabı da bir liste değil, baştan sona kurulmuş bir uygulamadır. Bir kripto para borsası kuracağız — kitabın örnek deposundaki gerçek uygulamalara (canlı borsa ve kripto besleyici) çok benzeyen, ama burada tek bir Python istemcisiyle, tek bir TCP bağlantısı üzerinden anlatılan bir borsa. Tüm örnekler tek bir senaryonun evrimidir; kopuk parçalar değil.

32.1 Motor seçimi: veri şekli, doğru motoru söyler

Bir veriyi hangi motora koyacağınıza karar verirken sorulacak soru "hangisi daha hızlı?" değildir. Doğru soru şudur: **bu verinin doğal şekli ne, ve ona ne soracağım?** Motor seçimi, veri şeklinin bir sonucudur; bir zevk meselesi değil.

Veri şekli	Örnek	Doğru motor	Neden
Değişken şemalı, iç içe, kimlik odaklı	Kullanıcı profili, KYC kaydı, ürün kataloğu	**Belge**	Şema evrilir; kayıt tek parça okunur; iç içe alanlar bölünmeden durur
Sabit şemalı, ilişkisel, join'li, raporlanan	Emirler, işlemler, muhasebe defteri	**SQL**	Bütünlük kısıtları, join, GROUP BY, tarihsel analitik
Yüksek hacimli ölçüm akışı, zaman eksenli	Fiyat tick'leri, metrikler, sensörler	**TSDB**	Nokta başına ~0,3 bayt sıkıştırma; zaman kovaları; blok bazlı saklama süresi

Veri şekli	Örnek	Doğru motor	Neden
Büyük, opak bayt yığını	Fatura PDF'i, ürün görseli, kimlik taraması	**S3 nesne**	Belge/satır içinde taşınmaz; akışla yazılır; içeriğinden metin çıkarılabilir
Sıcak, geçici, kısa ömürlü durum	Oturum belirteci, hız sınırı sayacı, son fiyat	**OxiMem**	Mikrosaniye erişim, TTL, dayanıklılık gerektirmez

Bu tablonun altında yatan tek bir ilke vardır: **veriyi, ona soracağınız soruya göre yerleştirin.** Fiyat tick'lerini bir belge koleksiyonuna da yazabilirsiniz — çalışır. Ama günde on milyon tick'e ulaştığınızda, her tick'i bir JSON belgesi olarak saklamanın bedelini (nokta başına onlarca bayt, indeks şişmesi, "son bir saatin dakikalık mumları" sorusuna verilecek pahalı cevap) ödersiniz. Aynı veriyi TSDB'ye yazdığınızda, o soru motorun doğal dilidir.

Ters yönü de doğrudur: kullanıcı profilini SQL'e sıkıştırmaya çalışmak, her yeni alan için bir `ALTER TABLE` demektir; oysa KYC alanları ülkeye göre değişir. Değişken şema, belge motorunun var oluş sebebidir.

32.2 Aynı sunucu, ayrı isim uzayları

Bu beş yüzeyin hepsi **tek bir sunucu sürecinde** yaşar ve — belge, SQL ve TSDB söz konusu olduğunda — **tek bir TCP bağlantısı** üzerinden konuşulur. Yine de birbirlerinin ayağına basmazlar; çünkü aralarında paylaşılan hiçbir durum yoktur.

Ayrımın ilk katmanı **dosya sistemidir**. Bir veritabanının veri dizini içinde, belge koleksiyonları kendi dosyalarında, SQL motoru `sql/` altında, TSDB motoru `tsdb/` altında, bloblar `_blobs/` altında yaşar:

```
# Tek bir veritabanının (varsayılan: oxidb) disk yerleşimi.
oxidb_data/
├── _auth/                  # sunucu geneli: kullanıcılar, roller
├── _audit/                 # sunucu geneli: denetim günlüğü
└── oxidb/                  # "oxidb" veritabanı (ADR-0012: her veritabanı bir alt dizin)
    ├── users.dat           # belge motoru: koleksiyon başına append-only dosya
    ├── users.wal
    ├── _blobs/             # nesne depolama: kova/nesne
    │   └── kyc/
    ├── _fts/               # tam metin indeksi
    ├── sql/                # SQL motoru: kendi tabloları, kendi WAL'i
    └── tsdb/               # TSDB motoru: bloklar + MANIFEST + WAL
```

Bu yüzden `users` adında bir **koleksiyon** ile `users` adında bir **tablo** asla çakışmaz: farklı motorlara, farklı dosyalara, farklı isim uzaylarına aittirler. İkisi de var olabilir, ikisi de tamamen farklı veri tutabilir. Bu bir kaza değil, bir tasarım kararıdır (ADR-0010): ikinci motor, birincisinin yanına *eklenir*, içine karışmaz.

Ayrımın ikinci katmanı **tel protokolüdür**. Her istekte isteğe bağlı bir `engine` alanı vardır; yokluğunda varsayılan `"doc"`tur. Yani eski istemciler bayt bayt aynı şekilde çalışmaya devam eder:

```
{"cmd": "insert", "collection": "users", "doc": {"email": "ada@ornek.com"}}
{"engine": "sql",  "cmd": "sql",  "sql": "SELECT * FROM orders WHERE id = ?", "params": [7]}
{"engine": "tsdb", "cmd": "tsdb", "op": "query", "measurement": "ticks", "field": "price"}
```

Üç mesaj, üç motor, tek bağlantı. Sunucu, isteği okuduğu anda `engine` alanına bakar ve doğru motora yönlendirir; gerisi o motorun kendi dünyasında geçer.

OxiMem ve nesne depolama biraz farklıdır. OxiMem, Redis'in RESP protokolünü konuştuğu için kendi dinleyicisinde (`OXIDB_OXIMEM_PORT`) durur — mevcut Redis istemcileriyle konuşabilmesi için. Nesne depolama ise hem ana TCP protokolünden (`put_object` / `get_object` komutlarıyla) hem de isteğe bağlı S3-uyumlu HTTP kapısından (`OXIDB_S3_PORT`) erişilebilir.

32.3 Sunucuyu ayağa kaldırmak

Kapalı bir motorun maliyeti sıfırdır: açılmayan bir motor için ne dosya açılır, ne iş parçacığı başlatılır, ne bellek ayrılır. Bu yüzden ikinci ve üçüncü motorlar varsayılan olarak **kapalıdır**; onları açmak bilinçli bir karardır.

```
# OxiTrade borsasının sunucusu: üç motor + iki ek yüzey açık.
export OXIDB_DATA=/var/lib/oxitrade            # tüm veritabanlarının kökü
export OXIDB_ADDR=0.0.0.0:4444                 # ana TCP protokolü (belge + SQL + TSDB)

export OXIDB_SQL=1                             # ikinci motor: SQL  (ADR-0010)
export OXIDB_TSDB=1                            # üçüncü motor: zaman serisi

export OXIDB_OXIMEM_PORT=6380                  # OxiMem: RESP dinleyicisi
export OXIDB_S3_PORT=9000                      # nesne depolama: S3-uyumlu HTTP kapısı

export OXIDB_SLOW_QUERY_MS=50                  # 50 ms'yi aşan komutları profille
export OXIDB_AUDIT=true                        # denetim günlüğü

oxidb-server
```

Bu beş satırın her biri bir yüzeyi açar. `OXIDB_SQL` ve `OXIDB_TSDB` tanımlanmazsa, sunucu tam olarak eskiden olduğu gibi — sadece belge motoruyla — çalışır ve o motorların tek bir baytlık disk izi bile olmaz.

32.4 Uygulama: OxiTrade borsası

Şimdi uygulamayı kuralım. Senaryomuz bir kripto borsası: kullanıcılar kayıt olur, KYC belgelerini yükler, emir verir; borsa fiyat akışını içeri alır, mumları hesaplar ve gün sonunda rapor üretir.

Tek bir istemci nesnesi, üç motorun da kapısıdır:

```
import oxidb

db = oxidb.OxiDb(host="127.0.0.1", port=4444)
print(db.ping())                    # 'pong' — sunucu ayakta

# Bu bağlantı üç motoru da görür: belge (varsayılan), SQL, TSDB.
print(db.list_collections())        # belge motoru: koleksiyonlar
print(db.sql("SHOW TABLES"))        # SQL motoru:   tablolar
```

TSDB için Python istemcisinde henüz hazır bir sarmalayıcı yok; tel mesajını kendimiz kuracağız. Bu, aslında iyi bir şey: motorun protokolünün ne kadar basit olduğunu doğrudan görürsünüz.

```
def tsdb(db, **op):
    """TSDB motoruna tek bir istek gönderen ince yardımcı.

    Python istemcisinin ham istek/yanıt kanalını kullanır; tel biçimi
    {"engine": "tsdb", "cmd": "tsdb", "op": ...} kalıbından ibarettir.
    """
    resp = db._request({"engine": "tsdb", "cmd": "tsdb", **op})
    if not resp.get("ok"):
        raise oxidb.OxiDbError(resp.get("error", "bilinmeyen hata"))
    return resp.get("data")
```

32.4.1 Adım 1 — Kullanıcılar: belge motoru

Kullanıcı kaydı, değişken şemanın klasik örneğidir. Türkiye'deki bir kullanıcının KYC alanları ile Almanya'daki bir kullanıcınınki aynı değildir; üstelik bu alanlar zamanla değişir. Belge motoru bunu doğal karşılar:

```
db.create_collection("users")
db.create_unique_index("users", "email")     # e-posta tekil olsun
db.create_index("users", "status")           # duruma göre filtre hızlansın

db.insert("users", {
    "email": "ada@ornek.com",
    "ad": "Ada Lovelace",
    "status": "kyc_bekliyor",
    "created_at": "2026-07-14T09:00:00Z",
    "kyc": {                                  # iç içe, ülkeye göre değişken alanlar
        "ulke": "TR",
        "tc_kimlik_dogrulandi": False,
        "belgeler": []
    },
    "tercihler": {"dil": "tr", "bildirim": ["email", "push"]}
})

ada = db.find_one("users", {"email": "ada@ornek.com"})
print(ada["_id"], ada["status"])
```

Aynı koleksiyona, `kyc.ulke` alanı `"DE"` olan ve bambaşka alt alanlar taşıyan bir kullanıcı eklemek için hiçbir şema değişikliği yapmanız gerekmez. Bu esneklik, belge motorunun tam da vaadidir.

