

# Veri yoğunluklu sistemlerde 5 kırılma noktası

En ucuz müdahaleden en pahalısına bir sıra ve her bileşeni gerekçeli bir Laravel production stack'i.

Muhammet Şafak • 3 Ekim 2026



# Beş kırılma noktası

Working set'ten ayrı veri katmanına: en ucuz müdahaleden en pahalısına.

ÖLÇEĞİ BELİRLEYEN

# Request değil, veri profili.

Bir sistem dakikada 50 request alıp yine de veri tarafından boğulabilir; bir başkası saniyede 5.000 request'i rahat karşılarken veri katmanı el değmemiş durur.

# Beş eksen



## Working set / RAM oranı

Sık okunan veri belleğe sığıyor mu?



## Okuma / yazma oranı

Yük hangi yöne ağır basıyor?



## Tek tablo boyutu

Tek bir tablo ne kadar büyüdü?



## Yazma throughput'u

Primary saniyede ne kadar yazıyor?



## Tutarlılık gereksinimi

Hangi okuma en güncel veriyi görmek zorunda?

BAŞLANGIÇ MİMARİSİ

# Tek primary ile başlamak

PHP-FPM, PostgreSQL'e pgBouncer üzerinden, Redis'e doğrudan bağlanır. Birkaç on GB veri ve dakikada binlerce request için yeterli; working set RAM'e sığıldığı sürece.

**PHP-FPM**

uygulama

**pgBouncer**

bağlantı havuzu

**PostgreSQL 16**

tek primary

**Redis 7**

cache · session · lock

# Sinyal: cache hit ratio

Sık okunan veri RAM'e sığmıyor. %99 üzeri sağlıklı. %95'e doğru düşüş, disk IOPS'u ile birlikte okunduğunda working set'in RAM'den taştiğini gösterir.

pg\_stat\_database.sql

```
1 SELECT
2   sum(blks_hit) * 100
3   / nullif(sum(blks_hit) + sum(blks_read), 0)
4   AS cache_hit_ratio
5 FROM pg_stat_database;
```

# Önce ucuz olanı tüketin

1

## Index disiplini

gereksiz taramayı kes

2

## Ölü veriyi ayır

sıcak veriyi küçült

3

## Vertical scale

en son, daha çok RAM

# Okuma replica'ya, yazma primary'ye

Sinyal: CPU yüksek, SELECT baskın,  
pgBouncer'da `cl_waiting` birikiyor.  
`sticky`, aynı request içinde  
yazmadan sonraki okumaları  
primary'ye yollar.

config/database.php

```
1 'pgsql' => [  
2   'driver' => 'pgsql',  
3   'read' => ['host' => ['10.0.0.2']], // replica  
4   'write' => ['host' => ['10.0.0.1']], // primary  
5   'sticky' => true,  
6   // ...ortak ayarlar  
7 ],
```

# Replica'nın getirdiği yeni sorun



## GEÇİKME

Replication lag: replica geride kalır; read-after-write garantisi kaybolur.



## İZLEYİN

pg\_stat\_replication'da replay\_lsn farkını lag\_bytes olarak okuyun.



## PRIMARY

Bakiye, stok, yetki gibi tutarlılık isteyen okumalar primary'de kalır.



## SINIR

Read replica yazma kapasitesine hiçbir şey katmaz.

# Zamana göre bölünen tablo

Sinyal: `n_dead_tup` birikiyor,  
`last_autovacuum` geride kalıyor.  
Partition'ları elle açmak yerine  
`pg_partman` kullanılabilir.

events.sql

```
1 CREATE TABLE events (  
2   id          bigserial,  
3   occurred_at timestamptz NOT NULL,  
4   payload     jsonb       NOT NULL  
5 ) PARTITION BY RANGE (occurred_at);  
6  
7 CREATE TABLE events_2026_05 PARTITION OF events  
8   FOR VALUES FROM ('2026-05-01') TO ('2026-06-01');
```

# Partitioning ne kazandırır?



## Partition pruning

Sorgu yalnız ilgili partition'a bakar.



## Ucuz silme

Eski veri DROP TABLE events\_2026\_01 ile gider.



## Sınırlı vacuum

Vacuum tüm tabloyu değil, partition'ı tarar.



## Primary key

Partition anahtarını içermeli: (id, occurred\_at).



## BRIN

Zamana göre sıralı veride küçük ve ucuz index.

# Sinyal: I/O wait, checkpoint ve WAL baskısı



GERÇEKÇİ BEKLENTİ

# Çoğu sistem 4. noktaya varmaz.

Acele etmeyin: çoğu sistem 4. noktaya hiç ulaşmaz, ulaşanların çoğu da aslında 1. noktayı (eksik index, şişmiş working set) atlamiş olduğu için erken gelir.

# Ayrı veri katmanı



## HAMLE

Veri-baskın alan ayrı bir PostgreSQL instance'ına taşınır.



