VERİTABANI SİSTEMLERİ

Veritabanı Sistemleri

Ders notları ve alıştırmalar

0 Veritabanı nedir: dosyadan veritabanına

1 İlişkisel model ve ilişkisel cebir

2 SQL temel: tablo kurmak ve sorgulamak

3 SQL orta: birleştirme ve gruplama

4 SQL ileri: alt sorgu ve pencere fonksiyonu

5 Veritabanı programlama ve saklı yordamlar

6 Veri bütünlüğü ve güvenlik

7 ER ve EER ile kavramsal tasarım

8 Normalizasyon

9 Yarı yapısal veri: XML ve JSON

10 Fiziksel tasarım ve indeksler

11 Sorgu işleme ve iyileştirme

12 İşlemler, eşzamanlılık ve kurtarma

13 NoSQL, dağıtık ve vektör veritabanları

HEKZON

hekzon.com/dersler

14 bölüm · 40 konu

İçindekiler

Bu notlar, YouTube'daki Veritabanı Sistemleri dersinin yazılı hâlidir. Her konunun yanındaki zaman, uzun videoda o konunun başladığı yeri gösterir. Her bölümün sonunda sorular var; cevaplarını ve çözümlerini en sonda bulabilirsin. Aynı soruları hekzon.com/dersler adresinde, cevabını hemen göreceğin alıştırmalar olarak da çözebilirsin.

0 **Veritabanı nedir: dosyadan veritabanına** 4
Üç Excel dosyası · Disk değişince

1 **İlişkisel model ve ilişkisel cebir** 12
İki Ahmet · Bir soru · Bütün zorunlu dersler

2 **SQL temel: tablo kurmak ve sorgulamak** 22
Tek cümlelik istek · Bir komut, dört kupon · Kayıp müşteri

3 **SQL orta: birleştirme ve gruplama** 32
Numara var, ad yok · Yirmi kalem · Tek tablo, iki bakış

4 **SQL ileri: alt sorgu ve pencere fonksiyonu** 43
Boş liste · Sıfır ürün · GROUP BY yetmez

5 **Veritabanı programlama ve saklı yordamlar** 54
Düğmenin arkası · Üç program, bir kural

6 **Veri bütünlüğü ve güvenlik** 62
Eksi 349 lira · Ece hesabını kapatıyor · Bir tırnak işareti

7 **ER ve EER ile kavramsal tasarım** 74
Kâğıt · Üç soru · Aynı tablo mu?

8 **Normalizasyon** 85
Bir telefon, kaç satır? · Duvardaki program · İtiraz etmeyen veritabanı · Tekrar nereden geliyor?

9 **Yarı yapısal veri: XML ve JSON** 96
Kamera ile beden · Pil süresi nerede?

10 **Fiziksel tasarım ve indeksler** 104
Bir milyon sipariş · Bir aralık · Tek okuma

11 **Sorgu işleme ve iyileştirme** 114
Ne, nasıl değil · 851 bin mi, 3 bin mi? · Mekanik klavyeyi kim aldı?

12 **İşlemler, eşzamanlılık ve kurtarma** 125
Son koltuk · Neden sırayla değil · COMMIT ve elektrik

13 **NoSQL, dağıtık ve vektör veritabanları** 135
Yüz kat büyüyen uygulama · İki ekran, iki sayı · Ortak kelime yok

VERİTABANI SİSTEMLERİ

0 Veritabanı nedir: dosyadan veritabanına

Bu bölümde kayıtları ayrı dosyalarda tutmanın neden er geç sorun çıkardığını ve veritabanı yönetim sisteminin bu sorunları nasıl üstlendiğini görüyoruz. Sonra veritabanının içine bakıyoruz: şema ile o anki satırlar, iki dil, üç düzey ve diskteki düzen değişince programların neden bozulmadığı.

Üç Excel dosyası

videoda 0:11

Bir lise bütün kayıtlarını Excel dosyalarında tutuyor. Kayıt listesi müdür yardımcısında, servis listesi servis sorumlusunda, yemek listesi muhasebede. Her dosya kendi işini görüyor. Peki ilk ne ters gider?

Irmak Aksoy'un adresi hem kayıt listesinde hem servis listesinde yazılı. Aile taşınır, müdür yardımcısı kayıt dosyasını düzeltir; servis dosyası kimsenin aklına gelmez:

dosya	adres
kayıt	Bahar Sok. 4
servis	Gazi Mah. 12

Artık Irmak'ın iki adresi var ve pazartesi sabahı servis eski adrese gider. Aynı bilgi birkaç yerde tutulursa er geç birbirini tutmaz hâle gelir: **tekrar** (redundancy), **tutarsızlığı** (inconsistency) doğurur.



Şekil 0.1 Kayıt listesinde Irmak'ın adresi Bahar Sok. 4, servis listesinde Gazi Mah. 12: aynı öğrenci, iki adres.

Dosyaların başka dertleri de var. Müdür sorar: 3 numaralı servis hattındaki öğrencilerden kimin yemek ücreti ödenmedi? Cevap iki dosyada; biri servis listesini, öteki yemek listesini açar ve adları tek tek eşleştirir. Muhasebe Kerem'in soyadını büyük harflerle ("Kerem POLAT") yazmış; ad

tutmaz, Kerem gözden kaçar. Elle bulunan cevap yalnız Arda Yurt'tur, doğrusu Kerem Polat ve Arda Yurt. Bu işi bir program yapsa bile her yeni soru için yeni bir program yazmak gerekir.

Dönem başında iki görevli aynı dosyanın birer kopyasında çalışır: biri 3 yeni öğrenci ekler, öteki 2 adresi düzeltir. İkisi de dosyayı ortak klasöre kaydeder; sonra kaydeden, öncekinin bütün işini siler. Dosya, iki kişinin aynı anda yazdığını bilmez.

Yeni gelen Efe Erdem'in kaydı üç dosyaya da yazılacak. Kayıt ve servis listesi tamam, tam yemek listesine geçerken elektrik kesilir. Kayıt listesi 6, servis listesi 5, yemek listesi 5 satır kalır: Efe iki dosyada var, birinde yok. İş yarım kaldı ama hiçbir yerde yarım olduğu yazmıyor.

Son dert, kimin neyi göreceği. Servis şoförüne yalnız ad ve adres gerekir; kayıt dosyasında ise Irmak'ın fıstık alerjisi gibi sağlık notları da duruyor. Ya bütün dosya şoföre verilir ya da yalnız ad ve adresle yeni bir kopya açılır. Servis listesi zaten tam da böyle doğmuştu; her yeni kopya, yeni bir tutarsızlık demek.

Bu dertlerin ortak çözümü, veriyi tek bir yerde düzenli tutmak ve her programın ona aynı yazılımın elinden ulaşmasıdır. Düzenli tutulan bu veri topluluğuna **veritabanı** (database), onu yöneten yazılıma **veritabanı yönetim sistemi** (VTYS; database management system) denir. SQLite, PostgreSQL, MySQL ve Oracle bunlardan bazıları. SQLite öyle yaygın ki Android ve iPhone'daki rehber de mesajlar da onun küçük veritabanı dosyalarında durur. VTYS, dosyalarla tek tek uğraşan programların işini üstlenir:

iş	ne yapar	çözdüğü dert
tanımlama	verinin kalıbı bir kez ve tek yerde yazılır	aynı adres iki yerde
sorgulama	yeni bir soru program yazmadan tek bir sorguyla sorulur	her soru yeni bir iş
bütünlük	kurallar korunur, aynı öğrenci numarası ikinci kez yazılamaz	tutarsız kayıt
eşzamanlılık	iki kişi aynı anda yazsa da biri ötekinin işini silmez	son kaydeden öncekini siler
kurtarma	elektrik kesilirse yarım kalan iş geri alınır	yarım kalan iş
güvenlik	herkes yalnız görmesi gerekeni görür	kim neyi görür

Bütünlük kuralı işlerken, aynı numarayla ikinci bir öğrenci eklenmek istenince SQLite eklemeyi reddeder:

```
HATA: IntegrityError: UNIQUE constraint failed: ogrenci.no
```



Şekil 0.2 Programlar veritabanına VTYS üzerinden ulaşır; sağda VTYS'nin altı işi ve bütünlüğün hata mesajı.

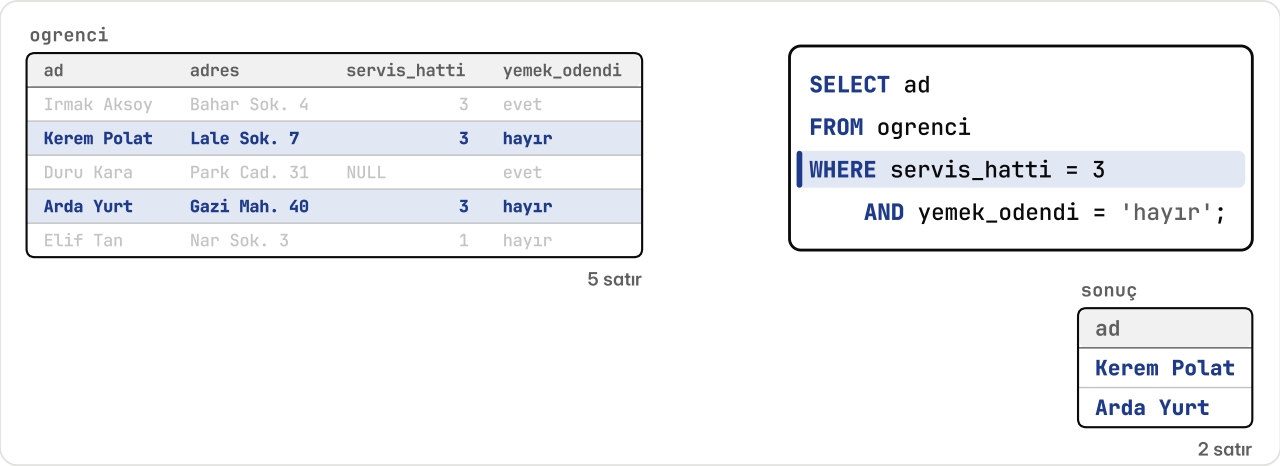
Okulun verisi tek bir veritabanında, `ogrenci` tablosunda dursa, Irmak'ın adresi tek satırda, tek kez yazılır. Aile taşınınca bir kez düzeltilir; servis de muhasebe de aynı satırı okur:

ad	adres	servis_hatti	yemek_odendi
Irmak Aksoy	Bahar Sok. 4	3	evet
Kerem Polat	Lale Sok. 7	3	hayır
Duru Kara	Park Cad. 31	NULL	evet
Arda Yurt	Gazi Mah. 40	3	hayır
Elif Tan	Nar Sok. 3	1	hayır

Duru servise binmediği için hattı boştur (NULL). Müdürün sorusu 4 satırlık bir sorguya döner ve Kerem'le Arda'yı birlikte bulur:

```
SELECT ad
FROM ogrenci
WHERE servis_hatti = 3
    AND yemek_odendi = 'hayır';
```

ad
Kerem Polat
Arda Yurt



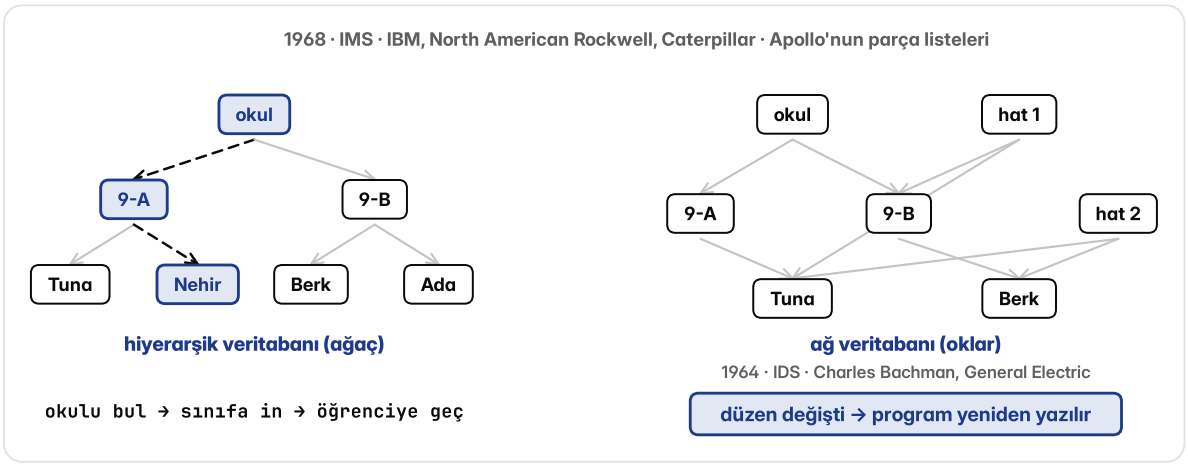
Şekil 0.3 Tek ogrenci tablosu, müdürün dört satırlık sorgusu ve sonucu: Kerem Polat, Arda Yurt.

Disk değişince

videoda 2:50

Bir okulun servis programı, yemekhane programı ve sitesi aynı veritabanını kullanıyor. Veritabanı daha hızlı çalışsın diye kayıtlar diskte başka bir düzene geçiriliyor. Bu üç program yeniden mi yazılmalı?

1960'larda cevap evetti. 1968'de, Apollo programının parça listelerini tutmak için IBM, North American Rockwell ve Caterpillar'ın birlikte geliştirdiği IMS çalışmaya başladı. Bu veritabanı kayıtları bir ağaç gibi diziyordu; buna **hiyerarşik veritabanı** denir. Charles Bachman'ın General Electric'te geliştirdiği IDS (1964) gibi sistemler ise kayıtları birbirine oklarla bağlıyordu; bunlar **ağ veritabanı**dır. Program veriye bu yolu adım adım yürüyerek ulaşırdı: önce okulu bul, oradan sınıfa in, sınıftan öğrenciye geç. Yol programın içine yazılıydı; diskteki düzen değişince programlar da yeniden yazılırdı.



Şekil 0.4 Solda okul, sınıf ve öğrenciden oluşan hiyerarşik ağaç ve programın izlediği yol; sağda kayıtları oklarla bağlayan ağ veritabanı.

Bugünün veritabanları veriyi bir **veri modeli**yle anlatır, yani verinin hangi kalıplarla tarif edileceğiyle. En yaygını, veriyi tablolarla anlatan ilişkisel model. Bir tablonun iki yüzü var. **Şema** (schema) tablonun kalıbıdır: `servis` tablosunda öğrenci numarası, ad, adres ve hat sütunları olacak. **Örnek** (instance) ise o anki satırlardır: sabah 4 öğrenci var, öğlen Yiğit Er'in kaydı gelir ve 5 olur. Örnek gün içinde defalarca değişir, şema ise belki yılda bir kez.