32.4.2 Adım 2 — Emirler ve defter: SQL motoru

Emirler ve muhasebe defteri ise bunun tam zıddıdır. Şemaları sabittir, sayısal bütünlükleri kritiktir, ve onlara soracağınız sorular join ve gruplama içerir. Burası SQL motorunun yeridir.

```
-- OxiTrade'in ilişkisel çekirdeği: emirler, gerçekleşen işlemler, defter.
CREATE TABLE orders (
  id        INT PRIMARY KEY,
  user_id   TEXT NOT NULL,          -- belge motorundaki kullanıcının _id'si
  symbol    TEXT NOT NULL,          -- 'BTCTRY', 'ETHTRY', ...
  side      TEXT NOT NULL,          -- 'buy' | 'sell'
```

```
  price     DOUBLE NOT NULL,
  qty       DOUBLE NOT NULL,
  status    TEXT NOT NULL,          -- 'open' | 'filled' | 'cancelled'
  created_at TIMESTAMP NOT NULL
);

CREATE TABLE trades (
  id        INT PRIMARY KEY,
  order_id  INT NOT NULL,
  price     DOUBLE NOT NULL,
  qty       DOUBLE NOT NULL,
  fee       DOUBLE NOT NULL,
  ts        TIMESTAMP NOT NULL
);

CREATE INDEX idx_orders_user   ON orders (user_id);
CREATE INDEX idx_orders_symbol ON orders (symbol);
CREATE INDEX idx_trades_order  ON trades (order_id);
```

Bu DDL'i istemciden çalıştırmak tek satırdır; `db.sql()` çağrısı, tel üzerinde `engine: "sql"` etiketli bir mesaja dönüşür:

```
DDL = open("schema.sql").read()          # yukarıdaki metnin tamamı
for stmt in DDL.split(";"):
    if stmt.strip():
        db.sql(stmt)                     # her ifade ayrı ayrı çalıştırılır

# Emir vermek: SQL motorunun kendi işlemi içinde, tek atomik birim.
db.sql("BEGIN")
db.sql(
    "INSERT INTO orders (id, user_id, symbol, side, price, qty, status, created_at) "
    "VALUES (?, ?, ?, ?, ?, ?, 'open', NOW())",
    [1001, ada["_id"], "BTCTRY", "buy", 2_450_000.0, 0.05],
)
db.sql("COMMIT")
```

Dikkat edin: `user_id` alanı, belge motorundaki kullanıcının kimliğini taşıyan bir **metin alanıdır**; bir yabancı anahtar değildir ve olamaz. Motorlar arasında referans bütünlüğü kısıtı yoktur. Kullanıcı kimliğini SQL tarafına yazarken uygulamanız, o kimliğin belge tarafında gerçekten var olduğundan emin olmak zorundadır. Bu, motorları birlikte kullanmanın ilk gerçek bedelidir ve bölümün sonunda buna döneceğiz.

32.4.3 Adım 3 — Fiyat akışı: TSDB motoru

Borsanın kalp atışı, fiyat tick'leridir. Saniyede yüzlerce, günde milyonlarca nokta. Her birini bir belge yapmak israftır; TSDB motoru bunları Gorilla sıkıştırmasıyla nokta başına yarım bayttan az yerde tutar.

```
import time

def now_ms():
    return int(time.time() * 1000)

# Borsanın eşleştiricisi her gerçekleşen işlemde bir tick yazar.
```

```
tsdb(db, op="write", points=[
    {
        "measurement": "ticks",
        "tags": {"symbol": "BTCTRY", "venue": "oxitrade"},   # seri kimliği
        "fields": {"price": 2_450_000.0, "qty": 0.05},       # sayısal alanlar
        "ts": now_ms(),                                       # epoch ms
    }
])
```

Aynı yazma, tel üzerinde şu mesajdır — istemci sadece bunu sarmalar:

```
{
  "engine": "tsdb",
  "cmd": "tsdb",
  "op": "write",
  "points": [
    {
      "measurement": "ticks",
      "tags": {"symbol": "BTCTRY", "venue": "oxitrade"},
      "fields": {"price": 2450000.0, "qty": 0.05},
      "ts": 1752480000000
    }
  ]
}
```

Dış borsalardan gelen akışı içeri almak için satır protokolü daha da pratiktir; Influx ekosisteminin standart biçimini olduğu gibi kabul ederiz:

```
# Bir dış besleyiciden gelen ham satırlar (InfluxDB line protocol).
lp = "\n".join([
    "ticks,symbol=BTCTRY,venue=binance price=2451300,qty=0.011 %d" % now_ms(),
    "ticks,symbol=ETHTRY,venue=binance price=131450,qty=0.4    %d" % now_ms(),
])
print(tsdb(db, op="write_lp", lp=lp))     # {'written': 2}
```

32.4.4 Adım 4 — Mumlar: sürekli toplama (rollup)

Bir borsanın grafiği, ham tick'lerden değil **mumlardan** çizilir. Her istekte bir milyon tick'i yeniden toplamak yerine, tamamlanmış zaman kovalarını bir kez hesaplayıp türetilmiş bir ölçüme yazarız. TSDB motorunun rollup'ları tam olarak bunu yapar; üstelik seri başına kalıcı bir su işareti tuttukları için yeniden başlatmada aynı kovayı iki kez saymazlar.

```
# 1 dakikalık mum kuralı: 'ticks' ölçümünün her sayısal serisini
# 60_000 ms'lik kovalara toplayıp 'ticks@1m' ölçümüne yaz.
tsdb(db, op="rollup_add",
     measurement="ticks",
     interval=60_000,
     label="1m",
     aggs=["first", "max", "min", "last", "sum", "count"])   # açılış/en yüksek/en düşük/kapanış...

# Kapanmış kovaları işle (bir zamanlayıcıdan periyodik olarak çağrılır).
print(tsdb(db, op="rollup_refresh"))   # {'written': N}  — sadece tamamlanmış kovalar
print(tsdb(db, op="rollups"))          # tanımlı kurallar
```

Rollup, `ticks@1m` adında türetilmiş bir ölçüm üretir; alan adları `<alan>_<toplama>` kalıbındadır — `price_first`, `price_max`, `price_min`, `price_last`. Yani klasik OHLCV mumu, doğrudan alan adlarında durur.[1]

```
# Grafik ekranı: son 6 saatin BTCTRY dakikalık kapanışları.
end = now_ms()
start = end - 6 * 60 * 60 * 1000

kapanislar = tsdb(db, op="query",
                  measurement="ticks@1m",
                  field="price_last",
                  tags={"symbol": "BTCTRY"},
                  start=start, end=end,
                  agg="last")

for seri in kapanislar:
    print(seri["tags"], len(seri["points"]))
    for p in seri["points"][:3]:
        print("  ", p["ts"], p["value"])
```

Ham tick'lere de aynı sorgu diliyle inebiliriz; `interval` verdiğinizde motor, kovaları epoch hizalı olarak kendisi keser:

```
# Anlık gösterge paneli: son 1 saatin 5 dakikalık ortalama fiyatı ve p95'i.
end, start = now_ms(), now_ms() - 60 * 60 * 1000

ort = tsdb(db, op="query", measurement="ticks", field="price",
           tags={"symbol": "BTCTRY"}, start=start, end=end,
           interval=5 * 60 * 1000, agg="mean")

p95 = tsdb(db, op="query", measurement="ticks", field="price",
           tags={"symbol": "BTCTRY"}, start=start, end=end,
           interval=5 * 60 * 1000, agg="p95")     # yüzdelikler için kısayol
```