## GEÇİŞ

pgBackRest restore, logical replication ve kısa bir cutover.



## BEDEL

İkinci backup, monitoring ve upgrade; iki veri arasında JOIN yok.



## KURAL

Ayırma kararı veri profiliyle verilir, organizasyon şemasıyla değil.

# Yanlış kırılma noktaları

| GÖRÜNEN            | GERÇEK                               | DOĞRU HAMLE                      |
|--------------------|--------------------------------------|----------------------------------|
| "Veritabanı yavaş" | N+1: uygulama 200 sorgu atıyor       | Eager loading; sorguları say     |
| "Replica lazım"    | Tek sorguda eksik index              | EXPLAIN ANALYZE, index ekle      |
| "Sharding lazım"   | Replica, partitioning denenmedi      | Önce 2. ve 3. noktayı tüket      |
| "NoSQL'e geçelim"  | jsonb, BRIN, partitioning atıl       | PostgreSQL'i sonuna kadar kullan |
| "Yazma için cache" | Cache okumayı azaltır, yazmayı değil | 4. noktaya bak                   |

ORTAK HATA KALIBI

# Hiçbiri yazma kapasitesi eklemmez.

Read replica okumayı ölçerler. Cache okumayı azaltır.  
Partitioning tek tabloyu evcilleştirir. Hiçbiri yazma  
throughput'unu artırmaz.

JSONB · LISTEN/NOTIFY · PARTITIONING · BRIN · LOGICAL REPLICATION

# %20'si kullanıldı. Bu bir sınır değil.

Çoğu ekip PostgreSQL'in %20'sini kullanıp "PostgreSQL yetmiyor" sonucuna varır.

# En ucuzdan en pahalıya



# Laravel stack

"Her şey gerekli" demiyorum — gerekli olduğunu anlayana kadar minimum tutmayı savunan biriyim — ama her bileşenin kazandırdığı somut.

## ANATOMİ

# Her bileşen neden orada?

İstek Cloudflare'den PHP-FPM'e iner. PHP-FPM üç arka servise ayrı bağlanır: pgBouncer üzerinden PostgreSQL, Redis ve RabbitMQ. Worker'lar Supervisor altında Horizon ile koşar.

### Cloudflare

DDoS, WAF, rate limiting, TLS, CDN

### Nginx

TLS, static asset, fastcgi tek config

### PHP-FPM

opcache + preload

### pgBouncer

transaction mode bağlantı havuzu

### PostgreSQL

pgBackRest: full + incremental + WAL PITR

### Redis · RabbitMQ

cache/session/lock · queue

REDİS

# Tek instance, üç şapka

Cache (`Cache::remember`), session (`SESSION_DRIVER=redis`) ve lock (`Cache::lock`) tek Redis'te güvenle çalışır; yeter ki bu üç ayar yerinde olsun.

redis.conf

```
1 maxmemory-policy volatile-lru
2 appendonly no
3 save 900 1 300 10 60 10000
```

# RabbitMQ ile Redis kuyruğu

|                | Redis                                               | RabbitMQ                          |
|----------------|-----------------------------------------------------|-----------------------------------|
| PERSISTENCE    | Bellekten servis eder; diske RDB ya da AOF ile iner | ✓ Kalıcı mesajları diske yazar    |
| YENİDEN TESLİM | retry_after zaman aşımına bakar                     | ✓ Consumer'ın bağlantısına bakar  |
| ROUTING        | Yönlendirme kararı uygulamada kalır                 | ✓ Exchange'ler karar verir        |
| MONITORING     | Horizon; uygulamanın dispatch ettiğiyle sınırlı     | ✓ Broker düzeyinde yönetim paneli |

# Hangi parçası kaldırılırsa ne kaybedersiniz?

| BİLEŞEN           | ÇIKARINCA OLAN                                                               |
|-------------------|------------------------------------------------------------------------------|
| <b>pgBouncer</b>  | Connection storm'larında PostgreSQL bağlantısı tükenir, request'ler 500 atar |
| <b>Supervisor</b> | Worker'lar crash sonrası geri gelmez, alarm kurmak zorundasınız              |
| <b>Horizon</b>    | Görünürlük çöker, "queue neden yavaş" sorusunu kör cevaplarsınız             |
| <b>opcache</b>    | Her request PHP dosyasını disk'ten okur ve parse eder, ~5x yavaşlama         |
| <b>pgBackRest</b> | pg_dump kalır: PITR gider, incident'ta hata payınız bir günlük yedek         |
| <b>Redis lock</b> | Race condition'lara açıksınız, distributed-safe mutex'iniz kalmaz            |

BORİNG STACK

# Nicel değerli, değişimi kolay.

Boring stack'in gücü tam burada: her parça nicel olarak değerli ve değişimi kolay.

# Teşekkürler

sade.dev

Önce veri profilini ölçün, sonra en ucuz müdahaleyi seçin.