Bu iki yüz için iki ayrı dil var. **Veri tanımlama dili** (DDL, data definition language) kalıbı kurar ve değiştirir; tabloyu açan komut bu dildendir:

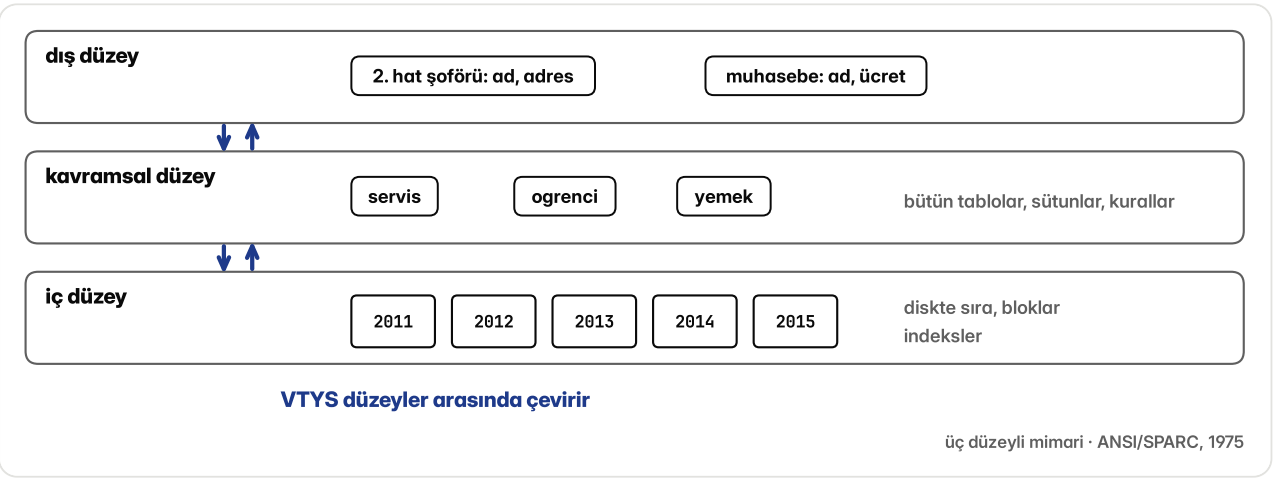
```
CREATE TABLE servis (
    no INTEGER PRIMARY KEY,
    ad TEXT NOT NULL,
    adres TEXT,
    hat INTEGER
);
```

Veri işleme dili (DML, data manipulation language) satırlarla çalışır: INSERT yeni satır ekler, SELECT satırları sorar.

```
INSERT INTO servis VALUES
    (2015, 'Yiğit Er', 'Gül Cad. 2', 1);
SELECT * FROM servis;
```

Bugün ikisi de SQL'in içindedir.

Veritabanı yönetim sistemi veriyi üç düzeyde ele alır. En altta **iç düzey**: kayıtların diskte hangi sırayla, hangi bloklarda durduğu ve aramayı hızlandıran indeksler. Ortada **kavramsal düzey**: okulun bütün tabloları, sütunları ve kuralları, diskten bağımsız olarak tek bir şemada. En üstte **dış düzey**: her kullanıcının kendi görünümü. 2. hattın şoförü yalnız kendi hattındaki öğrencilerin adını ve adresini görür, muhasebe yalnız ücretleri. Sistem bu düzeyler arasında çeviri yapar. Bu üç düzeyli mimari 1975'te ANSI/SPARC raporuyla tanımlandı.

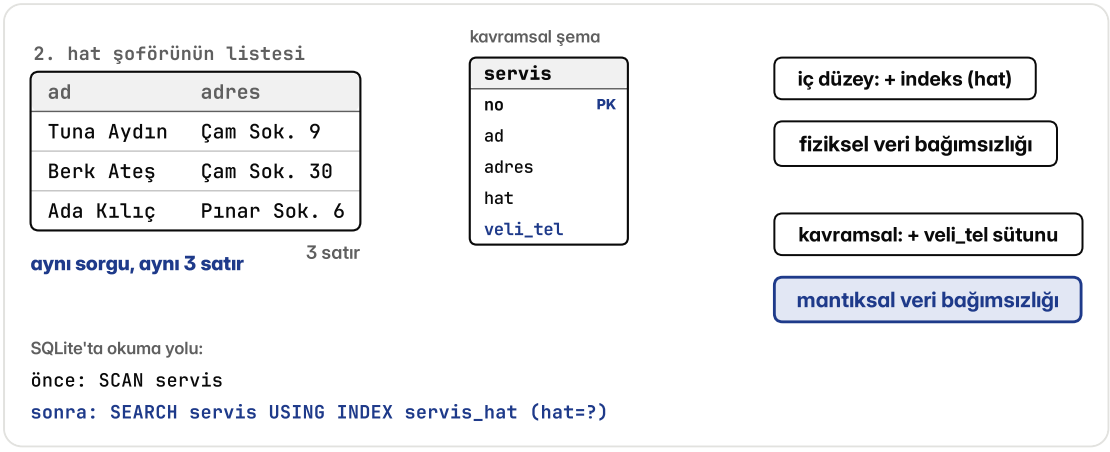


Şekil 0.5 Üç düzey üst üste: dış düzeyde şoförün ve muhasebenin görünümü, kavramsal düzeyde tablolar, iç düzeyde disk blokları.

Bu katmanların kazancı **veri bağımsızlığı**dır. 2. hattın şoförünün listesi şudur:

ad	adres
Tuna Aydın	Çam Sok. 9
Berk Ateş	Çam Sok. 30
Ada Kılıç	Pınar Sok. 6

`servis` tablosuna hat sütunu için bir indeks eklensin; iç düzey değişir. Şoförün listesi aynı sorguyla yine aynı 3 öğrenciyi getirir. Yalnız sistem artık bütün tabloyu baştan sona okumaz, indeksten bulur. SQLite'ın okuma yolu önce `SCAN servis` iken sonra `SEARCH servis USING INDEX servis_hat (hat=?)` olur. Buna **fiziksel veri bağımsızlığı** denir. Sonra kavramsal şemaya veli telefonu (`veli_tel`) diye yeni bir sütun eklensin. Şoförün görünümü değişmez, yine ad ve adres: bu da **mantıksal veri bağımsızlığı**.



Şekil 0.6 Şoförün 3 satırlık listesi; indeks ve veli_tel sütunu eklendikten sonra da aynı, altında SQLite'ın iki okuma yolu.

Kavramsal şemayı kuran, kimin neyi göreceğine karar veren, yedek alan ve yavaşlayan sorguları izleyen kişiye **veritabanı yöneticisi** (database administrator, DBA) denir. Programcı veriyi kullanır, veritabanı yöneticisi verinin düzenini ve güvenliğini korur.

Okulun kayıtları diskte yeniden dizilse de servis programı, yemekhane programı ve site tek satırı değişmeden çalışır. Programlar yalnız neyi istediklerini söyler; verinin diskte nasıl durduğunu ve nasıl bulunacağını veritabanı yönetim sistemi bilir.

Özet

- Aynı bilgiyi birkaç dosyada tutmak tekrar yaratır, tekrar da er geç tutarsızlığa döner.
- Dosyalarla her yeni soru yeni bir iştir; aynı anda yazan iki kişi birbirinin işini siler, kesilen elektrik işi yarım bırakır, yetki ancak yeni kopyayla verilir.
- Veritabanı yönetim sistemi tanımlama, sorgulama, bütünlük, eşzamanlılık, kurtarma ve güvenlik işlerini programların yerine üstlenir.
- Şema tablonun kalıbıdır, örnek o anki satırlardır; kalıbı veri tanımlama dili, satırları veri işleme dili işler.
- Veritabanı iç, kavramsal ve dış düzeyde ele alınır; indeks eklemek programları değiştirmez (fiziksel veri bağımsızlığı), yeni sütun da görünümleri değiştirmez (mantıksal veri bağımsızlığı).

Sorular

1 **Irmak'ın adresi kayıt listesinde "Bahar Sok. 4", servis listesinde "Gazi Mah. 12" yazıyor. Dosyalarla çalışmanın hangi derdi bu?**

A İki kişinin aynı anda yazması

B Aynı bilginin tekrarından doğan tutarsızlık

C Yarım kalan iş

D Kimin neyi göreceği

2 **Yeni öğrencinin kaydı kayıt ve servis listesine yazıldı, yemek listesine geçerken elektrik kesildi. Veritabanı yönetim sisteminin hangi işi bu derdi çözer?**

A Güvenlik

B Sorgulama

C Tanımlama

D Kurtarma

3 **Servis listesi ile yemek listesi adla eşleştirildiğinde 3. hatta ödenmemiş yalnız Arda bulunuyordu, çünkü muhasebe "Kerem POLAT" yazmıştı. Muhasebe adı "Kerem Polat" diye düzeltse, aynı eşleştirme 3. hatta ödenmemiş kaç öğrenci bulur?**

Cevap:

4 **Öğlen servis tablosuna Yiğit Er'in kaydı eklendi; satır sayısı 4'ten 5'e çıktı. Ne değişti?**

A Örnek değişti, şema aynı kaldı

B Şema değişti, örnek aynı kaldı

C İkisi de değişti

D Hiçbiri değişmedi

5 **Aşağıdaki komutlardan hangisi veri tanımlama diline (DDL) aittir?**

A SELECT * FROM servis;

B INSERT INTO servis VALUES (2015, 'Yiğit Er', 'Gül Cad. 2', 1);

C CREATE TABLE servis (no INTEGER PRIMARY KEY, ad TEXT NOT NULL, adres TEXT, hat INTEGER);

D UPDATE servis SET hat = 2 WHERE no = 2015;

6 **Servis tablosunun hat sütununa bir indeks eklendi. Şoförün listesi aynı sorguyla yine aynı 3 öğrenciyi getirdi, yalnız SQLite artık tabloyu baştan sona okumadı. Buna ne denir?**

A Mantıksal veri bağımsızlığı

B Fiziksel veri bağımsızlığı

C Tutarsızlık

D Eşzamanlılık

7 **Notta gösterilen 5 satırlık ogrenci tablosunda SELECT count(*) FROM ogrenci WHERE servis_hatti = 3 AND yemek_odendi = 'evet'; sorgusu hangi sayıyı döndürür?**

Cevap: _____

8 **Hiyerarşik ve ağ veritabanlarında diskteki düzen değişince programların neden yeniden yazılması gerekiyordu?**

A Programlar SQL bilmediği için

B Veritabanı her değişiklikte silindiği için

C Veriye giden yol (okulu bul, sınıfa in, öğrenciye geç) programın içine yazılı olduğu için

D Bu sistemler tablo kullandığı için

VERİTABANI SİSTEMLERİ

1 İlişkisel model ve ilişkisel cebir

Bu bölümde verinin tablolarla nasıl anlatıldığını, satırların birbirinden neyle ayrıldığını ve tabloların nasıl birbirine bağlandığını görüyoruz. Sonra tablo alıp tablo veren birkaç işlemle, yani ilişkisel cebirle, sorulara nasıl cevap bulunduğuna bakıyoruz.

İki Ahmet

videoda 5:27

Bir üniversitenin öğrenci listesinde iki satırda da ad Ahmet Yalçın, bölüm Bilgisayar. Veritabanı bu iki kişiyi nasıl karıştırmaz?

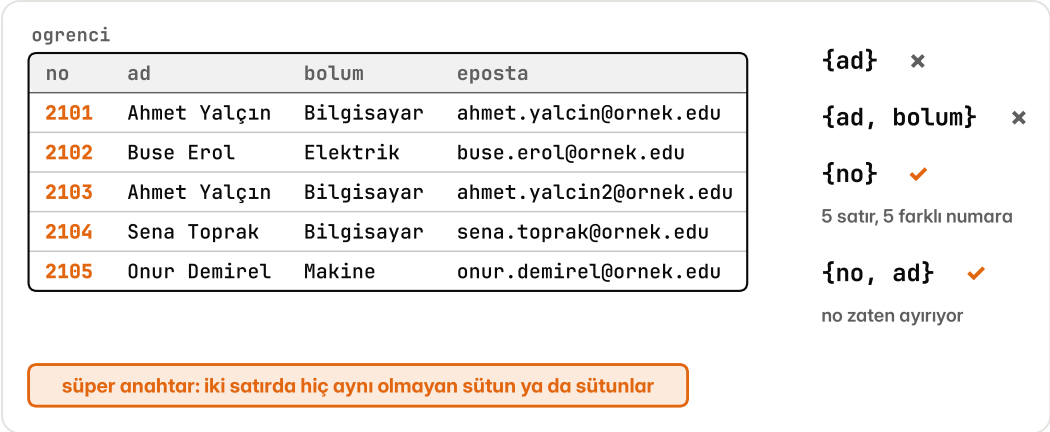
no	ad	bolum	eposta
2101	Ahmet Yalçın	Bilgisayar	ahmet.yalcin@ornek.edu
2102	Buse Erol	Elektrik	buse.erol@ornek.edu
2103	Ahmet Yalçın	Bilgisayar	ahmet.yalcin2@ornek.edu
2104	Sena Toprak	Bilgisayar	sena.toprak@ornek.edu
2105	Onur Demirel	Makine	onur.demirel@ornek.edu

İlişkisel modelde veri tablolarda durur ve bu tabloya **ilişki** (relation) denir. Her satır bir kayıttır, burada bir öğrenci; her sütun bir **nitelik**tir: numara, ad, bölüm, e-posta. Her niteliğin bir **değer alanı** vardır, yani alabileceği değerlerin kümesi; numara için 4 basamaklı tam sayılar. Tablonun kalıbı kısaca `Ogrenci(no, ad, bolum, eposta)` diye yazılır.

Tablo üç kurala uyar. Satırların sırası yoktur; ada göre de numaraya göre de dizsek aynı tablodur. Aynı satır iki kez bulunmaz. Her hücrede tek değer durur; bir öğrencinin üç dersi tek hücreye "VT, ALG, AYR" diye yazılmaz, ayrı bir tabloda üç satır olur. İlk iki kural keyfi değil: ilişki, matematikteki bir kümedir; kümede sıra yoktur ve aynı öğe iki kez sayılmaz.

Bu modeli 1970'te IBM'in araştırma laboratuvarında Edgar F. Codd yayımladı. O yıllarda programlar veriye, diskteki kayıtları birbirine bağlayan yolları izleyerek ulaşıyordu; düzen değişince programlar da değişiyordu. Codd'un fikri: veriyi kullanan, onun makinede nasıl durduğunu değil, ne olduğunu bilmeli. Satırları birbirine bağlayan gizli bir işaret yoktur; her şey değerlerle anlatılır.

Öyleyse her satırı ötekilerden ayıran bir değer gerekir. Ad ayırmıyor; ad ve bölüm birlikte de ayırmıyor. Numara ise her öğrencide farklı. İki satırda hiçbir zaman aynı olmayan sütuna ya da sütun grubuna **süper anahtar** (superkey) denir. {no} bir süper anahtardır; {no, ad} da öyle, çünkü numara zaten tek başına ayırıyor.



Şekil 1.1 Öğrenci tablosunda {ad} ve {ad, bolum} ayırmıyor, {no} ve {no, ad} ayırıyor; altta süper anahtarın tanımı.

Fazlalık atılınca **aday anahtar** (candidate key) kalır: hiçbir sütunu çıkarılamayan süper anahtar. {no, ad} aday değildir, çünkü ad çıkarılınca kalan {no} yine ayırır. Burada iki aday var: {no} ve {eposta}, çünkü her öğrencinin adresi ayrıdır. Tasarımcı birini seçer; seçilene **birincil anahtar** (primary key) denir. Burada numara. Anahtar bugünkü satırlara değil, kurala bakılarak seçilir: aynı adlı iki kişi hiç olmasa bile ad anahtar olamaz, çünkü bir gün olabilir.

Kimin hangi derse yazıldığı ayrı bir tabloda, `Kayit(no, kod, notu)` tablosunda durur:

no	kod	notu
2101	VT	BA
2101	ALG	AA
2101	AYR	NULL
2102	VT	CB
2102	ALG	BB
2102	FOT	AA
2103	VT	NULL
2104	VT	AA

(11 satırın ilk 8'i)

Burada numara tek başına ayırmaz, 2101 üç derse yazılmış; ders kodu da ayırmaz, veritabanı dersini (VT) 4 kişi alıyor. Ama ikisi birlikte hiç tekrar etmez, çünkü bir öğrenci aynı derse iki kez yazılamaz. Birden çok sütundan oluşan böyle bir anahtara **bileşik anahtar** (composite key) denir: (no, kod). Kayıttaki numara bir iş daha görür: öğrenci tablosundaki bir satırı gösterir. Başka bir tablonun anahtarını gösteren sütuna **yabancı anahtar** (foreign key) denir.



Şekil 1.2 Kayıt tablosunda {no} ve {kod} tekrar ediyor, {no, kod} etmiyor; kayit.no'dan ogrenci.no'ya giden yabancı anahtar oku.

Kayıt tablosunda iki not boş: dersler sürüyor, notlar henüz verilmedi. Böyle boş hücreye **NULL** denir: bilinmeyen bir değer, ya da ikinci telefonu olmayan birinin ikinci telefonu gibi hiç olmayan bir değer. NULL sıfır da boş yazı da değildir; 0 bir nottur, NULL notun henüz olmadığını söyler.