32.4.5 Adım 5 — Belgeler ve görseller: nesne depolama

KYC için yüklenen kimlik taraması, PDF olarak gelen fatura, ürün görseli — bunların hiçbiri bir belgenin ya da satırın içinde taşınmamalıdır. Bunlar **opak bayt yığınlarıdır**; onları kovalara koyar, kayıtta yalnızca anahtarlarını tutarız.

```
db.create_bucket("kyc")
db.create_bucket("faturalar")

# Kullanıcının kimlik taramasını yükle.
with open("ada-kimlik.pdf", "rb") as f:
    icerik = f.read()

db.put_object("kyc", f"{ada['_id']}/kimlik.pdf", icerik,
              content_type="application/pdf",
              metadata={"user_id": ada["_id"], "tur": "kimlik"})
```

[1] Belge motorunun toplama işlem hattındaki `$ohlcv` aşaması da aynı işi belge tarafında yapar. Aynı fikir, iki motorun kendi diliyle ifade edilmiş halidir; hangisini kullanacağınız, tick'lerin nerede durduğuna bağlıdır.

```
# Belge tarafında SADECE referansı tut — baytları değil.
db.update_one("users", {"_id": ada["_id"]}, {
    "$push": {"kyc.belgeler": {"bucket": "kyc",
                               "key": f"{ada['_id']}/kimlik.pdf",
                               "yuklendi": "2026-07-14T09:05:00Z"}},
    "$set":  {"status": "kyc_incelemede"}
})
```

Nesne depolamanın küçük ama etkili bir hediyesi vardır: yüklenen dosyaların içeriğinden metin çıkarabilir ve o metni aranabilir kılar. Uyum ekibi, "bu kullanıcının belgelerinde şu ad geçiyor mu?" sorusunu, PDF'i indirmeden sorar:

```
# PDF'in içindeki metni çıkar (yerleşik çıkarıcılar: PDF, DOCX, HTML, XLSX...)
metin = db.extract_text("kyc", f"{ada['_id']}/kimlik.pdf")
print(metin[:200])

# Tüm kova genelinde tam metin araması.
for hit in db.search("Lovelace", bucket="kyc", limit=5):
    print(hit["bucket"], hit["key"], hit["score"])
```

32.4.6 Adım 6 — Sıcak durum: OxiMem

Geriye bir tür veri kaldı: **kısa ömürlü, sıcak, kaybolması dert olmayan** durum. Oturum belirteçleri, saniyedeki istek sayacı, ekranda gösterilen son fiyat. Bunlar için diske yazan bir motor kullanmak, çiviyi buldozerle çakmaktır. OxiMem, RESP protokolünü konuştuğu için standart bir Redis istemcisiyle konuşulur:

```
import redis

mem = redis.Redis(host="127.0.0.1", port=6380, decode_responses=True)
mem.ping()

# 1) Oturum: giriş yapan kullanıcı için 30 dakikalık belirteç.
import secrets
token = secrets.token_urlsafe(24)
mem.setex(f"session:{token}", 1800, ada["_id"])     # TTL ile kendiliğinden silinir

# 2) Hız sınırı: kullanıcı başına dakikada en çok 60 emir.
def emir_verebilir_mi(user_id: str) -> bool:
    anahtar = f"rate:{user_id}:{int(time.time() // 60)}"
    sayac = mem.incr(anahtar)
    if sayac == 1:
        mem.expire(anahtar, 120)        # pencere geçince kendi kendine yok olsun
    return sayac <= 60
```

Sıcak fiyat önbelleği, motorların birlikte çalışmasının en güzel örneğidir: tick akışı TSDB'ye **kalıcı** olarak yazılırken, aynı tick'in son değeri OxiMem'e **geçici** olarak yazılır. Web sayfası, saniyede binlerce kez "BTCTRY kaç?" diye sorduğunda cevabı TSDB'den değil, bellekten alır:

```
def tick_isle(symbol: str, price: float, qty: float):
    """Bir tick geldiğinde: kalıcı seriye yaz, sıcak önbelleği tazele."""
    ts = now_ms()
```

```
    # (a) Kalıcı kayıt — kaynak-doğruluk burada.
    tsdb(db, op="write", points=[{
        "measurement": "ticks",
        "tags": {"symbol": symbol, "venue": "oxitrade"},
        "fields": {"price": price, "qty": qty},
        "ts": ts,
    }])

    # (b) Sıcak önbellek — kaybolursa (a)'dan yeniden üretilebilir.
    mem.hset(f"last:{symbol}", mapping={"price": price, "ts": ts})
    mem.expire(f"last:{symbol}", 300)

def son_fiyat(symbol: str):
    """Önce bellekten sor; yoksa kalıcı seriden tazele (cache-aside)."""
    h = mem.hgetall(f"last:{symbol}")
    if h:
        return float(h["price"])
    seri = tsdb(db, op="query", measurement="ticks", field="price",
                tags={"symbol": symbol},
                start=now_ms() - 60_000, end=now_ms(), agg="last")
    if not seri or not seri[0]["points"]:
        return None
    fiyat = seri[0]["points"][-1]["value"]
    mem.hset(f"last:{symbol}", mapping={"price": fiyat, "ts": now_ms()})
    return fiyat
```

Buradaki hiyerarşi, on üçüncü bölümde tanıdığımız bellek–önbellek–disk piramidinin uygulama düzeyindeki yankısıdır: **kaynak-doğruluk kalıcı motordadır, önbellek yalnızca bir hızlandırıcıdır.** OxiMem'i kaybetseniz uygulama yavaşlar ama yanlış cevap vermez. Bu ayrımı korumak, çok motorlu bir sistemi ayakta tutan disiplinlerin en önemlisidir.

32.4.7 Adım 7 — Raporlama: SQL'in evinde

Gün sonu raporu, tam olarak SQL'in var olma sebebidir: iki tabloyu birleştir, grupla, sırala.

```
-- Sembol bazında günlük hacim, komisyon geliri ve ortalama işlem büyüklüğü.
SELECT o.symbol,
       COUNT(*)                    AS islem_sayisi,
       SUM(t.qty * t.price)        AS hacim,
       SUM(t.fee)                  AS komisyon,
       AVG(t.qty * t.price)        AS ortalama_islem,
       COUNT(DISTINCT o.user_id)   AS aktif_kullanici
FROM trades t
JOIN orders o ON o.id = t.order_id
WHERE t.ts >= date_trunc('day', NOW())
GROUP BY o.symbol
HAVING SUM(t.qty * t.price) > 0
ORDER BY hacim DESC;
```

```
[rapor] = db.sql(open("gunluk_rapor.sql").read())
print(rapor["columns"])                    # ['symbol', 'islem_sayisi', 'hacim', ...]
```

```
for satir in rapor["rows"]:
    print(satir)

# Raporun kendisi bir belgedir: geçmişi belge motorunda saklayalım.
db.insert("raporlar", {
    "tur": "gunluk_hacim",
    "tarih": "2026-07-14",
    "kolonlar": rapor["columns"],
    "satirlar": rapor["rows"],          # iç içe dizi — belge motoru için doğal
})
```

Şu son on satır, bölümün ana fikrinin küçük bir özetidir: rapor **SQL'de** hesaplanır (çünkü join ve gruplama oranın işidir), ama **belge motorunda** saklanır (çünkü bir raporun şekli değişkendir ve tek parça okunur). Her veri, kendi sorusunun evine gider.

32.5 Sınırlar: motorlar arası atomiklik yoktur

Şimdi dürüst konuşma vakti. OxiDB'nin üç motoru, tasarım gereği hiçbir durumu paylaşmaz. Bunun bir güzel sonucu vardır (birbirlerini yavaşlatmazlar, birbirlerinin hatasından etkilenmezler) ve bir de sert sonucu: **bir belge koleksiyonu ile bir SQL tablosunu tek bir atomik işlemde güncelleyemezsiniz.**

Her motorun kendi içinde tam atomikliği vardır — belge motorunun üç fazlı OCC commit'i, SQL motorunun tek fsync'lik toplu WAL kaydı. Ama iki WAL'i kapsayan bir commit protokolü **henüz yoktur**. ADR-0011 bunu tasarlar (paylaşılan bir commit saati ve iki fazlı bir commit ile), ama durumu "Önerildi"dir: tasarım var, uygulama yok. Kitabın yazıldığı sürümde, iki motora yazan bir kod parçası çöktüğünde birinin uygulanmış, diğerinin uygulanmamış olması mümkündür.

Bu, çözülemez bir sorun değildir; ama **görmezden gelinemez** bir sorundur. Üç telafi deseni yeter:

1. Kaynak-doğruluk motorunu seçin. Her iş gerçeğinin tek bir sahibi olsun. Emrin gerçeği SQL'dedir; kullanıcının gerçeği belge motorundadır; fiyatın gerçeği TSDB'dedir. Diğer motorlardaki kopyalar *türev*dir ve kaynaktan yeniden üretilebilir olmalıdır.

2. Yazma sırasını, kaynak önce gelecek şekilde kurun ve idempotent yazın. Emir SQL'e yazılır (gerçek budur); ardından belge motoruna bir denetim izi düşülür (türev budur). Aradaki çökme, sadece bir izin eksik kalması demektir — düzeltilebilir bir durum. Ters sıra ise "olmayan bir emrin izi" üretirdi.

```
def emir_ver(user_id: str, symbol: str, side: str, price: float, qty: float,
             istemci_ref: str):
    """İki motora yazan bir iş: kaynak SQL, türev belge. Idempotent."""
    if not emir_verebilir_mi(user_id):                # OxiMem: hız sınırı
        raise RuntimeError("hız sınırı aşıldı")

    # (1) KAYNAK-DOĞRULUK: emrin kendisi SQL motorunda, kendi işlemi içinde.
    db.sql("BEGIN")
    try:
        db.sql(
            "INSERT INTO orders (id, user_id, symbol, side, price, qty, status, created_at) "
            "VALUES (?, ?, ?, ?, ?, ?, 'open', NOW())",
            [siradaki_id(), user_id, symbol, side, price, qty],
        )
        db.sql("COMMIT")
    except Exception:
        db.sql("ROLLBACK")
```

```
        raise

    # (2) TÜREV: denetim izi belge motorunda. Burada çökersek emir yine geçerlidir;
    #     iz eksik kalır ve mutabakat işi onu tamamlar. Aynı istemci referansıyla
    #     ikinci kez yazılırsa üzerine yazar (idempotent) — kopya üretmez.
    db.update_one("audit_orders", {"ref": istemci_ref}, {
        "$set": {"ref": istemci_ref, "user_id": user_id, "symbol": symbol,
                 "side": side, "price": price, "qty": qty,
                 "kaynak": "sql:orders", "ts": now_ms()}
    })
```

3. Nihai mutabakat (reconciliation) işleri yazın. Türev tarafın kaynaktan sapıp sapmadığını periyodik olarak kontrol edin ve düzeltin. Bu, on birinci bölümdeki nihai tutarlılık fikrinin uygulama düzeyindeki karşılığıdır:

```
def mutabakat():
    """SQL'deki emirlerle belge motorundaki denetim izlerini karşılaştır.
    Eksik izleri tamamla. Günde bir kez, ya da bir zamanlayıcıyla çalışır."""
    [r] = db.sql(
        "SELECT id, user_id, symbol, side, price, qty FROM orders "
        "WHERE created_at >= NOW() - INTERVAL '1 day'"
    )
    for oid, uid, sym, side, price, qty in r["rows"]:
        ref = f"order:{oid}"
        if db.find_one("audit_orders", {"ref": ref}) is None:
            # Kaynakta var, türevde yok → türevi tamamla (yeniden üretilebilirlik).
            db.insert("audit_orders", {"ref": ref, "user_id": uid, "symbol": sym,
                                       "side": side, "price": price, "qty": qty,
                                       "kaynak": "sql:orders", "onarildi": True})
```

Aynı mantık, nesne depolamaya yüklenen bir dosya ile onun belge tarafındaki referansı arasında da geçerlidir: **önce nesneyi yükleyin, sonra referansı yazın.** Ters sıra, var olmayan bir dosyaya işaret eden bir kayıt üretir — ki bu, sahipsiz kalmış bir nesneden çok daha kötüdür. Sahipsiz nesneler bir temizlik işiyle toplanabilir; kırık referanslar kullanıcıya hata olarak döner.

Kural cümlesi şudur: **çökme, en fazla bir "fazlalık" bıraksın, asla bir "eksiklik" bırakmasın.** Fazlalık toplanabilir; eksiklik, veri kaybıdır.

32.6 İşletim: tek sunucu, ayrı ömürler

Motorlar durumu paylaşmadığı için ömürleri de ayrıdır ve bu, işletmeyi hem kolaylaştırır hem de bir tuzak taşır.

Kolaylaştırır: her motorun bakım işi kendine aittir. TSDB'nin saklama süresini uygularsınız, blok bazlı olduğu için süresi dolmuş bloklar bütünüyle düşer; belge motorunun sıkıştırmasını çalıştırırsınız, silinmiş kayıtların yeri geri alınır. Biri diğerini beklemez.

```
# Gecelik bakım: her motor kendi işini görür.
kesim = now_ms() - 90 * 24 * 60 * 60 * 1000     # 90 günden eski tick'ler
print(tsdb(db, op="retention", cutoff=kesim))   # {'removed': N} — blok bazında düşer
print(tsdb(db, op="rollup_refresh"))            # kapanan mumları işle
print(tsdb(db, op="checkpoint"))                # bloklar + MANIFEST'i diske sabitle
```

```
print(tsdb(db, op="stats"))                    # {'series':.., 'points':.., 'bytes':..}

print(db.compact("audit_orders"))              # belge motoru: yeri geri kazan
```

Tuzak da tam burada: **yedekleme kapsamı**. Belge motorunun `backup()` çağrısı, belge motorunun dosyalarını yedekler; SQL motorunun WAL'ini ve TSDB'nin bloklarını değil. Üç motorlu bir sistemi yedeklemenin doğru yolu, veri dizininin tamamını — tutarlı bir kesitte — almaktır:

```
# Doğru yedekleme kapsamı: veri dizininin TAMAMI, tek bir kesitte.
# (Motorların WAL'leri ayrıdır; birini alıp diğerini bırakmak tutarsız bir
#  yedek üretir: SQL'de var olan bir emrin belgesi eksik kalabilir.)
oxidb-cli tsdb checkpoint                      # TSDB'yi sabit bir noktaya çek
rsync -a --delete /var/lib/oxitrade/ /yedek/oxitrade-$(date +%F)/

# Zamanın bir noktasına dönüş (PITR) yalnızca belge motorunu kapsar:
export OXIDB_PITR=1                            # açıksa arşivleyici sealed WAL'leri kopyalar
```

Son bir işletim notu: kapalı motorun sıfır maliyeti, geri dönüşü olan bir karardır. `OXIDB_TSDB` olmadan başlatılan bir sunucu, TSDB dizinini hiç oluşturmaz; sonradan açtığınızda motor boş bir durumla doğar. Yani üçüncü motoru, ihtiyacınız olduğu gün — mevcut verinize dokunmadan — açabilirsiniz. Bu, "her şeyi baştan doğru seçmek zorundasınız" baskısını ortadan kaldırır; motor seçimi, uygulamanızla birlikte evrilebilir.

32.7 Bu bölümün bıraktığı yer

Kitabın başında bir soru sormuştuk: veri neye benzer, ve onu nasıl saklamalıyız? Kısım I bu soruyu kuramsal olarak, Kısım II belge modelinin gözünden, Kısım III de OxiDB'nin somut motorlarında yanıtladı. Bu bölüm, o yanıtların hepsini tek bir çalışan sistemde bir araya getirdi: değişken şemalı kullanıcı belge motorunda, sabit şemalı emir SQL'de, milyonlarca tick sıkıştırılmış zaman serisinde, PDF nesne kovasında, oturum belleğin içinde — hepsi tek bir sunucu sürecinde, çoğu tek bir TCP bağlantısında.

Bu birlikteliğin sırrı sofistike bir entegrasyon katmanı değil, tam tersidir: motorların **birbirine hiç dokunmamasıdır**. Paylaşılmayan durum, bağımsız ömür, ayrı dosyalar. Bedeli de açıktır ve saklamadık: motorlar arası atomik işlem yoktur; onu, kaynak-doğruluk seçimi, idempotent yazma ve nihai mutabakat ile uygulama katmanında telafi edersiniz. ADR-0011 o boşluğu kapatmayı önerir; o gün geldiğinde, bu bölümün telafi desenlerinin çoğu gereksizleşecek — ama seçim tablosunun tek bir satırı bile değişmeyecektir. Çünkü hangi verinin hangi motora gideceği sorusu, bir commit protokolünün değil, verinin şeklinin sorusudur.

Ve bu, aslında bütün kitabın tek cümlelik özetidir: **veriyi, ona soracağınız soruya benzeyen yere koyun.** Gerisi mühendisliktir.

Ek A — Sözlük

Bu sözlük, kitap boyunca kullanılan başlıca terimleri kısaca tanımlar. Tanımlar, terimin ilk geçtiği bölümdeki ayrıntılı anlatımın yerini tutmaz; hızlı bir hatırlatma olarak düşünülmüştür.

ACID Bir işlemin verdiği dört güvencenin kısaltması: atomiklik, tutarlılık, yalıtım ve dayanıklılık.

Açma (unwind) Bir toplama aşaması; içinde liste barındıran bir belgeyi, listenin her öğesi için bir tane olmak üzere birçok belgeye genişletir.

AEAD (kimliği doğrulanmış şifreleme) Gizliliği ve bütünlüğü birlikte sağlayan; çözerken bir doğrulama etiketiyle veriyi kurcalamaya karşı koruyan şifreleme kipi (örneğin AES- GCM).

Anahtar-değer modeli Veriyi bir anahtardan opak bir değere giden eşleme olarak tutan, en yalın veri modeli.

Anlık görüntü yalıtımı (snapshot isolation) Her işlemin, başladığı andaki tutarlı bir veri anlık görüntüsünü gördüğü MVCC tabanlı yalıtım düzeyi; serileştirilebilir değildir (yazma eğriltmesine açıktır).

Append-only (yalnızca-ekleme) Var olan veriyi yerinde değiştirmeyen, her yazmayı dosyanın sonuna ekleyen depolama yaklaşımı; ardışık yazma hızlıdır, ölü alan biriktirir.

ARIES Yazma-öncesi günlüğe dayanan klasik kurtarma yöntemi; analiz, yineleme (redo) ve geri-alma (undo) fazlarıyla çökme sonrası tutarlılığı sağlar.