Anahtarların işe yaraması için veritabanı iki kuralı korur. **Varlık bütünlüğü**: birincil anahtar NULL olamaz, çünkü numarası boş bir satır hangi kişi olduğunu söyleyemez. **Referans bütünlüğü**: yabancı anahtar var olan bir satırı göstermeli; öğrenci tablosunda olmayan 2109 numarasına kayıt yazılamaz, dersleri olan 2104'ün satırı da silinemez. SQLite üç denemeyi de reddeder (yabancı anahtar denetimi açıkken):

```
INSERT INTO ogrenci VALUES (NULL, 'Derin Uçar', 'Kimya', 'derin.ucar@ornek.edu');
HATA: IntegrityError: NOT NULL constraint failed: ogrenci.no
INSERT INTO kayit VALUES (2109, 'VT', NULL);
HATA: IntegrityError: FOREIGN KEY constraint failed
DELETE FROM ogrenci WHERE no = 2104;
HATA: IntegrityError: FOREIGN KEY constraint failed
```



Şekil 1.3 Numarasız öğrenci, olmayan 2109'a kayıt ve 2104'ün silinmesi; üçü de reddediliyor, sağda iki şema kartı ve yabancı anahtar.

İki Ahmet Yalçın'ı adları değil numaraları ayırır: 2101 ve 2103. İkinci bir 2101 eklenmek istenirse veritabanı reddeder:

```
HATA: IntegrityError: UNIQUE constraint failed: ogrenci.no
```

Bir soru

videoda 8:59

Kayıt tablosunda 11 satır var, her satırda bir öğrenci numarası ve bir ders kodu. Soru şu: veritabanı dersine (VT) yazılıp algoritmalar dersine (ALG) yazılmayan öğrenci kim? Bunu bulmak için yalnız üç basit işlem yeter.

Bu işlemler **ilişkisel cebiri** (relational algebra) kurar. Toplama nasıl iki sayı alıp bir sayı veriyorsa, bu işlemler de bir ya da iki tablo alıp yeni bir tablo verir. Sonuç yine tablo olduğu için işlemler art arda dizilebilir; birinin çıktısı ötekinin girdisi olur. SQL ile yazılan bir sorgu da veritabanının içinde böyle bir işlem dizisine çevrilir. İlişkisel cebir, SQL'in altındaki matematiktir.

İlk işlem **seçme** (σ): bir koşul alır, koşulu sağlayan satırları bırakır, ötekileri atar. Koşul harfin altına yazılır; düz yazıda `σ bolum = 'Bilgisayar' (Ogrenci)`. Öğrenci tablosundan bölümü Bilgisayar olanları seçince 5 satırdan 3'ü kalır: 2101, 2103 ve 2104. Sütunlara dokunulmaz, yalnız satırlar süzülür.

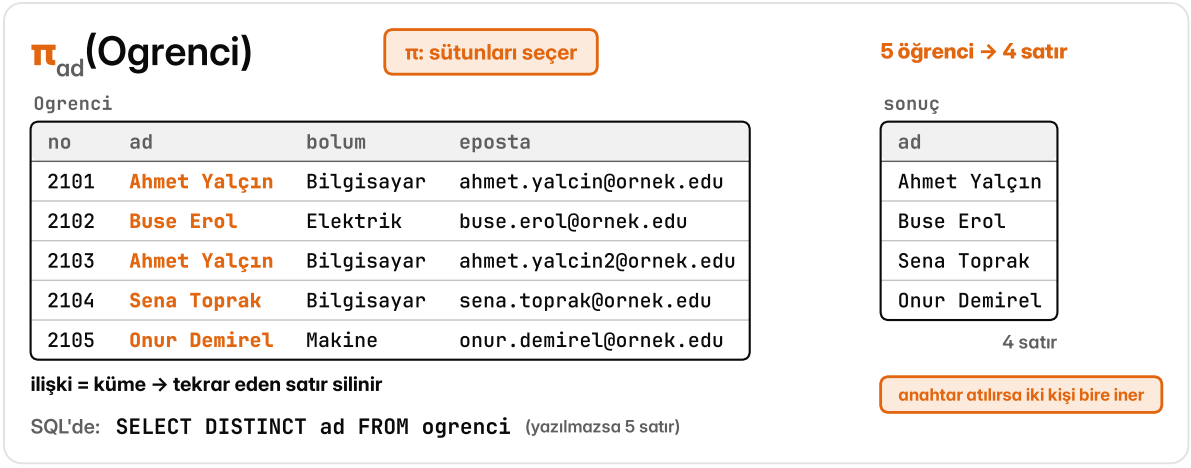
İkincisi **izdüşüm** (π): istenen sütunları bırakır, ötekileri atar. `π ad (Ogrenci)` yalnız ad sütununu alır. 5 öğrenci var ama sonuç 4 satır, çünkü iki Ahmet Yalçın tek satıra iner:

ad
Ahmet Yalçın
Buse Erol
Sena Toprak
Onur Demirel

Sebebi tablonun bir küme olması; kümede aynı öğe iki kez bulunmaz, izdüşüm de tekrar eden satırları siler. SQL ise tekrarları bırakır; silmek için DISTINCT yazılır, yazılmazsa sonuç 5 satırdır:

```
SELECT DISTINCT ad FROM ogrenci
```

Buradan bir ders çıkar: izdüşümde anahtar atılırsa iki kişi tek satıra inebilir.



Şekil 1.4 π ad (Ogrenci): 5 öğrenciden 4 satırlık sonuç, iki Ahmet Yalçın tek satır; altta SQL'deki karşılığı SELECT DISTINCT.

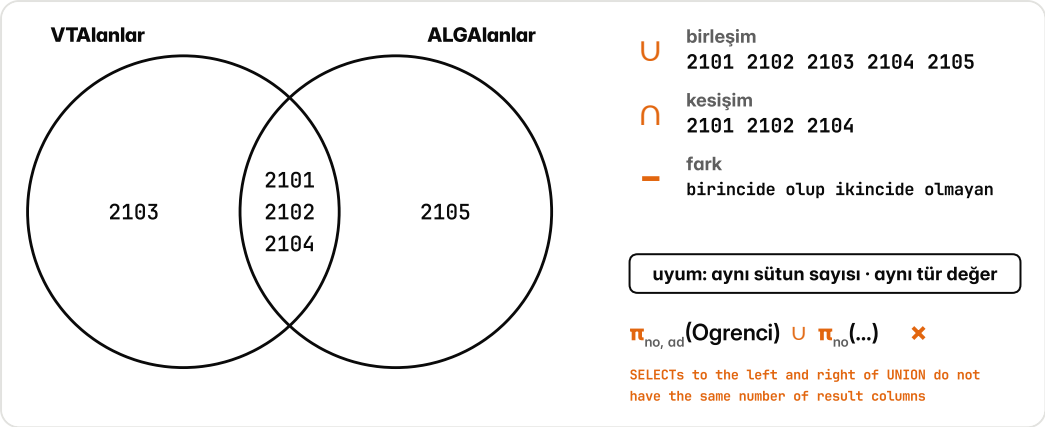
Üçüncüsü **yeniden adlandırma** (ρ): bir tabloya ya da sütunlarına yeni bir ad verir, içeriğe dokunmaz. Uzun bir ifadenin ara sonucuna ad koyup sonra o adla anmaya yarar; aynı tablo bir ifadede iki kez geçecekse iki kopyayı ayırmaya da. Kayıt tablosundan VT satırlarını seçip izdüşümle yalnız numaraları alalım ve sonuca VTAlanlar diyelim:

```
ρ VTAlanlar (π no (σ kod = 'VT' (Kayit)))    → 2101, 2102, 2103, 2104
ρ ALGAlanlar (π no (σ kod = 'ALG' (Kayit)))  → 2101, 2102, 2104, 2105
```

İki tablo da tek sütunlu birer numara listesi ve küme oldukları için **küme işlemleri** de kullanılır. **Birleşim** (∪) iki tablodan herhangi birinde olanları verir: 5 öğrencinin hepsi. **Kesişim** (∩) ikisinde de olanları verir: 2101, 2102 ve 2104. **Fark** (−) birincide olup ikincide olmayanları verir. Bu işlemler yalnız **uyumlu** tablolar arasında yapılır: sütun sayıları aynı olmalı, karşılıklı sütunlar aynı türden değer taşımalı. İki sütunlu bir tabloyla tek sütunlu bir tablonun birleşimini SQLite reddeder:

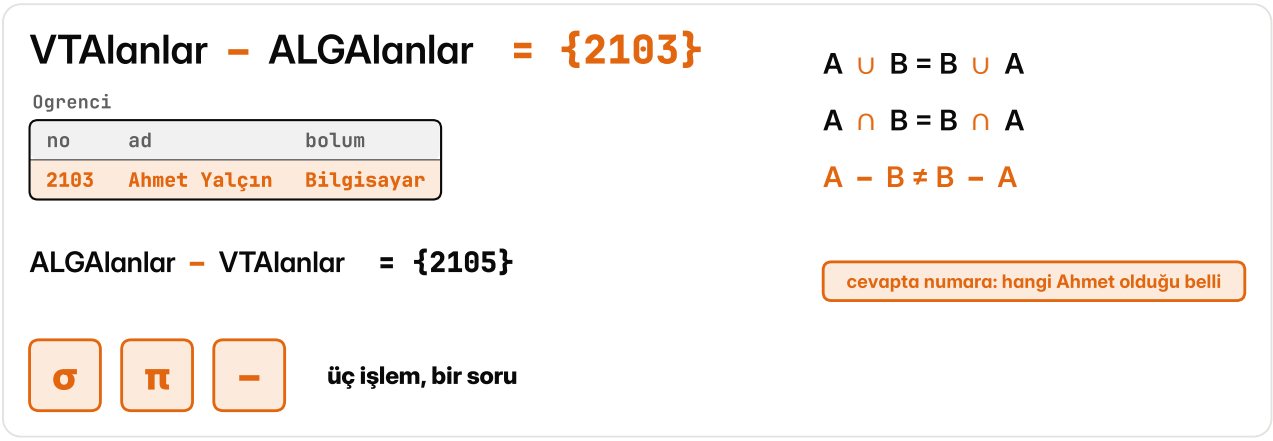
```
SELECT no, ad FROM ogrenci
UNION
SELECT no FROM kayit WHERE kod = 'VT';
HATA: OperationalError: SELECTs to the left and right of UNION do not have the same
number of result columns
```

Numara listesiyle ad listesini birleştirmek ise sütun sayıları tutsa da anlamsızdır.



Şekil 1.5 VTAlanlar ve ALGAlanlar iki daire: yalnız solda 2103, ortada 2101, 2102, 2104, yalnız sağda 2105; yanda birleşim, kesişim ve fark.

Şimdi soru tek bir ifade: `VTAlanlar - ALGAlanlar`. Sonuç tek satır: 2103. Öğrenci tablosunda bu numara ikinci Ahmet Yalçın'ın. Sırayı ters çevirirsek, `ALGAlanlar - VTAlanlar`, başka biri kalır: 2105. Birleşimde ve kesişimde sıra önemli değildir, farkta önemlidir. Seçme, izdüşüm ve fark: üç işlem, bir soru. Cevapta ad değil numara durduğu için hangi Ahmet olduğu da belli.



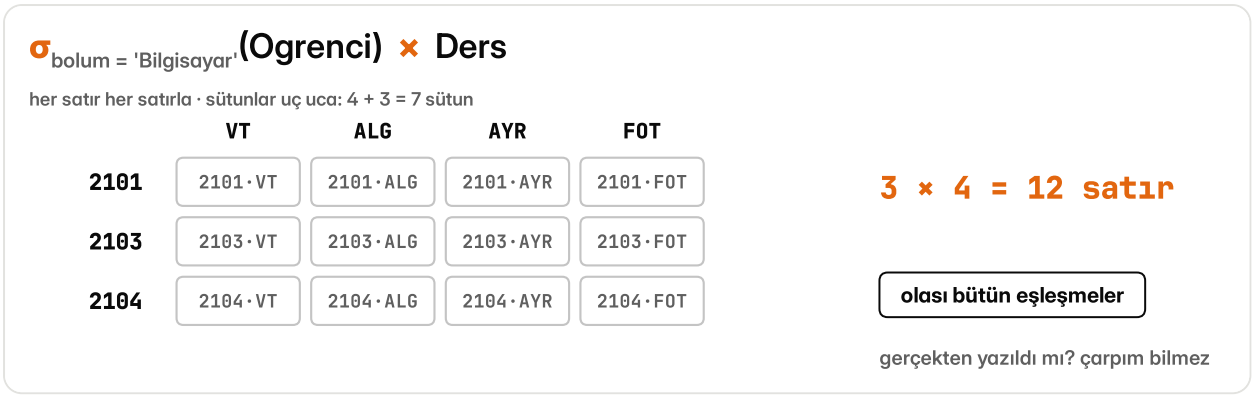
Şekil 1.6 VTAlanlar − ALGAlanlar = {2103}, 2103'ün öğrenci satırı; ters yön {2105}; σ, π ve − kutuları.

Bütün zorunlu dersler

videoda 11:39

Üniversitede 5 öğrenci ve 4 ders var; derslerden üçü zorunlu (VT, ALG, AYR), biri seçmeli (FOT, Fotoğrafçılık). Kayıt tablosu kimin hangi derse yazıldığını tutuyor. Bütün zorunlu dersleri almış öğrenciler kim? Bunun için tabloları bir araya getirmek gerekir.

En kaba yol **kartezyen çarpım** (×): birinci tablonun her satırı ikincinin her satırıyla yan yana konur, sütunlar uç uca eklenir. `σ bolum = 'Bilgisayar' (Ogrenci) × Ders` ifadesinde 3 öğrenci 4 dersle çarpılır: ilk öğrenci 4 dersle 4 satır verir, ikincisi 4, üçüncüsü 4; 3 × 4 = 12 satır. Sütun sayısı da 4 + 3 = 7 olur. Bu satırların çoğu anlamsızdır; yalnız bütün olası eşleşmeleri dizer.



Şekil 1.7 Bilgisayar bölümünün 3 öğrencisi ile 4 ders: 3 × 4 = 12 hücrelik çarpım ızgarası.

Anlamlı eşleşmeyi **birleştirme** (join, ⋈) verir: çarpımın ardından bir seçme, `R ⋈θ S = σθ (R × S)`. Öğrenci tablosuyla kayıt tablosunu çarpınca 5 × 11 = 55 satır çıkar. Öğrencinin numarasıyla kaydın numarası aynı olanları seçince 11 satır kalır; her kayıt kendi öğrencisinin yanına oturur. Koşulun açıkça yazıldığı bu türe **koşullu birleştirme** denir; iki numara sütunu da sonuçta kalır. **Doğal birleştirmede** (natural join) koşul yazılmaz: iki tablodaki aynı adlı sütunlar eşit olmalıdır ve o sütundan tek kopya kalır. `Ogrenci ⋈ Kayit` için ortak sütun no; sonuç yine 11 satır.

Şimdi soruya dönelim. Kayıt tablosundan numara ve ders kodunu alalım; zorunlu derslerin kodlarından da küçük bir tablo yapalım:

```
π no, kod (Kayit) ÷ π kod (σ zorunlu = 'evet' (Ders))
```

Bölme (÷), ikinci tablonun her satırıyla eşleşen numaraları verir, yani üç zorunlu dersin üçüne de yazılmış öğrencileri:

no	VT	ALG	AYR
2101	var	var	var
2102	var	var	yok
2103	var	yok	yok
2104	var	var	var
2105	yok	var	yok

2102 de üç derse yazılmış ama biri seçmeli Fotoğrafçılık; Ayrık Matematik eksik. Sonuç 2 öğrenci: 2101 (Ahmet Yalçın) ve 2104 (Sena Toprak). Adı çarpımın tersi olmasından gelir: bu iki numarayı üç zorunlu dersle çarparsak çıkan 6 satırın hepsi kayıt tablosunda vardır. Soruda "bütün" sözcüğü geçiyorsa çoğu zaman bölme gerekir.