Atlama listesi (skip list) Sıralı bağlı listeye olasılıksal "ekspres şeritler" ekleyerek logaritmik aramayı kilitsiz biçimde sağlayan, dengeli ağaçlara olasılıksal bir alternatif.

Atomiklik Bir işlemin tüm adımlarının ya hep birlikte gerçekleşmesi ya da hiç gerçekleşmemesi güvencesi.

Ayrıştırma (parsing) Ham bir sorguyu, sistemin üzerinde işlem yapabileceği yapısal bir koşul ağacına çevirme adımı.

B-ağacı Veriyi sıralı ve dengeli tutan, hem tekil hem aralık aramalarında verimli bir ağaç yapısı; sayfa tabanlı depolama motorlarının ve sıralı indekslerin temelidir.

B+ağacı Verinin yalnızca yapraklarda tutulduğu, yaprakların bağlı liste oluşturduğu B-ağacı türevi; aralık taramalarını hızlandırır.

Bayt düzeyinde süzme OxiDB'nin, bir belgeyi koşula uyup uymadığını anlamak için onu nesneye çevirmeden, kodlanmış baytları üzerinde denetleyen ve eşleşmeyenleri hiç çözmeden eleyen tekniği.

Belge (doküman) Alanlardan oluşan, değerleri skaler, liste ya da iç içe belge olabilen, kendini tanımlayan veri birimi.

Bellek-zoru fonksiyon (memory-hard) Kasıtlı olarak çok bellek isteyen, böylece özel donanımla kaba kuvvet saldırısını pahalılaştıran parola özetleme yaklaşımı (scrypt, Argon2).

Bileşik indeks Birden çok alanın birleşimi üzerine kurulu indeks; alan sırasının baştan başlayan bir önekini kullanan sorgulara yarar (önek kuralı).

Biriktirici (accumulator) Gruplama sırasında her grup için bir özet değer hesaplayan işlem; sayma, toplama, ortalama, en büyük, en küçük gibi.

Birleştirme algoritmaları (join) İki veri kümesini eşleştirme yöntemleri; iç içe döngü, hash birleştirmesi ve sırala-birleştir, farklı boyut ve sıralılık koşullarında üstün gelir.

Bloom filtresi Bir öğenin kümede "kesinlikle yok" ya da "olabilir" olduğunu az yer kullanarak söyleyen olasılıksal yapı; LSM okuma yolunda gereksiz disk erişimini önler.

Boyut-katmanlı sıkıştırma (size-tiered) LSM'de benzer boyuttaki parçaları birleştiren; yazma büyütmesini düşük, ama yer ve okuma büyütmesini yüksek tutan birleştirme stratejisi.

CAP Ağ bölündüğünde, bir dağıtık sistemin tutarlılık ile erişilebilirlik arasında seçim yapmak zorunda olduğunu ifade eden içgörü.

Çalışma kümesi (working set) Verinin, herhangi bir anda etkin biçimde kullanılan sıcak kısmı; belleğe sığıp sığmaması performansı belirler.

Çift yazma (double-write) Yarım yazma riskine karşı, sayfayı önce ayrı bir tampona, sonra asıl yerine yazan koruma; append-only tasarımlar buna gerek duymaz.

Çoğunluk (quorum) Bir kararın geçerli sayılması için gereken düğüm çoğunluğu; herhangi iki çoğunluğun kesişmesi, konsensüsün güvenliğini sağlar.

Dağıt-topla (scatter-gather) Sharding'de, parça anahtarı içermeyen bir isteğin tüm parçalara gönderilip kısmi yanıtların birleştirilmesi örüntüsü.

Dayanıklılık (durability) Bir işlem tamamlandı dendikten sonra, hiçbir çökmenin onu geri alamaması güvencesi.

Denetim (audit) "Kim, ne zaman, ne yaptı" sorusunu yanıtlayan, hesap verebilirlik için tutulan kayıt.

Denetim noktası (checkpoint) Yazma-öncesi günlüğün, değişiklikler asıl depoya güvenle yansıdıktan sonra kısaltıldığı senkronizasyon anı.

Disk-öncelikli (disk-first) OxiDB'nin, bellekte yalnızca kompakt bir kimlik-konum dizini tutup belge gövdelerini diske bırakan, bellek-tutumlu depolama kipi.

Doğrusallaştırılabilirlik (linearizability) Her işlemin, çağrısıyla dönüşü arasında tek bir anda gerçekleşmiş gibi göründüğü, gerçek-zaman sırasına saygılı en güçlü tutarlılık.

Dolum oranı (fill factor) Bir B-ağacı düğümünün ne kadarının dolu olduğu; düşük doluluk yer israfı, yüksek doluluk sık bölünme demektir.

Erken sonlanma Bir sorgunun, gereken sonuca ulaştığı an durup geri kalanı üretmemesi; sıralı indeksli "ilk N" sorgularında ve tekil işlemlerde kullanılır.

Eşzamansız replikasyon Liderin, bir yazmayı takipçilere iletmeyi beklemeden onayladığı, hızlı ama kayıp riski taşıyan replikasyon biçimi.

Failover (devralma) Lider çöktüğünde, bir takipçinin yeni lider olarak yükseltilmesi süreci.

Fanout (çıkış genişliği) Bir B-ağacı düğümünün çocuk sayısı; yüksek fanout ağacı sığlaştırır ve aramada gereken disk erişimini azaltır.

fdatasync fsync'in, yalnızca veriyi diske işleyip üstveri güncellemesini atlayabilen, biraz daha hafif türevi.

Fence pointer (sınır işaretçisi) Bir SSTable içinde aranan anahtara hızlıca atlamayı sağlayan seyrek indeks.

FFI Bir dilde yazılmış çekirdeği, C uyumlu bir arayüz üzerinden başka dillerden çağrılabilir kılan köprü.

FLP imkânsızlığı Tümüyle eşzamansız bir ağda tek bir düğüm bile sessizce durabiliyorsa, konsensüsün her durumda sonlanmasının garanti edilemeyeceğini gösteren sonuç.

fsync Yazılmış verinin gerçekten kalıcı ortama işlendiğinden emin olmak için verilen, güçlü ama yavaş boşaltma emri.

Gömme (embedding) İlişkili veriyi, ona ait olduğu belgenin içine yerleştirme; yerellik kazandırır, çoğaltma ve sınırsız büyüme riski getirir.

Gönderim listesi (posting list) Ters indekste bir sözcüğün altında tutulan; o sözcüğü içeren belgelerin ve sıklık/konum bilgisinin listesi.

Grup commit (group commit) Birçok işlemin tamamlamasını tek bir fsync'te toplayarak dayanıklılık maliyetini paylaştıran teknik; gecikme karşılığında iş hacmi kazandırır.

Gruplama Belgeleri bir anahtara göre gruplara ayırıp her grup için biriktiricilerle özet hesaplayan, satırları çökerten toplama işlemi.

GSN (küresel sıra numarası) Zaman-noktasına kurtarmada tüm yazmalara verilen, koleksiyonlar arası tek ve monoton artan sıralama ekseni.

İki fazlı kilitleme (2PL) İşlemin önce yalnızca kilit aldığı (büyüme), sonra yalnızca bıraktığı (küçülme) iki fazla serileştirilebilirliği sağlayan kilit protokolü.

İndeks Bir alanın değerlerinden belgelerin konumuna giden, aramayı hızlandıran, veriden türetilmiş yardımcı yapı.

İşlem (transaction) Birçok okuma-yazma adımını tek bir bölünmez birim olarak ele alan, "ya hep ya hiç" güvencesi veren kavram.

İyimser eşzamanlılık denetimi (OCC) Çatışmaların nadir olduğunu varsayan, kilit almadan çalışıp tamamlama anında sürüm doğrulayan ve çatışmada iptal eden yaklaşım; OxiDB'nin kullandığı model.

JSON JavaScript nesne gösteriminden türeyen, dilden bağımsız, metinsel veri değiş-tokuş biçimi; belge yapısının yazıya dökülmüş hâli.

Kapsayan indeks (covering) Bir sorgunun ihtiyaç duyduğu her şeyi tek başına sağlayan ve böylece belgeye hiç dokunmadan yanıt üretilmesini mümkün kılan indeks durumu.

Kararlı sıralama (stable sort) Eşit anahtarlı öğelerin göreli sırasını koruyan sıralama; pencere fonksiyonlarında doğru sonuç için gereklidir.

Kardinalite tahmini Bir koşulun kaç belgeyle eşleşeceğinin önceden kestirilmesi; sorgu eniyileyicinin indeks ve plan seçiminin temelidir.

Kilitlenme (deadlock) İki işlemin her birinin, diğerinin tuttuğu kaynağı beklemesiyle oluşan, çözülmediğinde sonsuza dek süren döngüsel bekleme.

Kirli sayfa tablosu (dirty page table) ARIES kurtarmasında, hangi değişikliklerin henüz diske yansımadığını izleyen ve yinelemenin nereden başlayacağını belirleyen yapı.

Koleksiyon Belgelerin gruplandığı birim; ilişkisel tablonun karşılığıdır ama belgelerin aynı biçimde olmasını zorunlu kılmaz.

Konsensüs Bir grup makinenin, bazıları çökse bile ortak bir karar üzerinde güvenle anlaşmasını sağlayan, çoğunluk oylamasına dayanan mekanizma.

Kurtarma (recovery) Çökme sonrası, yazma-öncesi günlüğü oynatarak veritabanını tutarlı bir duruma getiren süreç.

LSM ağacı Yazmaları bellekte biriktirip sıralı parçalar halinde diske yazan ve bunları arka planda birleştiren, yazma-yoğun log-yapılı depolama tasarımı.

LSN (günlük sıra numarası) Her WAL kaydına verilen artan kimlik; sayfa-LSN'iyle kıyaslanarak bir değişikliğin uygulanıp uygulanmadığı anlaşılır (idempotent yineleme).

Maliyet modeli (cost model) Sorgu eniyileyicinin, alternatif planların disk ve CPU maliyetini tahmin edip en ucuzunu seçmesini sağlayan model.

memtable LSM'de yazmaların önce biriktiği bellekteki sıralı yapı; dolunca diske bir SSTable olarak boşaltılır.

mmap (belleğe yansıtma) Bir dosyayı belleğin bir parçasıymış gibi erişilebilir kılan ve önbellek yönetimini büyük ölçüde işletim sistemine devreden teknik.

MVCC Verinin birden çok sürümünü tutarak okuyucuların tutarlı bir anlık görüntü görmesini, okuyucu ile yazıcının birbirini beklememesini sağlayan yaklaşım.

Nihai tutarlılık Kopyaların geçici olarak anlaşmazlığa düşmesine izin veren ama yazmalar durursa aynı değere yakınsamayı garanti eden tutarlılık biçimi.

Normalleştirme Her bilgiyi tek bir yerde tutarak çoğaltmayı önleme; ilişkisel modelin güncellemeyi ucuzlatan ilkesi.

Okuma anlık görüntüsü (read snapshot) OxiDB'nin, yalnızca okuma yolunu değiştiren hafif çoksürümlü (MVCC) yaklaşımı; bir okuma, başladığı andaki tutarlı durumu görür ve süregelen yazmalardan etkilenmez. Toplama varsayılan olarak bu sayede anlık-görüntü tutarlıdır; okuyucular yazarları, yazarlar okuyucuları bekletmez.

OxiPool OxiDB'nin sharding (parçalama) katmanı; bir koleksiyonu parça anahtarına göre birden çok bağımsız OxiDB düğümüne dağıtan ve sorguları dağıt-topla yöntemiyle yanıtlayan ön yüz.

OxiWire OxiDB'nin sunucu iletişiminde kullandığı, uzunluk önekli, JSON ve daha hızlı bir ikili biçimi destekleyen tel protokolü.

Ölü alan Append-only depolamada, güncellenen ya da silinen kayıtların geride bıraktığı, artık geçerli olmayan veri; sıkıştırmayla geri kazanılır.

Önce-olur ilişkisi (happened-before) Dağıtık olaylar arasında nedensel sıralamayı tanımlayan; mantıksal ve vektör saatlerinin temelindeki bağıntı.

Önek kuralı Bir bileşik indeksin, ancak alan sırasının baştan başlayan bir önekini kullanan sorgulara yaradığı kural.

Paxos Çoğunluk anlaşmasıyla konsensüs sağlayan, Raft'tan önceki temel protokol; iki aşamalı (söz iste, sonra öner) bir yapıya dayanır.

Pencere fonksiyonu Her belgeyi koruyarak, ona komşu belgelerden oluşan bir pencereye dayalı bir değer ekleyen analitik işlem; kümülatif toplam, hareketli ortalama, sıralama gibi.

PITR (zaman-noktasına kurtarma) Veritabanını yalnızca son tutarlı duruma değil, geçmişteki belirli bir ana geri döndürebilme yeteneği.

Pipeline Toplamanın, her biri akışı dönüştürüp bir sonrakine veren aşamalardan oluşan, bileşilebilir modeli.

RBAC (rol tabanlı erişim denetimi) Yetkileri rollerde gruplayıp kullanıcılara rol atayarak yetkilendirmeyi yöneten model.

Replikasyon Aynı veriyi birden çok makinede kopya halinde tutma; dayanıklılık, erişilebilirlik, okuma ölçeği ve gecikme için yapılır.

Sağlama toplamı (checksum) Bir kaydın içeriğinden hesaplanan, yarım kalmış ya da bozulmuş kayıtları yakalamaya yarayan doğrulama değeri.

Sayfa Diskten tek seferde okunup yazılan, sabit boyutlu blok; sayfa tabanlı depolama motorlarının temel birimi.

Sayfa hatası (page fault) Belleğe yansıtılmış ama o an fiziksel bellekte bulunmayan bir sayfaya erişildiğinde, işletim sisteminin sayfayı diskten getirmesi.

SCRAM Parolayı hattan hiç geçirmeden, çentik (nonce) ve tuzlu özet temelli bir meydan-okuma-yanıt ile kimlik doğrulayan mekanizma (OxiDB'de SCRAM- SHA-256).

Seçicilik Bir indeks alanının değerlerinin ne kadar az belgeyle eşleştiği; yüksek seçicilik, indeksi daha yararlı kılar.

Serileştirilebilir anlık görüntü yalıtımı (SSI) Anlık görüntü yalıtımına, tehlikeli okuma-yazma bağımlılıklarını saptayıp işlemi iptal eden bir denetim ekleyerek tam serileştirilebilirliği sağlayan yöntem.

Seviyeli sıkıştırma (leveled) LSM'de parçaları örtüşmeyen seviyelere ayıran; okuma ve yer büyütmesini düşük tutan, ama daha çok yazma büyütmesi getiren birleştirme stratejisi.

Sharding (parçalama) Veri kümesini makineler arasında bölerek kapasiteyi ve yazma hacmini ölçeklendirme tekniği.

Sıfır-kopya Sıkıştırılmamış ve şifrelenmemiş veriye, belleğe yansıtılmış dosyadan hiç kopyalamadan ve çözmeden doğrudan erişebilme.

Sıkıştırma (compaction) Append-only bir veri dosyasını baştan yazıp yalnızca yaşayan kayıtları tutarak ölü alanı geri kazanan bakım işlemi.

Split-brain (ikiye bölünmüş beyin) Bir ağ bölünmesinde, aynı anda iki liderin ortaya çıkıp veriyi çelişkili iki gerçeğe ayırması tehlikesi; çoğunluk mutabakatı bunu önler.

SSTable LSM'de diske yazılan, sıralı ve değişmez veri parçası (sorted string table).

Sürüm numarası Her belgeye iliştirilen, her değişiklikte artan sayaç; iyimser eşzamanlılık denetiminde çatışmayı saptamaya yarar.

Sürüm zinciri (version chain) MVCC'de bir belgenin ardışık sürümlerinin, okuyucuların doğru anlık görüntüyü bulabilmesi için birbirine bağlandığı yapı.

Şema-okumada / şema-yazmada Şemanın okuma anında (uygulamada, örtük) mı yoksa yazma anında (veritabanında, açık) mı dayatıldığını ayıran kavramlar.

Tahliye (eviction) Dolu bir önbellekte, yeni bir şeye yer açmak için bir öğenin atılması; yaygın bir politika "en uzun süredir kullanılmayanı at" (LRU) ilkesidir.

Tam tarama (full scan) Aranan kaydı bulmak için tüm belgeleri tek tek okuma; uygun bir indeks yoksa başvurulan, maliyetli yol.

Tampon havuzu (buffer pool) Veritabanının disk sayfalarını bellekte önbelleklediği; kirli sayfaları WAL kuralına göre diske yazan yönetilen havuz.

Ters indeks (inverted index) "Sözcük, o sözcüğü içeren belgeler" eşlemesi; metin içinde sözcük aramayı ve alaka puanlamasını mümkün kılar.

Tutarlı hash (consistent hashing) Düğüm eklenip çıktığında anahtarların yalnızca küçük bir kısmının yer değiştirmesini sağlayan, bir halka üzerine yerleştirmeye dayanan dağıtım yöntemi.

Tutarlılık (tek makine / dağıtık) Tek makinede, verinin geçerli kurallara uyması; dağıtık sistemde, kopyaların ne ölçüde aynı değeri gösterdiği (güçlüden nihaiye uzanan bir tayf).

Vektör saatleri (vector clocks) Dağıtık olaylar arasındaki neden-sonuç ilişkisini ve eşzamanlılığı saptayan, düğüm başına bir sayaç dizisi.

Veri modeli Verinin nasıl yapılandırıldığını, ilişkilerin nasıl ifade edildiğini ve hangi işlemlerle erişildiğini belirleyen temel karar.

WAL (yazma-öncesi günlük) Asıl veriyi değiştirmeden önce niyeti kaydedip dayanıklı kılan, dayanıklılık ve çökme güvenliğinin temelindeki günlük.

Yalıtım (isolation) Aynı anda çalışan işlemlerin birbirini etkilememesi; sonucun, işlemler sırayla çalışmış gibi olması güvencesi.