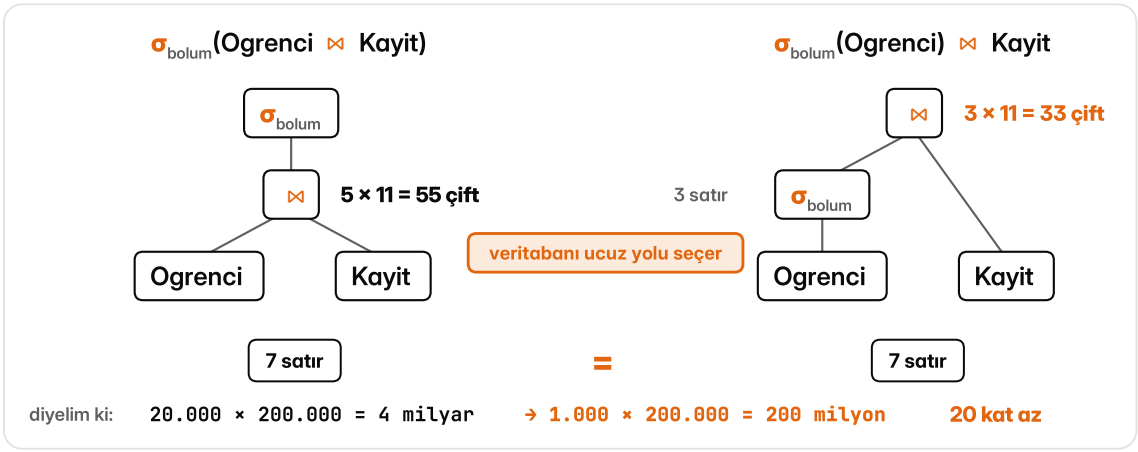


Şekil 1.8 Bölme: her öğrencinin üç zorunlu derste işaretleri, 2101 ve 2104 tam; sağda sonuç ve bölen tablo.

İlişkisel cebirde aynı soruyu birden çok ifade cevaplayabilir. Bilgisayar öğrencilerinin kayıtlarını isteyelim:

```
σ bolum = 'Bilgisayar' (Ogrenci ⋈ Kayit)
σ bolum = 'Bilgisayar' (Ogrenci) ⋈ Kayit
```

Birincisi önce birleştirir, sonra seçer; çarpımdan düşünürsek 5 × 11 = 55 çift denenir. İkincisi önce Bilgisayar öğrencilerini seçer, sonra birleştirir: 3 × 11 = 33 çift. Sonuç ikisinde de aynı 7 satır. Diyelim ki üniversite 20.000 öğrencili ve 200.000 kayıtlı: birinci yol 4 milyar çift dener, 1.000 öğrencili bir bölüm önce seçilirse 200 milyon, 20 kat az. Veritabanı, sorguyu çalıştırmadan önce böyle eşdeğer ifadeler arasından ucuz olanı seçer.



Şekil 1.9 İki işlem ağacı: önce birleştirip 55 çift, önce seçip 33 çift; ikisinin sonucu da 7 satır.

Cebir bir tarif verir: önce şunu yap, sonra bunu. Aynı soru sonucun taşıması gereken koşulu söyleyerek de yazılabilir. Buna **ilişkisel hesap** (relational calculus) denir. Bilgisayar öğrencilerinin adları ilişkisel hesapta şöyle yazılır:

```
{ t.ad | t ∈ Ogrenci ∧ t.bolum = 'Bilgisayar' }
```

"Öğrenci tablosundaki öyle satırların adları ki bölümleri Bilgisayar olsun." Sonuç Ahmet Yalçın ve Sena Toprak. Nasıl bulunacağı yazılmaz, yalnız ne istendiği. Codd 1972'de, ilişkisel hesapla yazılabilen her sorunun ilişkisel cebirle de yazılabildiğini gösterdi.

SQL de ilişkisel hesap gibi ne istendiğini yazar; veritabanı onu içeride cebir işlemlerine çevirir ve eşdeğer ifadeler arasından ucuz olanı seçer. Bütün zorunlu dersleri almış iki öğrenciyi bulan da bu işlemlerdir: birleştirme, seçme ve bölme.

Özet

- İlişki bir tablodur: satırları kayıt, sütunları nitelik; bir küme olduğu için satırların sırası yoktur ve aynı satır iki kez bulunmaz.
- Süper anahtar satırları ayıran sütunlardır; fazlası atılınca aday anahtar kalır, seçilen aday birincil anahtardır; başka tablonun anahtarını gösteren sütun yabancı anahtardır.
- NULL bilinmeyen ya da olmayan değerdir; birincil anahtar NULL olamaz (varlık bütünlüğü), yabancı anahtar var olan bir satırı göstermelidir (referans bütünlüğü).
- İlişkisel cebir tablo alıp tablo veren işlemlerdir: seçme satır süzer, izdüşüm sütun seçer ve tekrarı siler, yeniden adlandırma ad verir; birleşim, kesişim ve fark uyumlu tablolarda çalışır.
- Birleştirme çarpımın ardından seçmedir; "bütün" soruları bölmeyle cevaplanır; önce süzmek aynı sonuca daha az işle ulaşır.
- İlişkisel hesap ne istendiğini yazar; Codd'a göre hesapla yazılan her soru cebirle de yazılır.

Sorular

1 **Kayıt tablosunda 2103'ün VT satırının not hücresinde NULL yazıyor. Bu ne anlatır?**

A 2103 dersten 0 aldı

B Kayıt silinmiş

C Not henüz verilmedi, değer bilinmiyor

D Not boş bir yazı olarak girildi

2 **Öğrenci tablosunda (no, ad, bolum, eposta) hangi sütun grubu süper anahtar değildir?**

A {no}

B {ad, bolum}

C {no, ad}

D {eposta}

3 **{no, ad} bir süper anahtar. Neden aday anahtar değildir?**

A İki sütunlu olduğu için; aday anahtar tek sütunlu olur

B Ad sütununda NULL olabileceği için

C Tasarımcı onu birincil anahtar seçmediği için

D Ad çıkarılınca kalan {no} da satırları ayırdığı için

4 π bolum (Ogrenci) kaç satır döndürür?

Cevap: _____

5 Ogrenci × Ders kartezyen çarpımı kaç satırdır? (Ogrenci 5, Ders 4 satır)

A 9 **B** 11

C 20 **D** 55

6 Yabancı anahtar denetimi açıkken kayıt tablosuna aşağıdaki eklemelerden hangisi reddedilir?

A (2105, 'VT', NULL) **B** (2103, 'ALG', NULL)

C (2104, 'FOT', 'AA') **D** (2106, 'ALG', NULL)

7 π no (σ kod = 'AYR' (Kayit)) ∩ π no (σ kod = 'ALG' (Kayit)) ifadesinin sonucundaki numaraları yaz.

Cevap: _____

8 π no, kod (Kayit) ÷ π kod (σ kod = 'VT' ∨ kod = 'ALG' (Ders)) kaç öğrenci döndürür?

Cevap: _____

VERİTABANI SİSTEMLERİ

2 SQL temel: tablo kurmak ve sorgulamak

Bu bölümde SQL ile ilk sorguyu yazıyoruz: bir tablodan istenen satırları ve sütunları seçmek, koşulları birbirine bağlamak, sonucu sıralamak. Sonra bir tablo kuruyor, satır ekliyor, değiştiriyor ve siliyoruz; bu sırada unutulan bir WHERE'in neden bütün tabloyu değiştirdiğini görüyoruz. Son olarak NULL'un neden = ile aranamadığını, desenle aramayı ve tekrarsız, sıralı, kısa sonuç almayı öğreniyoruz.

Tek cümlelik istek

videoda 14:44

Bir çevrimiçi mağazada binlerce ürün olabilir. Müşteri yalnız şunu istiyor: fiyatı 500 liranın altındaki kulaklıklar. Bu istek veritabanına tek bir cümleyle nasıl söylenir?

Bu bölümde ve sonraki bölümlerde küçük bir **mağaza veritabanı** kullanıyoruz. Beş tablosu var:

tablo	sütunlar	satır
kategori	id, ad, ust_id	8
urun	id, ad, kategori_id, fiyat, stok	12
musteri	id, ad, sehir, kayit_tarihi	8
siparis	id, musteri_id, tarih	10
siparis_kalemi	siparis_id, urun_id, adet, birim_fiyat	20

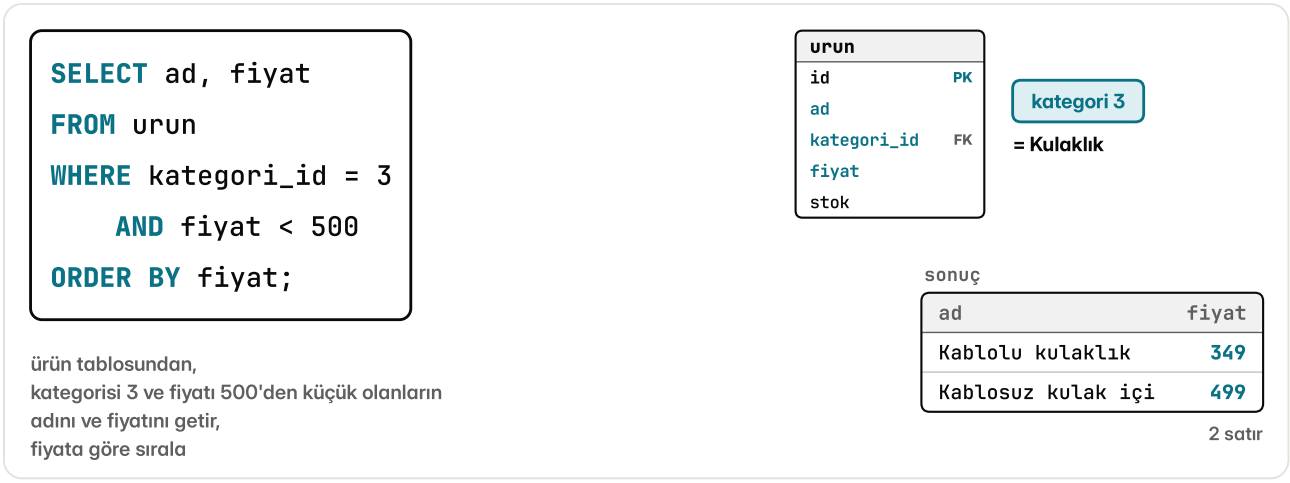
Her satır bir kayıttır: `urun` tablosunda her satır bir ürün, sütunlarda adı, kategorisinin numarası, fiyatı ve stoğu durur. Kategoriler iç içedir; `ust_id` bir kategorinin hangi kategorinin altında olduğunu söyler. `siparis` tablosu siparişi veren müşterinin yalnız numarasını tutar; `siparis_kalemi` bir siparişte hangi üründen kaç adet, hangi fiyata alındığını yazar.

Kulaklıklar 3 numaralı kategoride. İstek SQL'de şöyle yazılır:

```
SELECT ad, fiyat
FROM urun
WHERE kategori_id = 3
    AND fiyat < 500
ORDER BY fiyat;
```

Neredeyse İngilizce bir cümle gibi okunur: `urun` tablosundan (FROM), kategorisi 3 ve fiyatı 500'den küçük olanların (WHERE) adını ve fiyatını getir (SELECT), fiyata göre sırala (ORDER BY). Sonuç 2 satırdır:

ad	fiyat
Kablolu kulaklık	349
Kablosuz kulak içi	499



Şekil 2.1 Ürün tablosunun şeması, beş satırlık sorgu ve sonucu: Kablolu kulaklık 349, Kablosuz kulak içi 499.

Bu cümlede bir şey eksik: nasıl. Hangi dosyanın açılacağı, satırların hangi sırayla dolaşılacağı yazmıyor; döngü de sayaç da yok. Yalnız ne istediğimizi yazdık, nasıl bulunacağına veritabanı karar verir. Tablo bugün 12 satır, yarın 12 milyon olsun; sorgu aynı kalır.

Sorgu SELECT ile başlar, ama veritabanı onu bu sırayla çalıştırmaz. Önce FROM: hangi tablo? 12 ürünün hepsi masaya gelir. Sonra WHERE satırları süzer: kategorisi 3 olmayan 8 ürün düşer, 4 kulaklık kalır; fiyatı 500'ü geçen ikisi (2499 ve 1299) de düşer, 2 satır kalır. Ancak bundan sonra SELECT, kalan satırlardan yalnız ad ve fiyat sütunlarını alır. En son ORDER BY sıralar. Yazılış sırası SELECT, FROM, WHERE, ORDER BY; çalışma sırası FROM, WHERE, SELECT, ORDER BY.



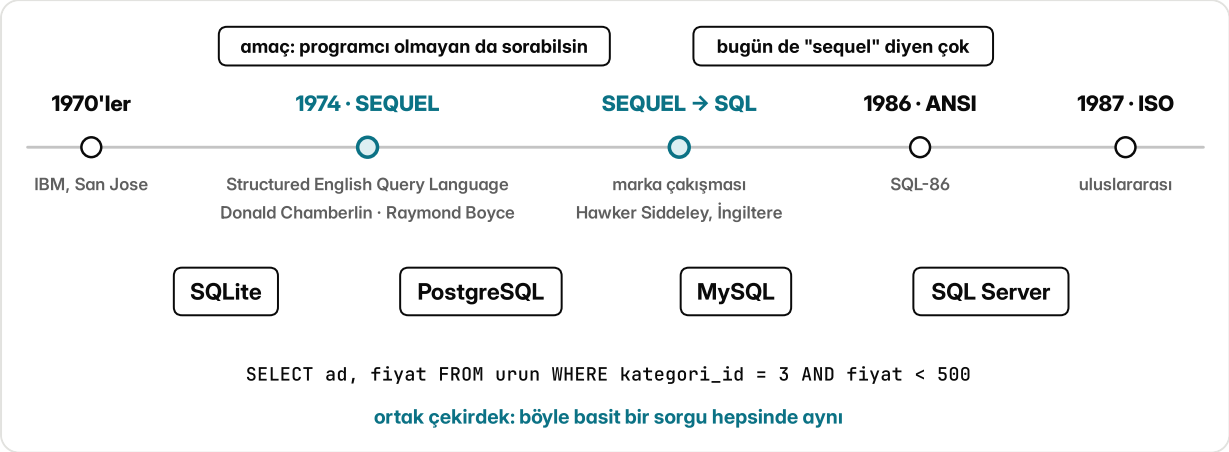
Şekil 2.2 Yazılış sırası ile çalışma sırası yan yana; FROM'un getirdiği 12 satır WHERE'in iki koşuluyla 2 satıra iner.

WHERE'de küçük, büyük, eşit ve eşit değil gibi karşılaştırmalar kullanılır (`=` , `<>` , `<` , `>` , `<=` , `>=`); koşullar da birbirine bağlanabilir. Aynı ürün tablosunda:

- AND iki koşulun ikisini de ister: 2 satır.
- OR birini yeterli sayar: 4 kulaklığa 500 liranın altındaki şarj aleti (449) ile kablosuz fare (389) de katılır; 6 satır.

- NOT koşulu tersine çevirir: `WHERE NOT kategori_id = 3` kategorisi 3 olmayan 8 ürünü getirir.
- BETWEEN bir aralık verir, iki ucu da dahil: `WHERE fiyat BETWEEN 1000 AND 2000` mekanik klavye (1150), oyuncu kulaklığı (1299) ve taşınabilir hoparlörü (1750) getirir.
- IN bir liste verir: `WHERE kategori_id IN (4, 5)` telefonları ve bilgisayar aksesuarlarını, 4 ürünü getirir.

Bu dil 1970'lerin başında IBM'in San Jose'deki araştırma laboratuvarında doğdu. Donald Chamberlin ve Raymond Boyce'un 1974'te tanıttığı dilin ilk adı SEQUEL'di (Structured English Query Language, yapısal İngilizce sorgu dili). Amaç, programcı olmayan birinin de veriye İngilizce bir cümleyle soru sorabilmesiydi. Ad, İngiliz havacılık şirketi Hawker Siddeley'nin markasıyla çakışınca kısaldı ve SQL oldu; bugün de birçok kişi ona "sequel" der. 1986'da Amerikan standartlar kurumu (ANSI), ertesi yıl uluslararası standartlar örgütü (ISO) dili standart yaptı. Bugün SQLite, PostgreSQL, MySQL ve SQL Server bu ortak çekirdekle yazılır; kulaklıkları getiren sorgu gibi basit bir sorgu hepsinde aynıdır. Bu bölümdeki örnekler SQLite'ta çalıştırıldı.



Şekil 2.3 1974'te SEQUEL, ad değişikliği, 1986 ANSI ve 1987 ISO standardı; aynı sorgu SQLite, PostgreSQL, MySQL ve SQL Server'da.

Bir komut, dört kupon

videoda 17:26

Bir kuponun indirimini %25'e çıkarmak isteyen tek satırlık bir komut, mağazadaki bütün kuponları nasıl %25'e çıkarır? Cevaba gitmek için önce kuponlara veritabanında yer açalım.

Tablo CREATE TABLE komutuyla kurulur: önce tablonun adı, sonra her sütunun adı ve tipi.

```
CREATE TABLE kupon (
    kod       VARCHAR(20) PRIMARY KEY,
    yuzde     INTEGER NOT NULL,
    alt_tutar DECIMAL(10,2),
    son_gun   DATE NOT NULL
);
```

Kupon tablosunun 4 sütunu var. Kod, en çok 20 karakterlik bir metin; yüzde, bir tam sayı; alt tutar, ondalık bir sayı: toplam 10 basamak, 2'si virgülden sonra, kuruşlar için; son gün, bir tarih. Tip, sütunun ne tür değer tuttuğunu söyler. SQLite bu konuda gevşektir: INTEGER sütuna 'yirmi'

yazılabilir. PostgreSQL gibi sunucular tipi sıkı denetler ve aynı denemeyi reddeder (PostgreSQL 19):

```
INSERT INTO kupon VALUES ('DENEME', 'yirmi', 0, '2026-12-31');
ERROR:  invalid input syntax for type integer: "yirmi"
```

Kodun yanındaki **birincil anahtar** (primary key), her kuponun kodunun tek olduğunu söyler; satır bu kodla bulunur. Yüzde ile son günün yanındaki NOT NULL kısıtı, bu sütunların boş bırakılamayacağını söyler. Satırlar INSERT komutuyla girer: önce sütunların adları, sonra değerler.

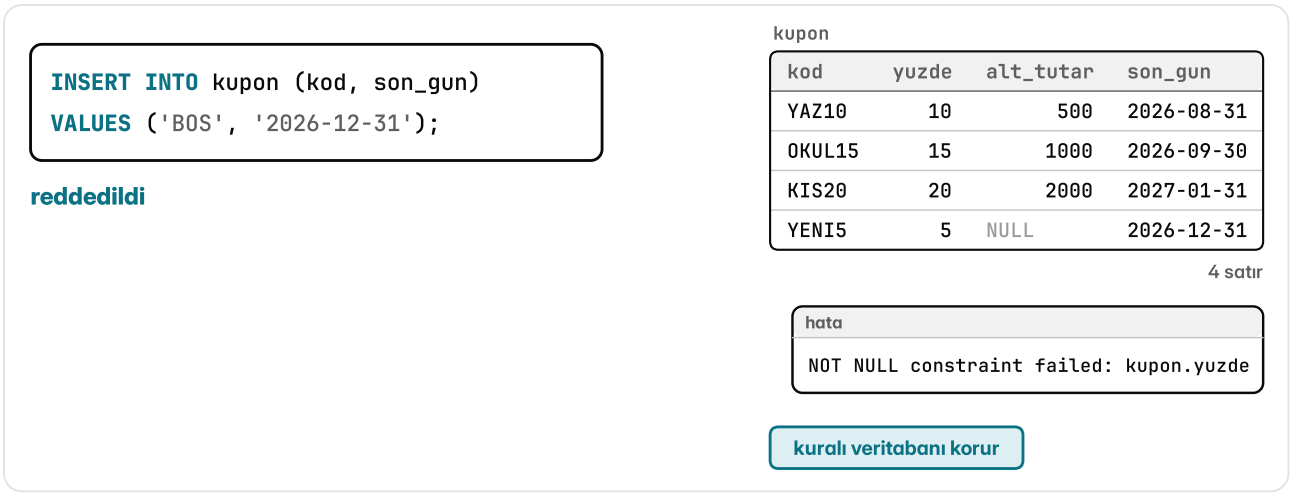
```
INSERT INTO kupon
    (kod, yuzde, alt_tutar, son_gun)
VALUES
    ('YAZ10', 10, 500, '2026-08-31'),
    ('OKUL15', 15, 1000, '2026-09-30'),
    ('KIS20', 20, 2000, '2027-01-31'),
    ('YENI5', 5, NULL, '2026-12-31');
```

kod	yuzde	alt_tutar	son_gun
YAZ10	10	500	2026-08-31
OKUL15	15	1000	2026-09-30
KIS20	20	2000	2027-01-31
YENI5	5	NULL	2026-12-31

YENI5'in alt tutarı yok; oraya NULL yazıldı, yani değer yok. Şimdi kuralları sınayalım. YAZ10 koduyla bir kupon daha eklemeye çalışınca veritabanı reddeder; yüzdesi verilmemiş bir kupon da reddedilir:

```
INSERT INTO kupon
VALUES ('YAZ10', 25, 0, '2026-12-31');
HATA: IntegrityError: UNIQUE constraint failed: kupon.kod

INSERT INTO kupon (kod, son_gun)
VALUES ('BOS', '2026-12-31');
HATA: IntegrityError: NOT NULL constraint failed: kupon.yuzde
```

İki denemeden sonra tabloda yine 4 satır var; kuralı program değil, veritabanının kendisi korur.



Şekil 2.4 Dört kuponla dolan tablo ve reddedilen iki ekleme: aynı kod ve boş yüzde.

Tablo kurulduktan sonra da değişebilir. `ALTER TABLE kupon ADD COLUMN aciklama TEXT;` kupona bir açıklama sütunu ekler; eski 4 satırın açıklaması yoktur, dördü de NULL ile dolar.

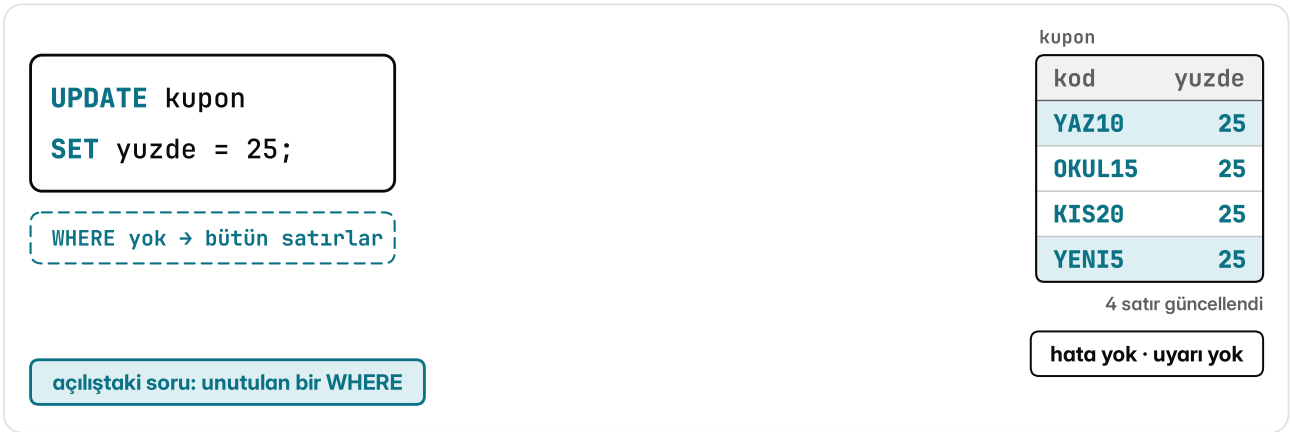
KIS20 kuponunun indirimi %25'e çıkacak:

```
UPDATE kupon
SET yuzde = 25
WHERE kod = 'KIS20';
```

Veritabanı 1 satırın güncellendiğini söyler. Şimdi aynı komutu WHERE kısmı olmadan çalıştıralım:

```
UPDATE kupon
SET yuzde = 25;
```

Hata yok, uyarı yok; komut yazıldığı gibi çalışır: 4 satır güncellendi. YAZ10 da, YENI5 de artık %25. WHERE, değişikliğin hangi satırlara uygulanacağını söyler; yazılmazsa cevap bütün satırlardır. Açılıştaki sorunun cevabı budur: unutulan bir WHERE.



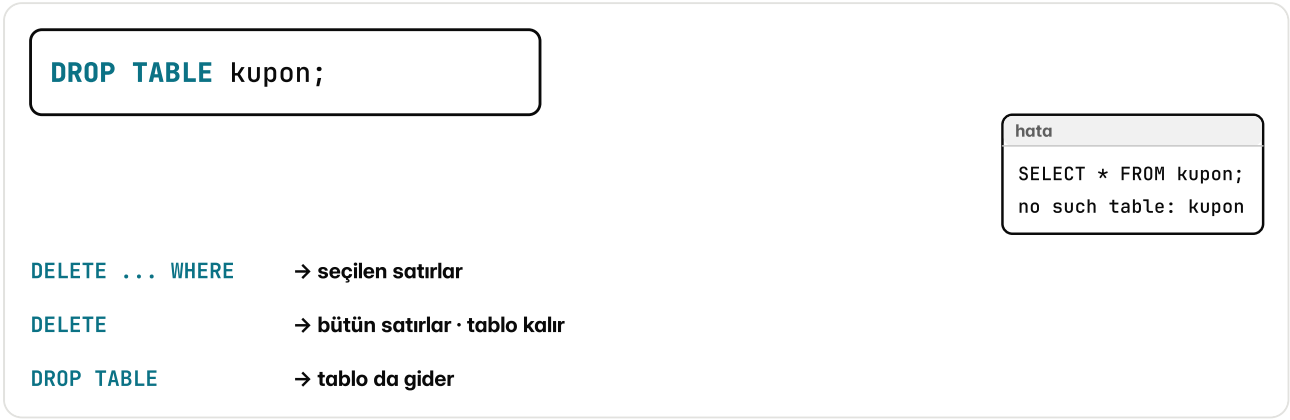
Şekil 2.5 WHERE'li UPDATE yalnız KIS20'yi değiştirir; WHERE'siz UPDATE dört kuponun yüzdesini birden 25 yapar.

Silme de aynı kurala uyar. `DELETE FROM kupon WHERE son_gun < '2026-10-01';` son günü ekimden önce olan YAZ10 ile OKUL15'i siler: 2 satır. Tarihler yıl-ay-gün sırasıyla yazıldığı için metin olarak karşılaştırmak tarih sırasıyla aynı sonucu verir. WHERE kısmı olmayan `DELETE FROM kupon;` kalan

bütün satırları siler; tablo boş kalır ama yerinde durur, sütunlarıyla birlikte. DROP TABLE ise tabloyu kendisiyle birlikte kaldırır; ardından gelen sorgu hata alır:

```
DROP TABLE kupon;

SELECT * FROM kupon;
HATA: OperationalError: no such table: kupon
```

Satırlarla birlikte tablonun kalıbı da gitmiştir.



Şekil 2.6 DELETE ... WHERE seçilen satırları, DELETE bütün satırları, DROP TABLE tabloyu satırlarıyla birlikte kaldırır.

Kayıp müşteri

videoda 19:38

Mağazanın müşteri tablosunda her müşterinin adı, şehri ve kayıt tarihi durur. 8 müşteriden birinin, Deniz Arslan'ın şehri boş:

id	ad	sehir
1	Ece Yıldız	Ankara
2	Mert Aydın	İstanbul
3	Zeynep Kaya	İzmir
4	Can Demir	Ankara
5	Selin Öztürk	Bursa
6	Emre Şahin	İstanbul
7	Deniz Arslan	NULL
8	Ozan Koç	İzmir

Şehri boş olanları getir diyen sorgu tek satır bile getirmez:

```
SELECT ad
FROM musteri
WHERE sehir = NULL;
```

Deniz tabloda dururken neden? Çünkü NULL bir değer değil, değerin bilinmediğini gösteren bir işarettir. Deniz'in şehri boş bir yazı değil; bilmiyoruz. Bilinmeyen bir şey neyle karşılaştırılırsa karşılaştırılsın, sonuç da bilinmez. SQLite'ta `SELECT 1 = 1 AS a, 1 = 2 AS b, NULL = NULL AS c;` şunu verir (1 doğru, 0 yanlış demektir):

a	b	c
1	0	NULL

SQL'in mantığında 2 değil 3 değer vardır: doğru, yanlış ve bilinmiyor. Buna **üç değerli mantık** denir. WHERE yalnız koşulu doğru olan satırları bırakır; bilinmeyenler de yanlışlar gibi düşer. `sehir = NULL` her satırda bilinmiyor çıkar, 8 satırın hepsi düşer.



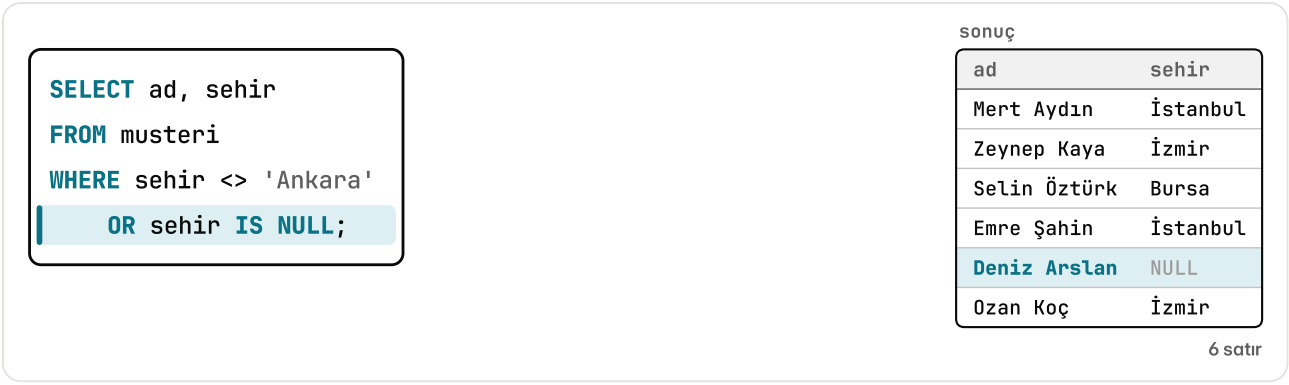
Şekil 2.7 Doğru, yanlış ve bilinmiyor: NULL'la her karşılaştırma bilinmiyor çıkar ve WHERE sekiz satırı da eler.

Boş olanı bulmanın yolu ayrı bir sınamadır: `WHERE sehir IS NULL`. IS NULL bir karşılaştırma yapmaz, yalnız işaretin olup olmadığına bakar; Deniz Arslan gelir.

Bu işaret daha sinsi bir yerde de iz bırakır. `WHERE sehir = 'Ankara'` 2 kişi getirir (Ece Yıldız, Can Demir), `WHERE sehir <> 'Ankara'` 5 kişi. 2 + 5 = 7 eder, ama müşteri 8. Kayıp yine Deniz: şehri bilinmediği için Ankaralı olup olmadığı da bilinmiyor, iki sorgudan da düşer. Onu da istiyorsak koşula açıkça ekleriz:

```
SELECT ad, sehir
FROM musteri
WHERE sehir <> 'Ankara'
    OR sehir IS NULL;
```

Bu kez 6 satır gelir; Deniz Arslan beşinci satırdadır.



Şekil 2.8 Ankara'ya eşit olanlar 2, eşit olmayanlar 5 kişi; iki sonuçta da olmayan Deniz, OR sehir IS NULL ile gelir.

Metinde tam eşitlik yerine bir desen de aranabilir: LIKE. Desende `%` herhangi sayıda karakter (hiç karakter de olabilir), `_` tam bir karakter demektir. `WHERE ad LIKE '%kulak%'` ürün adında "kulak" geçenleri arar: 4 kulaklığın dördü de gelir, kablosuz kulak içi dahil. `WHERE ad LIKE '_e%'` ikinci harfi e olan müşterileri getirir: Mert, Zeynep, Selin ve Deniz. SQLite'ta LIKE büyük ve küçük harfi yalnız İngilizce harflerde ayırmaz.