Yarı-eşzamanlı replikasyon Liderin, bir yazmayı en az bir takipçi onaylayana dek beklediği; eşzamansızın kayıp riskiyle eşzamanlının yavaşlığını dengeleyen orta yol.

Yazma eğriltme (write skew) İki işlemin ayrı ayrı geçerli ama birlikte bir değişmezi bozan kararlar aldığı, anlık görüntü yalıtımında görülebilen anormallik.

Yerellik İlişkili veriyi bir arada tutarak tek bir okumayla almayı sağlama; belge modelinin gömme yoluyla sunduğu kazanç.

Yetersayı (W + R > N) Lidersiz replikasyonda, yazma (W) ve okuma (R) kopya sayılarının toplamı kopya sayısını (N) aşacak biçimde seçilerek okumanın en güncel yazmayı görmesinin sağlanması.

Yetkilendirme (authorization) Kimliği bilinen bir kullanıcının hangi işlemleri yapabileceğine karar verme.

Yırtık yazma (torn write) Bir sayfa diske yazılırken sistemin çökmesiyle sayfanın yarısının eski, yarısının yeni kalması; sağlama toplamı ve append-only tasarımla ele alınır.

Yineleyici modeli (Volcano) Her plan düğümünün bir sonraki belgeyi "iste" çağrısıyla ürettiği, talepgüdümlü sorgu yürütme modeli; erken sonlanmayı doğal kılar.

Ek B — Kaynaklar

Bu kitap, belge veritabanlarını ve OxiDB'yi temelden, kod kullanmadan anlatmayı amaçladı. Daha derine inmek isteyenler için, burada konuları temalara göre gruplanmış bir okuma ve inceleme rehberi sunuyoruz. Önce kaynakları, ne içerdikleriyle ve hangi yöne baktıklarıyla temalara göre tanıtıyoruz; ardından, bu bölümün sonundaki **Kaynakça**'da, metin boyunca atıfta bulunduğumuz eserlerin tam künyelerini ve erişim bağlantılarını (URL/DOI) topluyoruz. Böylece hem bir pusula hem de doğrudan başvurabileceğiniz kesin bir referans listesi elinizde olur. Kitap boyunca, bir eserden ilk söz edilen sayfada onun künyesi **sayfa altındaki dipnotta** verilir; aşağıdaki Kaynakça ise bu eserleri tek bir yerde, erişim bağlantılarıyla birlikte toplar.

32.8 Genel: veri-yoğun sistemlerin temelleri

Bu kitabın ikinci kısmında ele aldığımız ilkelerin — depolama, dayanıklılık, indeksleme, replikasyon, tutarlılık — derinlemesine ve bütünsel bir incelemesi için, Martin Kleppmann'ın *Designing Data-Intensive Applications* adlı eseri (Kleppmann, 2017), alanın en yaygın başvurulan modern kaynağıdır. Veritabanı motorlarının iç mekanizmalarına — depolama düzenleri, B-ağaçları, LSM ağaçları, işlem ve dağıtım — daha doğrudan inen bir kaynak için Alex Petrov'un *Database Internals* kitabı önerilir (Petrov, 2019). Bu iki eser, bu kitabın ikinci kısmının doğal devamı niteliğindedir.

32.9 Veri modelleri ve ilişkisel temel

Üçüncü bölümde değindiğimiz ilişkisel modelin temelini atan, E. F. Codd'un 1970 tarihli "A Relational Model of Data for Large Shared Data Banks" makalesi, alanın kurucu metnidir ve mantık-fizik ayrışması fikrinin kaynağıdır. Belge modelinin pratiğini ve gömme-referans kararını daha somut görmek için, yaygın belge veritabanlarının (örneğin MongoDB'nin) veri modelleme kılavuzları, kavramları gerçek senaryolarla ilişkilendirir.

32.10 JSON ve belge biçimleri

Dördüncü bölümde tarihçesini anlattığımız JSON'un sade tanımı ve dilbilgisi için `json.org` adresi, biçimin kaynağıdır. Biçimin resmî, kesin tanımları, yazılım endüstrisi ve internet standart kuruluşlarının yayımladığı şartnamelerde bulunur. Veritabanlarının içeride kullandığı, JSON'un daha zengin ikili akrabaları (belge veritabanlarının ikili kodlamaları gibi) hakkında, ilgili sistemlerin biçim belgeleri ayrıntı sağlar.

32.11 Depolama motorları ve indeksleme

Beşinci ve yedinci bölümlerin konuları — B-ağaçları, log-yapılı ve LSM depolama, indeks yapıları — klasik veritabanı ders kitaplarında ve yukarıda anılan *Database Internals* eserinde kapsamlı biçimde işlenir. Tam

metin aramanın temelindeki alaka puanlama yöntemleri (terim sıklığı temelli yaklaşımlar ve onların olgun biçimleri), bilgi erişimi (information retrieval) alanının standart kaynaklarında ele alınır.

32.12 İşlemler, eşzamanlılık ve tutarlılık

Onuncu ve on birinci bölümlerin konuları — ACID, yalıtım düzeyleri, kilitleme, çok sürümlü ve iyimser eşzamanlılık denetimi — veritabanı kuramının klasik metinlerinde derinlemesine incelenir. Dağıtık tutarlılık tarafında, on birinci bölümde değindiğimiz CAP içgörüsü Eric Brewer'a; ilgili pratik tutarlılık modelleri ise dağıtık sistemler yazınına dayanır.

32.13 Ölçeklendirme ve konsensüs

On ikinci ve yirmi beşinci bölümlerde anlattığımız konsensüs, OxiDB'nin de dayandığı Raft protokolünün temelinde yatar; Diego Ongaro ve John Ousterhout'un "In Search of an Understandable Consensus Algorithm" başlıklı çalışması, Raft'ı anlaşılır biçimde tanıtan kurucu metindir ve konsensüsün neden çoğunlukla çalıştığını kavramak için en iyi başlangıçtır.

32.14 OxiDB'nin kendi kaynakları

Üçüncü kısımda anlattığımız OxiDB'nin somut tasarımına daha yakından bakmak isteyenler için en doğrudan kaynak, projenin açık kaynak deposudur (`parisxmas/OxiDB`). Deponun içinde, bu kitapta değindiğimiz birçok mühendislik kararının ardındaki gerekçeyi bulabileceğiniz birkaç materyal vardır. Mimari karar kayıtları (`docs/decisions/` altındaki belgeler), disk-öncelikli depolama ve sıkıştırma gibi konularda yapılan tercihleri ve ödünleşimleri belgeler. Karşılaştırmalı değerlendirme materyali (`tests/benchmark-1m/` altındaki rapor), yirmi yedinci bölümde özetlediğimiz ölçümlerin ayrıntısını içerir. Sürüm değişiklik günlüğü (`CHANGELOG.md`), bu kitap yazılırken eklenen yeteneklerin — çok-yönlü analiz, pencere fonksiyonları, sıkıştırmasız kip, per-koleksiyon ayarlar — gerekçeleriyle birlikte kaydını tutar. Ve doğrulama testleri (örneğin küme ve parçalar-arası toplama testleri), yirmi beşinci bölümde anlattığımız davranışların nasıl sınandığını gösterir.

32.15 Son bir söz

Bu kaynaklar, bir bitiş değil, bir başlangıçtır. Bu kitabın amacı, size belirli bir sistemi ezberletmek değil, herhangi bir belge veritabanına baktığınızda onun altındaki ilkeleri ve ödünleşimleri görebilecek bir bakış kazandırmaktı. O bakışla donanmış olarak, yukarıdaki kaynaklara — ya da karşınıza çıkacak herhangi bir veri sistemine — artık çok daha hazırlıklı yaklaşabilirsiniz. İyi okumalar.

32.16 Kaynakça

Aşağıdaki künyeler, kitap boyunca **sayfa altı dipnotlarda** anılan eserlerin kronolojik, tam listesidir. Bağlantılar mümkün olduğunda kalıcı DOI ya da resmî sayfa adresleridir; erişim tarihinde geçerliydiler. Bazı konferans yayınlarının, kitapların ve derleme ciltlerin (USENIX, BSDCan vb.) kalıcı bir DOI'si yoktur; bunlarda resmî sayfa adresi verilmiştir.

1. Denning, P. J. “The Working Set Model for Program Behavior.” *Communications of the ACM*, 11(5), 1968, ss. 323–333. https://doi.org/10.1145/363095.363141

2. Bloom, B. H. “Space/Time Trade-offs in Hash Coding with Allowable Errors.” *Communications of the ACM*, 13(7), 1970, ss. 422–426. https://doi.org/10.1145/362686.362692

3. Codd, E. F. “A Relational Model of Data for Large Shared Data Banks.” *Communications of the ACM*, 13(6), 1970, ss. 377–387. https://doi.org/10.1145/362384.362685

4. Bayer, R. ve McCreight, E. M. “Organization and Maintenance of Large Ordered Indexes.” *Acta Informatica*, 1(3), 1972, ss. 173–189. https://doi.org/10.1007/BF00288683

5. Codd, E. F. “Further Normalization of the Data Base Relational Model.” R. Rustin (ed.), *Data Base Systems* (Courant Computer Science Symposia 6), Prentice-Hall, 1972, ss. 33–64.