Müşteriler hangi şehirlerden? Şehri dolu 7 satırda Ankara, İstanbul ve İzmir ikişer kez geçer. DISTINCT tekrarları atar:

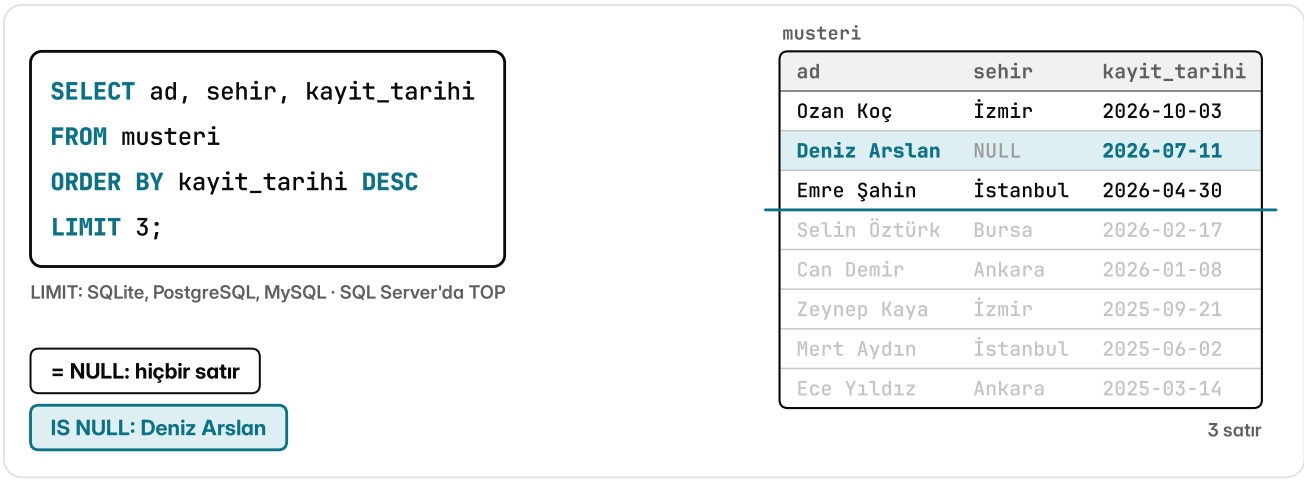
```
SELECT DISTINCT sehir
FROM musteri
WHERE sehir IS NOT NULL
ORDER BY sehir;
```

4 şehir kalır: Ankara, Bursa, İstanbul, İzmir. Son olarak en yeni 3 müşteri:

```
SELECT ad, sehir, kayit_tarihi
FROM musteri
ORDER BY kayit_tarihi DESC
LIMIT 3;
```

ad	sehir	kayit_tarihi
Ozan Koç	İzmir	2026-10-03
Deniz Arslan	NULL	2026-07-11
Emre Şahin	İstanbul	2026-04-30

ORDER BY kayıt tarihine göre sıralar; DESC sırayı büyükten küçüğe çevirir, LIMIT de 3. satırda keser. LIMIT SQLite, PostgreSQL ve MySQL'de aynıdır; SQL Server'da yerine TOP yazılır. Deniz'in şehri yine NULL: satır yerinde, yalnız o bilgi eksik. Onu soran sorgu = ile değil, IS NULL ile sorar.



Şekil 2.9 DISTINCT ile dört şehir ve kayıt tarihine göre en yeni üç müşteri; Deniz'in şehri NULL.

Özet

- SELECT hangi sütunları, FROM hangi tabloyu, WHERE hangi satırları istediğimizi söyler; ORDER BY sonucu sıralar. Nasıl bulunacağına veritabanı karar verir.
- Sorgu SELECT, FROM, WHERE, ORDER BY diye yazılır; FROM, WHERE, SELECT, ORDER BY sırasıyla çalışır.
- CREATE TABLE tabloyu sütun adları ve tipleriyle kurar; PRIMARY KEY ve NOT NULL kurallarını veritabanı korur.
- UPDATE ve DELETE, WHERE yazılmazsa bütün satırlara uygulanır; DROP TABLE tabloyu da kaldırır.
- NULL değer değil, bilinmeyenin işaretidir: = NULL hiçbir satır getirmez, IS NULL getirir; <> koşulu da NULL'lu satırları atlar.
- LIKE desenle arar (% herhangi sayıda, _ tek karakter); DISTINCT tekrarları atar, DESC ters sıralar, LIMIT ilk satırlarla keser.

Sorular

1 **Veritabanı SELECT ad, fiyat FROM urun WHERE kategori_id = 3 ORDER BY fiyat; sorgusunun parçalarını hangi sırayla çalıştırır?**

A SELECT, FROM, WHERE, ORDER BY
B FROM, SELECT, WHERE, ORDER BY
C FROM, WHERE, SELECT, ORDER BY
D WHERE, FROM, SELECT, ORDER BY

2 **Mağaza veritabanında SELECT ad, fiyat FROM urun WHERE fiyat BETWEEN 349 AND 499; kaç satır döndürür? (Fiyatlar: 349, 389, 449, 499 ve ötekiler 899'dan büyük.)**

A 2
B 4
C 3
D 0

3 **SELECT ... WHERE x = NULL koşulundaki x = NULL karşılaştırması bir satırda neye dönüşür?**

A Bilinmiyor (NULL)

B x boşsa doğru, doluysa yanlış

C Her zaman yanlış

D Hata verir

4 **Müşteri tablosunda 8 kişi var: 2'si İzmirli, birinin (Deniz Arslan) şehri NULL. SELECT ad, sehir FROM musteri WHERE sehir <> 'İzmir'; kaç satır döndürür?**

A 6

B 7

C 8

D 5

5 **Kupon tablosunda dört satır var: YAZ10 (yuzde 10), OKUL15 (15), KIS20 (20), YENI5 (5). UPDATE kupon SET yuzde = 30 WHERE yuzde < 15; kaç satırı günceller? Sayıyı yaz.**

Cevap: _____

6 **Mağazanın ürün adları: Kablolu kulaklık, Kablosuz kulak içi, Gürültü önleyici kulaklık, Oyuncu kulaklığı, Taşınabilir hoparlör, Akıllı telefon, Hızlı şarj aleti, Mekanik klavye, Kablosuz fare, Pamuklu gömlek, Kışlık mont, Koşu ayakkabısı. SELECT ad FROM urun WHERE ad LIKE 'K%'; kaç satır döndürür? Sayıyı yaz.**

Cevap: _____

7 **Dört satırlık kupon tablosunda DELETE FROM kupon; çalıştırıldıktan sonra SELECT * FROM kupon; ne verir?**

A no such table: kupon hatası

B 0 satır; sütun başlıkları yerinde durur

C Yalnız son satır kalır

D Komut WHERE istediği için silme yapılmamıştır, 4 satır

8 **Fiyatı 1000'den küçük ürünler: 349 ve 499 (kategori 3), 449 (kategori 4), 389 (kategori 5), 899 (kategori 7). SELECT DISTINCT kategori_id FROM urun WHERE fiyat < 1000; kaç satır döndürür? Sayıyı yaz.**

Cevap: _____

VERİTABANI SİSTEMLERİ

3 SQL orta: birleştirme ve gruplama

Bu bölümde iki tabloyu birleştirip satırlarını yan yana getiriyoruz: iç birleştirmede eşi olmayan satır düşer, dış birleştirmede boşlukları NULL doldurur. Sonra birçok satırı tek sayıya indiren toplama fonksiyonlarını, satırları gruplara ayıran GROUP BY'ı ve grupları süzen HAVING'i görüyor, mağazanın en çok satan beş ürününü buluyoruz. Son olarak bir tabloyu kendisiyle birleştiriyor ve iki sorgunun sonuçlarını küme işlemleriyle karşılaştırıyoruz. Örnekler mağaza veritabanında, SQLite'ta çalıştırıldı.

Numara var, ad yok

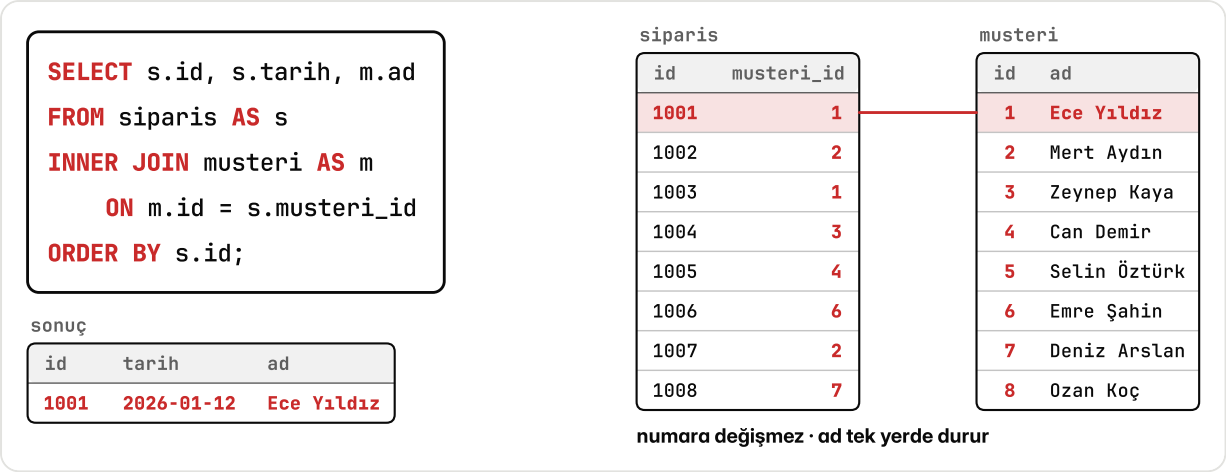
videoda 21:41

Mağazanın sipariş tablosunda 10 sipariş var; her satırda siparişin numarası, tarihi ve siparişi veren müşterinin numarası yazar. Müşterinin adı ise 8 satırlık müşteri tablosunda durur. Bu iki tabloyu eşleştirince kaç satır çıkar? Cevap tek değil: 9, 11, 10 ya da 12. Hangisi olacağını, istediğimiz birleştirme belirler.

Siparişe neden ad değil de numara yazılıyor? Ad her siparişe kopyalansaydı, müşteri adını değiştirdiğinde bütün siparişlerini tek tek düzeltmek gerekirdi; numara ise hiç değişmez ve ad tek yerde durur. Adı görmek için iki tablo **birleştirilir** (join):

```
SELECT s.id, s.tarih, m.ad
FROM siparis AS s
INNER JOIN musteri AS m
    ON m.id = s.musteri_id
ORDER BY s.id;
```

AS tabloya sorgu içinde kısa bir ad verir: `s` sipariş, `m` müşteri. INNER JOIN satırı iki tabloyu bağlar, ON'dan sonraki koşul eşleşmenin kuralını yazar: siparişteki müşteri numarası, müşterinin kendi numarasına eşit olmalı. Veritabanı her sipariş için numarası tutan müşteriyi arar. 1001 numaralı siparişin müşteri numarası 1; 1 numaralı müşteri Ece Yıldız. Eşleşen iki satır yan yana gelip tek satır olur.



Şekil 3.1 Sipariş ve müşteri tabloları; 1001'in müşteri numarası 1, Ece Yıldız'ın satırına çizgiyle bağlanır.

Sonuç 9 satırdır, 10 değil (9 satırın ilk 4'ü):

id	tarih	ad
1001	2026-01-12	Ece Yıldız
1002	2026-02-03	Mert Aydın
1003	2026-03-27	Ece Yıldız
1004	2026-04-15	Zeynep Kaya

1010 numaralı sipariş üye olmadan yapılmış bir misafir alışverişi; müşteri numarası boş, yani NULL. NULL hiçbir numaraya eşit sayılmaz, bu yüzden bu sipariş sonuçtan düşer. Hiç sipariş vermemiş Selin Öztürk ile Ozan Koç da listede yoktur. **İç birleştirmenin** (inner join) kuralı budur: yalnız iki tarafta da eşi olan satırlar kalır. INNER sözcüğü yazılmasa da olur; yalnız JOIN yazmak da iç birleştirme demektir.

Peki bütün müşteriler listede kalsın istiyorsak? Bunun için **dış birleştirme** (outer join) var:

```
SELECT m.ad, s.id, s.tarih
FROM musteri AS m
LEFT JOIN siparis AS s
    ON s.musteri_id = m.id
ORDER BY s.id;
```

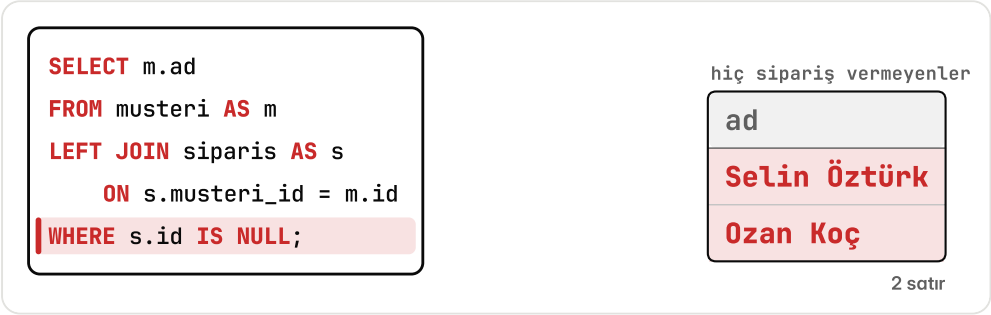
Sol, yazılışta önce gelen tablodur; sol dış birleştirme (LEFT JOIN) o tablonun her satırını tutar. Eşi bulunan müşteri siparişleriyle gelir, eşi bulunmayanın sipariş sütunları NULL'la dolar. Sonuç 11 satır: 9 eşleşme, artı Selin ile Ozan (11 satırın ilk 4'ü):

ad	id	tarih
Selin Öztürk	NULL	NULL
Ozan Koç	NULL	NULL
Ece Yıldız	1001	2026-01-12
Mert Aydın	1002	2026-02-03

Bu boşluklar işe de yarar. Sorguya yalnız sipariş numarası NULL olan satırları bırakan bir koşul eklenince geriye hiç sipariş vermemiş müşteriler kalır:

```
SELECT m.ad
FROM musteri AS m
LEFT JOIN siparis AS s
    ON s.musteri_id = m.id
WHERE s.id IS NULL;
```

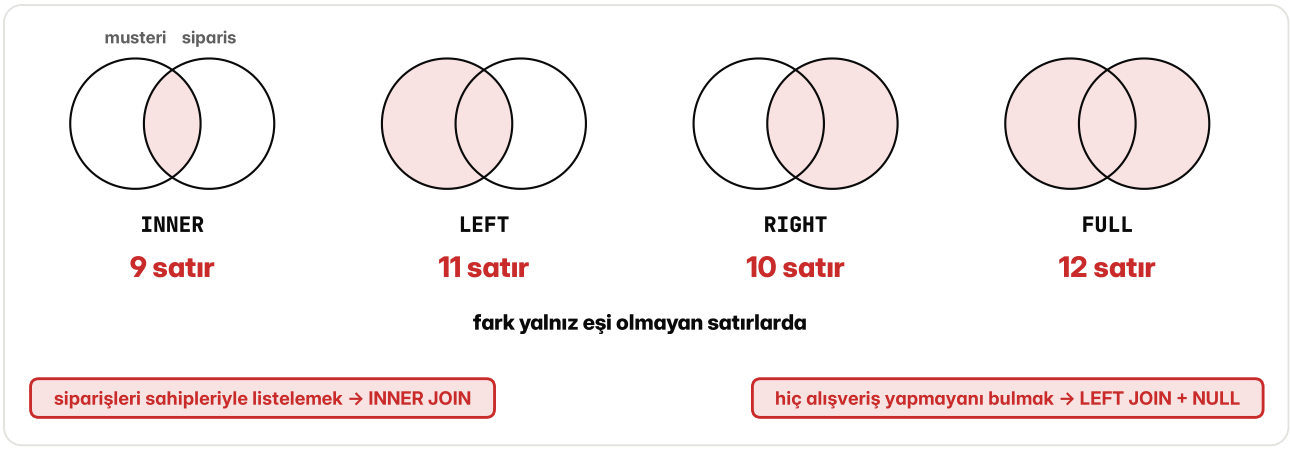
Sonuç 2 satır: Selin Öztürk ve Ozan Koç.



Şekil 3.2 Sol dış birleştirmeye WHERE s.id IS NULL eklenince hiç sipariş vermeyen iki müşteri kalır.

Sağ dış birleştirme (RIGHT JOIN) bunun aynasıdır: sağdaki tablonun, yani sipariş tablosunun her satırını tutar. `musteri AS m RIGHT JOIN siparis AS s` 10 satır verir; misafir alışverişi de gelir, müşteri adı NULL. Aynı sonuç, tablolar yer değiştirip sol birleştirmeyle de alınır (`siparis LEFT JOIN musteri` da 10 satır); bu yüzden çoğu yazılımcı yalnız sol birleştirme yazar. Tam dış birleştirme (FULL OUTER JOIN) ise iki tarafı birden tutar: 12 satır. 9 eşleşme, eşi olmayan 2 müşteri ve eşi olmayan 1 sipariş; sonucun son satırı `NULL | 1010`'dur. Her veritabanında yoktur: MySQL'de tam dış birleştirme bulunmaz, SQLite'a ise RIGHT ve FULL birleştirme 2022'de, 3.39 sürümüyle eklendi.

10 sipariş ve 8 müşteriden 4 ayrı cevap çıktı: iç birleştirmede 9, solda 11, sağda 10, tamda 12 satır. Fark yalnız eşi olmayan satırlarda. Hangi sayının doğru olduğunu soru belirler: siparişleri sahipleriyle listelemek için iç birleştirme, kimin hiç alışveriş yapmadığını bulmak için dış birleştirme.



Şekil 3.3 Dört birleştirme iki daireyle: INNER yalnız kesişim 9, LEFT sol daire 11, RIGHT sağ daire 10, FULL ikisi 12 satır.

Yirmi kalem

videoda 24:18

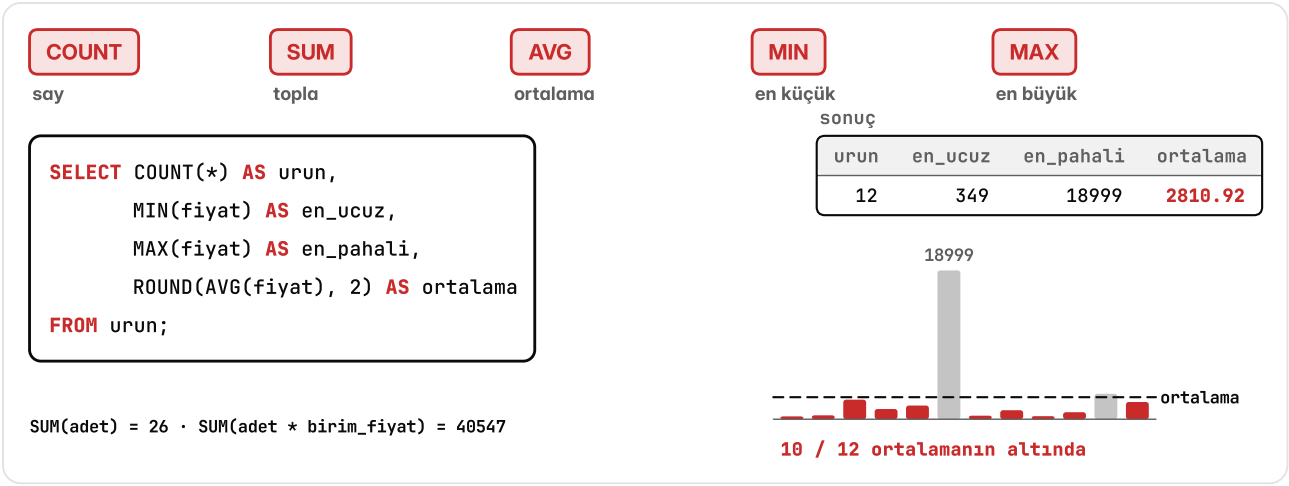
Bir mağaza en çok satan 5 ürününü nasıl bulur? Satışlar sipariş kalemi tablosunda durur: her satırda bir sipariş, o siparişteki bir ürün, kaç adet alındığı ve satış fiyatı yazar. Bu mağazada 20 kalem var. Cevap için iki şey gerekiyor: birçok satırı tek sayıya indirmek ve satırları gruplara ayırmak.

Birçok satırı tek sayıya indiren 5 **toplama fonksiyonu** (aggregate function) var: COUNT satırları sayar, SUM toplar, AVG ortalamayı, MIN en küçüğü, MAX en büyüğü verir. Ürün tablosuna soralım:

```
SELECT COUNT(*) AS urun,
       MIN(fiyat) AS en_ucuz,
       MAX(fiyat) AS en_pahali,
       ROUND(AVG(fiyat), 2) AS ortalama
FROM urun;
```

urun	en_ucuz	en_pahali	ortalama
12	349	18999	2810.92

12 ürünün en ucuzu 349, en pahalısı 18.999 lira, ortalaması 2.810 lira kadar (ROUND ortalamayı virgülden sonra 2 basamağa yuvarlar). Ortalamaya dikkat: 12 üründen 10'u bu sayının altında kalıyor, çünkü tek bir telefon ortalamayı yukarı çekiyor. Kalemlerde aynı yolla bütün satış bulunur: `SUM(adet)` 26 adet, `SUM(adet * birim_fiyat)` 40.547 lira ciro.



Şekil 3.4 Beş toplama fonksiyonu, ürün tablosunun tek satırlık sonucu ve fiyat çubukları; on iki fiyattan onu ortalama çizgisinin altında.

COUNT'un iki yazılışı farklı sayar. Parantezde yıldız varsa satırların hepsi sayılır; bir sütun adı varsa yalnız o sütunu dolu olan satırlar. Müşteri tablosunda `COUNT(*)` 8, `COUNT(sehir)` 7 verir, çünkü Deniz Arslan'ın şehri boş, NULL. Öteki toplama fonksiyonları da NULL'ları atlar; örneğin AVG, boş bir değeri sıfır sayıp ortalamayı düşürmez.

Bütün satışın toplamı yetmez, her ürünün kendi toplamı gerekir. Bunu GROUP BY yapar:

```
SELECT urun_id, SUM(adet) AS satilan
FROM siparis_kalemi
GROUP BY urun_id;
```

GROUP BY aynı ürün numarasını taşıyan satırları bir gruba toplar, SUM da her grubu ayrı toplar. Kablolu kulaklık (101) üç ayrı siparişte geçiyor: 1005'te 3, 1008'de 1 ve 1010'da 1 adet. Grubunda bunlar tek satıra iner, toplam 5. 20 kalemden 12 grup çıkar. Bir kural var: gruplu sorguda seçilen her sütun ya gruplamada yazılı olmalı ya da bir toplama fonksiyonunun içinde; yoksa grubun hangi satırından alınacağı belli olmaz. SQLite buna göz yumar, PostgreSQL hata verir.

Grupları da süzmek gerekebilir: en az 2 kez sipariş vermiş müşteriler kimler? İlk akla gelen, WHERE kısmına "sipariş sayısı en az 2" yazmak; veritabanı bunu hatayla reddeder:

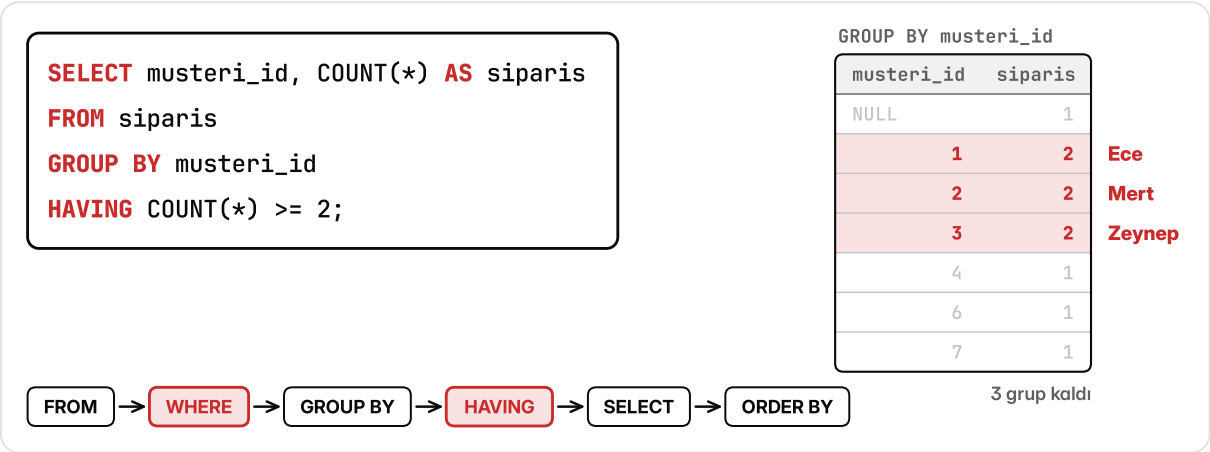
```
SELECT musteri_id, COUNT(*) AS siparis
FROM siparis
WHERE COUNT(*) >= 2
GROUP BY musteri_id;
HATA: OperationalError: misuse of aggregate: COUNT()
```

Çünkü WHERE satırları tek tek süzer; o sırada henüz grup yoktur, sayılacak bir şey yoktur. Grupları süzen HAVING'dir ve gruplar kurulduktan sonra çalışır:

```
SELECT musteri_id, COUNT(*) AS siparis
FROM siparis
GROUP BY musteri_id
HAVING COUNT(*) >= 2;
```

musteri_id	siparis
1	2
2	2
3	2

Sipariş tablosu müşteri numarasına göre 7 gruba ayrılır (biri misafir alışverişinin NULL grubu); HAVING 2 ve üstünü bırakır: Ece, Mert ve Zeynep. İkisi bir sorguda birlikte de yazılır ve sıra hep aynıdır: FROM, WHERE, GROUP BY, HAVING, SELECT, ORDER BY. Önce WHERE satırları süzer, sonra gruplar kurulur, en son HAVING grupları süzer.



Şekil 3.5 HAVING'li sorgu, yedi gruptan üçünün kalması ve FROM, WHERE, GROUP BY, HAVING, SELECT, ORDER BY sırası.

Şimdi açılıştaki soru. Kalemler ürün tablosuyla birleştirilir ki numaranın yerine ürünün adı görünsün; ürüne göre gruplanır, adetler toplanır. ORDER BY toplamı büyükten küçüğe sıralar, LIMIT ilk 5'i bırakır:

```
SELECT u.ad, SUM(k.adet) AS satilan
FROM siparis_kalemi AS k
JOIN urun AS u ON u.id = k.urun_id
GROUP BY u.id, u.ad
ORDER BY satilan DESC, u.ad
LIMIT 5;
```

ad	satilan
Kablolu kulaklık	5
Hızlı şarj aleti	4
Kablosuz kulak içi	4
Kablosuz fare	3
Pamuklu gömlek	3

"En çok satan" ciroyla (`SUM(k.adet * k.birim_fiyat)`) ölçülseydi liste değişirdi: başta bir kez satılan 18.999 liralık akıllı telefon olurdu, ardından kışlık mont (3.250), pamuklu gömlek (2.697), gürültü önleyici kulaklık (2.499) ve koşu ayakkabısı (2.199). Kablolu kulaklığın cirosu 1.745 lirayla ilk 5'e bile giremezdi. Sorgu, mağazanın hangi soruyu sorduğuna göre yazılır.



Şekil 3.6 En çok satan beş ürün adede ve ciroya göre yan yana: adette kablolu kulaklık, ciroda akıllı telefon başta.

Tek tablo, iki bakış

videoda 26:57

Bir mağaza, müşterisinin geri gelip gelmediğini nasıl anlar? Sipariş tablosunda her satır bir sipariş: numarası, tarihi ve siparişi veren müşterinin numarası. Bu mağazada 10 sipariş var. Cevap bu tek tablodan çıkar; yalnız tabloya iki ayrı gözle bakmak gerekir.

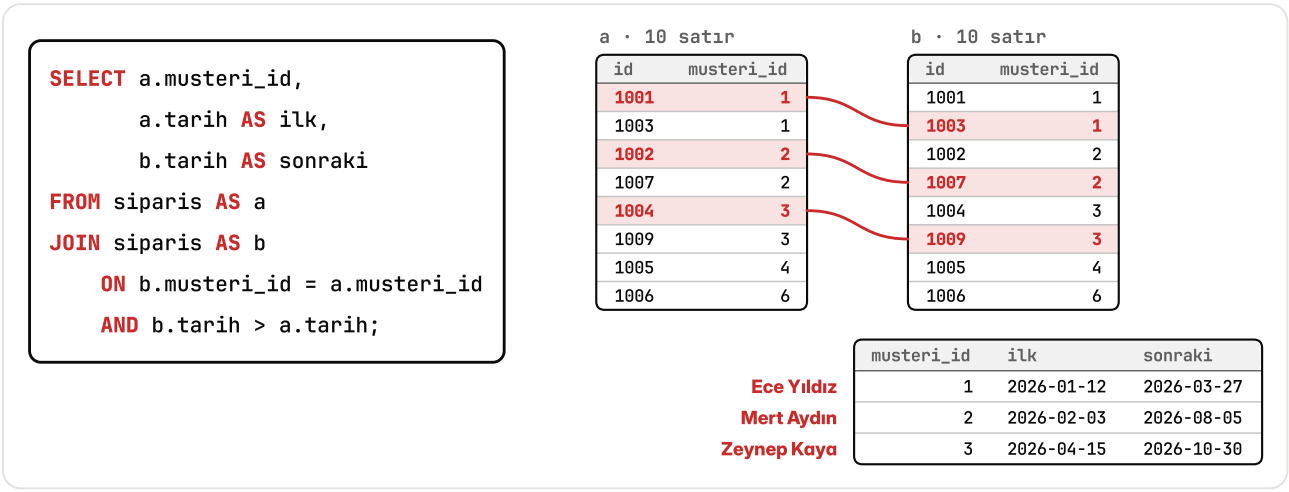
Bir müşterinin iki siparişini yan yana görmek için sipariş tablosu **kendisiyle birleştirilir** (self join). Aynı tablo sorguda iki kez yazılır, birine `a`, ötekine `b` adı verilir; veritabanı onları iki ayrı tablo gibi görür:

```
SELECT a.musteri_id,
       a.tarih AS ilk,
       b.tarih AS sonraki
FROM siparis AS a
JOIN siparis AS b
    ON b.musteri_id = a.musteri_id
    AND b.tarih > a.tarih;
```

Eşleşme koşulunun iki parçası var: iki siparişin müşteri numarası aynı olacak, `b`'deki sipariş de `a`'dakinden daha geç tarihli olacak. Sonuç 3 satır:

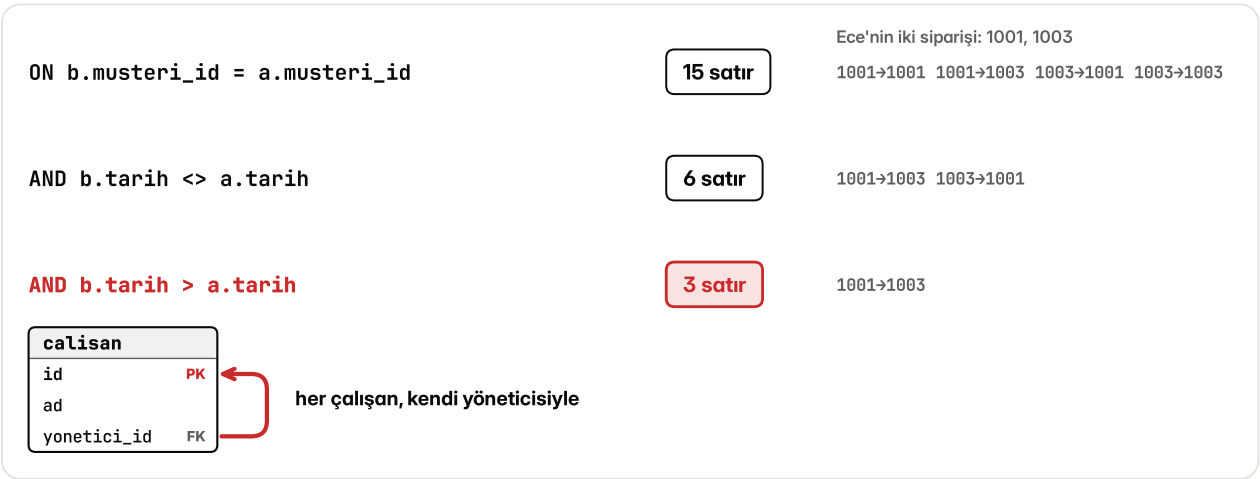
musteri_id	ilk	sonraki
1	2026-01-12	2026-03-27
2	2026-02-03	2026-08-05
3	2026-04-15	2026-10-30

Ece Yıldız (1) ocakta alışveriş yapıp martta geri geldi, Mert Aydın (2) şubatta gelip ağustosta, Zeynep Kaya (3) nisanda gelip ekimde. Üçü de geri gelen müşteri.