6. Bachman, C. W. “The Programmer as Navigator.” *Communications of the ACM*, 16(11), 1973, ss. 653–658. https://doi.org/10.1145/355611.362534

7. Lamport, L. “Time, Clocks, and the Ordering of Events in a Distributed System.” *Communications of the ACM*, 21(7), 1978, ss. 558–565. https://doi.org/10.1145/359545.359563

8. Gifford, D. K. “Weighted Voting for Replicated Data.” *Proc. ACM SOSP*, 1979, ss. 150–162. https://doi.org/10.1145/800215.806583

9. Selinger, P. G.; Astrahan, M. M.; Chamberlin, D. D.; Lorie, R. A.; Price, T. G. “Access Path Selection in a Relational Database Management System.” *Proc. ACM SIGMOD*, 1979, ss. 23–34. https://doi.org/10.1145/582095.582099

10. Gray, J. “The Transaction Concept: Virtues and Limitations.” *Proc. 7th Int. Conf. on Very Large Data Bases (VLDB ’81)*, 1981, ss. 144–154. https://dl.acm.org/doi/10.5555/1286831.1286846

11. Kung, H. T. ve Robinson, J. T. “On Optimistic Methods for Concurrency Control.” *ACM Transactions on Database Systems*, 6(2), 1981, ss. 213–226. https://doi.org/10.1145/319566.319567

12. Bernstein, P. A. ve Goodman, N. “Multiversion Concurrency Control—Theory and Algorithms.” *ACM Transactions on Database Systems*, 8(4), 1983, ss. 465–483. https://doi.org/10.1145/319996.319998

13. Fischer, M. J.; Lynch, N. A.; Paterson, M. S. “Impossibility of Distributed Consensus with One Faulty Process.” *Journal of the ACM*, 32(2), 1985, ss. 374–382. https://doi.org/10.1145/3149.214121

14. Herlihy, M. P. ve Wing, J. M. “Linearizability: A Correctness Condition for Concurrent Objects.” *ACM Transactions on Programming Languages and Systems*, 12(3), 1990, ss. 463–492. https://doi.org/10.1145/78969.78972

15. Pugh, W. “Skip Lists: A Probabilistic Alternative to Balanced Trees.” *Communications of the ACM*, 33(6), 1990, ss. 668–676. https://doi.org/10.1145/78973.78977

16. Gray, J. ve Reuter, A. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1992. ISBN 978-1-55860-190-1.

17. Mohan, C.; Haderle, D.; Lindsay, B.; Pirahesh, H.; Schwarz, P. “ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging.” *ACM Transactions on Database Systems*, 17(1), 1992, ss. 94–162. https://doi.org/10.1145/128765.128770

18. Graefe, G. “Volcano—An Extensible and Parallel Query Evaluation System.” *IEEE Transactions on Knowledge and Data Engineering*, 6(1), 1994, ss. 120–135. https://doi.org/10.1109/69.273032

19. Berenson, H.; Bernstein, P.; Gray, J.; Melton, J.; O’Neil, E.; O’Neil, P. “A Critique of ANSI SQL Isolation Levels.” *Proc. ACM SIGMOD*, 1995, ss. 1–10. https://doi.org/10.1145/223784.223785

20. O'Neil, P.; Cheng, E.; Gawlick, D.; O'Neil, E. "The Log-Structured Merge-Tree (LSM-Tree)." *Acta Informatica*, 33(4), 1996, ss. 351–385. https://doi.org/10.1007/s002360050048

21. Karger, D.; Lehman, E.; Leighton, T.; Panigrahy, R.; Levine, M.; Lewin, D. "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web." *Proc. ACM STOC*, 1997, ss. 654–663. https://doi.org/10.1145/258533.258660

22. Lamport, L. "The Part-Time Parliament." *ACM Transactions on Computer Systems*, 16(2), 1998, ss. 133–169. https://doi.org/10.1145/279227.279229

23. Adya, A.; Liskov, B.; O'Neil, P. "Generalized Isolation Level Definitions." *Proc. IEEE ICDE*, 2000, ss. 67–78. https://doi.org/10.1109/ICDE.2000.839388

24. National Institute of Standards and Technology. *FIPS PUB 197: Advanced Encryption Standard (AES)*. NIST, 2001. https://doi.org/10.6028/NIST.FIPS.197

25. Gilbert, S. ve Lynch, N. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services." *ACM SIGACT News*, 33(2), 2002, ss. 51–59. https://doi.org/10.1145/564585.564601

26. Megiddo, N. ve Modha, D. S. "ARC: A Self-Tuning, Low Overhead Replacement Cache." *Proc. USENIX FAST*, 2003. https://www.usenix.org/conference/fast-03/arc-self-tuning-low-overhead-replacement-cache

27. DeCandia, G.; Hastorun, D.; Jampani, M.; Kakulapati, G.; Lakshman, A.; Pilchin, A.; Sivasubramanian, S.; Vosshall, P.; Vogels, W. "Dynamo: Amazon's Highly Available Key-value Store." *Proc. ACM SOSP*, 2007, ss. 205–220. https://doi.org/10.1145/1294261.1294281

28. Cahill, M. J.; Röhm, U.; Fekete, A. D. "Serializable Isolation for Snapshot Databases." *Proc. ACM SIGMOD*, 2008, ss. 729–738. https://doi.org/10.1145/1376616.1376690

29. Manning, C. D.; Raghavan, P.; Schütze, H. *Introduction to Information Retrieval*. Cambridge University Press, 2008. https://nlp.stanford.edu/IR-book/information-retrieval-book.html

30. Percival, C. "Stronger Key Derivation via Sequential Memory-Hard Functions." *BSDCan*, 2009. https://www.tarsnap.com/scrypt/scrypt.pdf

31. Robertson, S. ve Zaragoza, H. "The Probabilistic Relevance Framework: BM25 and Beyond." *Foundations and Trends in Information Retrieval*, 3(4), 2009, ss. 333–389. https://doi.org/10.1561/1500000019

32. Newman, C.; Menon-Sen, A.; Melnikov, A.; Williams, N. *RFC 5802: Salted Challenge Response Authentication Mechanism (SCRAM) SASL and GSS-API Mechanisms*. IETF, 2010. https://doi.org/10.17487/RFC5802

33. Abadi, D. J. "Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story." *Computer (IEEE)*, 45(2), 2012, ss. 37–42. https://doi.org/10.1109/MC.2012.33

34. Brewer, E. "CAP Twelve Years Later: How the 'Rules' Have Changed." *Computer (IEEE)*, 45(2), 2012, ss. 23–29. https://doi.org/10.1109/MC.2012.37

35. Ongaro, D. ve Ousterhout, J. "In Search of an Understandable Consensus Algorithm." *Proc. 2014 USENIX Annual Technical Conference (USENIX ATC '14)*, ss. 305–319. https://raft.github.io/raft.pdf

36. T. Hansen. *RFC 7677: SCRAM-SHA-256 and SCRAM-SHA-256-PLUS Simple Authentication and Security Layer (SASL) Mechanisms*. IETF, 2015. https://doi.org/10.17487/RFC7677

37. Biryukov, A.; Dinu, D.; Khovratovich, D. "Argon2: New Generation of Memory-Hard Functions for Password Hashing and Other Applications." *Proc. IEEE EuroS&P*, 2016, ss. 292–302. https://doi.org/10.1109/EuroSP.2016.31

38. Lu, L.; Pillai, T. S.; Arpaci-Dusseau, A. C.; Arpaci-Dusseau, R. H. “WiscKey: Separating Keys from Values in SSD-Conscious Storage.” *Proc. USENIX FAST*, 2016. https://www.usenix.org/conference/fast16/technical-sessions/presentation/lu

39. Bray, T. (ed.). *RFC 8259: The JavaScript Object Notation (JSON) Data Interchange Format*. IETF, STD 90, 2017. https://www.rfc-editor.org/rfc/rfc8259.html

40. Ecma International. *ECMA-404: The JSON Data Interchange Syntax*. 2. baskı, 2017. https://ecma-international.org/publications-and-standards/standards/ecma-404/

41. Kleppmann, M. *Designing Data-Intensive Applications*. O’Reilly Media,

    2017. https://dataintensive.net/

42. Petrov, A. *Database Internals: A Deep Dive into How Distributed Data Systems Work*. O’Reilly Media, 2019. https://www.databass.dev/

43. Bormann, C. ve Hoffman, P. *RFC 8949: Concise Binary Object Representation (CBOR)*. IETF, 2020. https://doi.org/10.17487/RFC8949

44. Collet, Y. ve Kucherawy, M. (ed.). *RFC 8878: Zstandard Compression and the ‘application/zstd’ Media Type*. IETF, 2021. https://doi.org/10.17487/RFC8878

Aşağıdaki kaynaklar, sürümsüz/canlı çevrimiçi belgelerdir:

45. Crockford, D. “Introducing JSON.” json.org. https://www.json.org/

46. *BSON (Binary JSON) Specification*. MongoDB Inc. https://bsonspec.org/spec.html

47. *JSON Schema — Specification*. JSON Schema Org. https://json-schema.org/specification