Şekil 3.7 Sipariş tablosunun a ve b kopyaları; aynı müşterinin iki siparişi çizgiyle bağlanır, sonuç üç satır.

Tarih koşulu neden gerekli? O olmasa her sipariş kendisiyle de eşleşirdi; 2 siparişi olan bir müşteri 4 satır verirdi ve sonuç 15 satır olurdu (iki siparişi olan 3 müşteriden 4'er, tek siparişi olan 3 müşteriden 1'er satır; misafir siparişinin NULL'u kendisiyle bile eşleşmez). "Tarihler farklı olsun" (`b.tarih <> a.tarih`) demek de yetmez: Ece'nin 2 siparişi bir kez bu sırayla, bir kez ters sırayla gelir, 6 satır çıkar. "Daha geç" koşulu her çifti bir kez bırakır. Aynı yol, bir tablodaki iki satırı karşılaştırmak gereken her yerde kullanılır; örneğin çalışanlar tablosunda her çalışanı kendi yöneticisiyle eşleştirmek.



Şekil 3.8 Üç koşul ve satır sayıları: yalnız müşteri eşitliği 15, eşit olmayan tarih 6, daha geç tarih 3 satır.

Şimdi soruyu yıla yayalım: yılın ilk yarısında alışveriş yapanlardan kim ikinci yarıda da geldi, kim gelmedi? Bu kez satırları yan yana değil, iki sorgunun sonuçlarını alt alta koyuyoruz. İlk sorgu ocaktan hazirana kadarki siparişlerin müşteri numaralarını verir: 6 satır (1, 2, 1, 3, 4, 6), Ece 2 kez. İkinci sorgu temmuzdan sonrakileri verir: 4 satır (2, 7, 3, NULL), biri misafir alışverişi. İki sorgunun arasına bir **küme işlemi** yazılır:

```
SELECT musteri_id FROM siparis
WHERE tarih < '2026-07-01'
EXCEPT
SELECT musteri_id FROM siparis
WHERE tarih >= '2026-07-01';
```

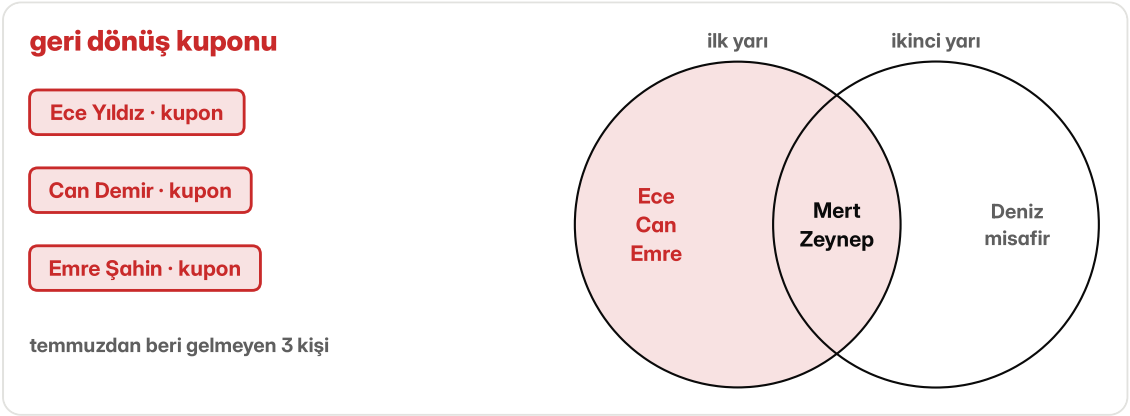
Tek şartı var: iki tarafın sütun sayısı ve türleri uyumlu olmalı; tutmazsa veritabanı hata verir. Solda 2, sağda 1 sütun olunca SQLite şöyle der:

```
SELECT musteri_id, tarih FROM siparis
UNION
SELECT id FROM musteri;
HATA: OperationalError: SELECTs to the left and right of UNION do not have the same
number of result columns
```

Arada hangi işlemin yazıldığına göre sonuç değişir:

- UNION ALL iki listeyi olduğu gibi alt alta koyar: 10 satır.
- UNION aynı değeri bir kez bırakır: 7 satır (NULL, 1, 2, 3, 4, 6, 7), 6 müşteri ve 1 misafir. Tekrarları ayıklamak ek iş olduğu için, tekrar olmayacağı bilinen yerde UNION ALL daha hızlıdır.
- INTERSECT iki listede de olanları verir: 2 ve 3 numara, yani Mert ile Zeynep.
- EXCEPT ilk listede olup ikincide olmayanları verir: 1, 4 ve 6 numara, yani Ece, Can ve Emre.

EXCEPT'te sıra önemlidir: iki yan yer değiştirince cevap da değişir, bu kez ikinci yarıda ilk kez gelenler çıkar: NULL ve 7, yani misafir alışverişi ve Deniz. Açılıştaki sorunun cevabı da böylece tamam: Mert ile Zeynep yılın iki yarısında da geldi; Ece, Can ve Emre temmuzdan beri hiç gelmedi. Mağaza geri dönüş kuponunu bu 3 kişiye gönderir.



Şekil 3.9 İlk yarı ve ikinci yarı daireleri: kesişimde Mert ve Zeynep, yalnız ilk yarıda Ece, Can ve Emre; kupon bu üç kişiye.

Özet

- JOIN iki tablonun satırlarını ON koşuluyla eşleştirir; iç birleştirmede (INNER JOIN) yalnız iki tarafta da eşi olan satırlar kalır.
- LEFT, RIGHT ve FULL OUTER JOIN eşi olmayan satırları da tutar ve boşlukları NULL'la doldurur; LEFT JOIN ile IS NULL, eşi olmayanları bulur.
- COUNT, SUM, AVG, MIN ve MAX birçok satırı tek sayıya indirir ve NULL'ları atlar; COUNT(*) bütün satırları, COUNT(sütun) dolu olanları sayar.
- GROUP BY satırları gruplara ayırır; WHERE satırları gruplamadan önce, HAVING grupları gruplamadan sonra süzer.
- Bir tablo iki ayrı adla kendisiyle birleştirilebilir; koşul her çifti bir kez bırakacak biçimde yazılır.
- UNION ALL, UNION, INTERSECT ve EXCEPT iki uyumlu sorgunun sonuçlarını alt alta birleştirir, ortak olanı ya da farkı verir; EXCEPT'te sıra önemlidir.

Sorular

1 **Sipariş tablosunda 10 sipariş var; biri misafir alışverişi, müşteri numarası NULL. SELECT COUNT(*), COUNT(musteri_id) FROM siparis; hangi iki sayıyı verir?**

A 10 ve 9

B 10 ve 10

C 9 ve 9

D 9 ve 10

2 **İzmirli iki müşteri var: Zeynep Kaya'nın 2 siparişi var, Ozan Koç hiç sipariş vermedi. SELECT COUNT(*) FROM siparis AS s JOIN musteri AS m ON m.id = s.musteri_id WHERE m.sehir = 'İzmir'; kaç verir? Sayıyı yaz.**

Cevap: _____

3 **musteri AS m LEFT JOIN siparis AS s ON s.musteri_id = m.id birleştirmesi 11 satır veriyor; hiç sipariş vermemiş Ozan Koç ve Selin Öztürk NULL'la geliyor. Ozan bir sipariş verirse aynı birleştirme kaç satır verir?**

A 12

B 10

C 13

D 11

4 En az 2 kalemde geçen ürünleri bulmak için hangi sorgu doğru çalışır?

A SELECT urun_id, COUNT(*) FROM siparis_kalemi WHERE COUNT(*) >= 2 GROUP BY urun_id;

B SELECT urun_id, COUNT(*) FROM siparis_kalemi GROUP BY urun_id HAVING COUNT(*) >= 2;

C SELECT urun_id, COUNT(*) FROM siparis_kalemi HAVING urun_id >= 2;

D SELECT urun_id, COUNT(*) FROM siparis_kalemi WHERE urun_id >= 2;

5 SELECT siparis_id, SUM(adet * birim_fiyat) AS tutar FROM siparis_kalemi GROUP BY siparis_id HAVING SUM(adet * birim_fiyat) > 3000; mağaza veritabanında kaç satır döndürür? Tutarlar: 1001: 1397, 1002: 21498, 1003: 1539, 1004: 3997, 1005: 1496, 1006: 2297, 1007: 4149, 1008: 2099, 1009: 1277, 1010: 798. Sayıyı yaz.

Cevap: _____

6 1001 numaralı siparişte 102 ve 107, 1006 numaralı siparişte 104 ve 102 numaralı ürünler var. İki siparişin urun_id listelerini UNION ile birleştiren sorgu kaç satır döndürür?

A 4 **B** 2

C 3 **D** 1

7 Aynı iki siparişte (1001: 102 ve 107; 1006: 104 ve 102) birinci sorgu 1001'in, ikinci sorgu 1006'nın urun_id'lerini veriyor. Aralarına EXCEPT yazılırsa sonuç ne olur?

A 104 **B** 102

C 102 ve 107 **D** 107

8 Sipariş tablosu kendisiyle birleştiriliyor: ON b.musteri_id = a.musteri_id AND b.tarih <> a.tarih. İki siparişi olan üç müşteri var (Ece, Mert, Zeynep), öteki müşterilerin birer siparişi var. Sorgu kaç satır döndürür? Sayıyı yaz.

Cevap: _____

VERİTABANI SİSTEMLERİ

4 SQL ileri: alt sorgu ve pencere fonksiyonu

Bu bölümde sorgunun içine sorgu koyuyoruz. Önce hiç sipariş vermemiş müşterileri ararken boş dönen bir listenin sebebini, NOT IN ile NULL'un tuzağını ve NOT EXISTS'i göreceğiz. Sonra iç içe kategorileri özyinelemeli WITH ile dolaşıp sorguya görünüm olarak kalıcı bir ad vereceğiz. En sonda pencere fonksiyonlarıyla satırları kaybetmeden sıra ve yürüyen toplam yazacağız. Örneklerin hepsi mağaza veritabanında, SQLite'ta çalıştırıldı.

Boş liste

videoda 29:22

Mağaza veritabanında müşteri tablosu 8 müşteriyi, sipariş tablosu her siparişi ve onu veren müşterinin numarasını tutar. Selin Öztürk ile Ozan Koç hiç sipariş vermedi. Onları bulmak için "numarası sipariş tablosundaki müşteri numaraları arasında olmayanları getir" diye yazalım:

```
SELECT ad
FROM musteri
WHERE id NOT IN (
    SELECT musteri_id
    FROM siparis
);
```

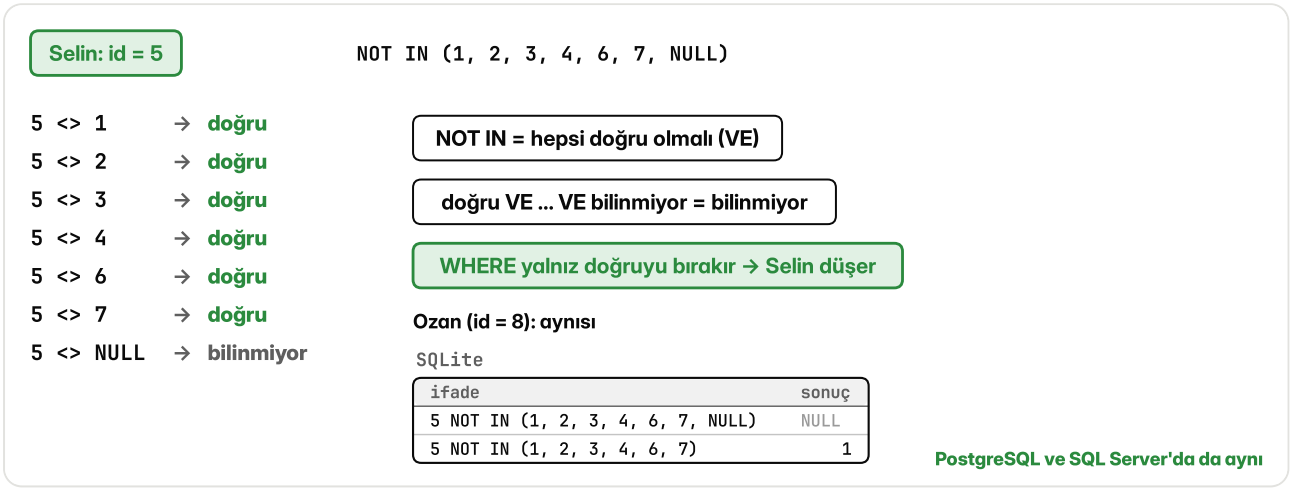
Sorgu hatasız çalışır ama sonuç 0 satırdır. Nedenini görmek için önce parantezin içine bakalım. Parantezdeki sorguya **alt sorgu** (subquery) denir: veritabanı önce onu çalıştırır, sonucunu dıştaki sorguda bir değer gibi kullanır. En basit hâli tek bir değer döndürür. Ortalamadan pahalı ürünleri isteyelim: içteki sorgu 12 ürünün ortalama fiyatını bulur (2810,92), dıştaki WHERE fiyatı bundan büyük olanları bırakır.

```
SELECT ad, fiyat
FROM urun
WHERE fiyat > (
    SELECT AVG(fiyat)
    FROM urun
);
```

ad	fiyat
Akıllı telefon	18999
Kışlık mont	3250

Alt sorgu bir liste de döndürebilir. Sipariş tablosundaki müşteri numaraları 1, 2, 3, 4, 6 ve 7'dir; listenin sonunda bir de NULL vardır, çünkü 1010 numaralı sipariş üye olmadan alışveriş yapan bir misafirindir. IN, değerin bu listede olup olmadığını sorar; IN ile yazılan sorgu sipariş vermiş 6 müşteriyi sorunsuz getirir.

IN yerine NOT IN yazınca iş değişir. Selin'in numarası 5'tir ve listedeki her değerle tek tek karşılaştırılır. 5, 1'den farklı mı? Doğru. 7'den farklı mı? Doğru. NULL'dan farklı mı? NULL bilinmeyen değerdir; bilinmeyenle karşılaştırmanın sonucu da bilinmiyordur. NOT IN bütün karşılaştırmaların doğru çıkmasını ister (hepsi VE ile bağlanır): 6'sı doğru, 1'i bilinmiyor, sonuç bilinmiyor. WHERE yalnız doğru olan satırı bıraktığı için Selin düşer; Ozan da öyle. SQLite'ta `5 NOT IN (1, 2, 3, 4, 6, 7, NULL)` NULL verir, NULL'suz listede 1. Listede tek bir NULL varsa NOT IN hiçbir satır döndürmez; PostgreSQL'de ve SQL Server'da da aynıdır.

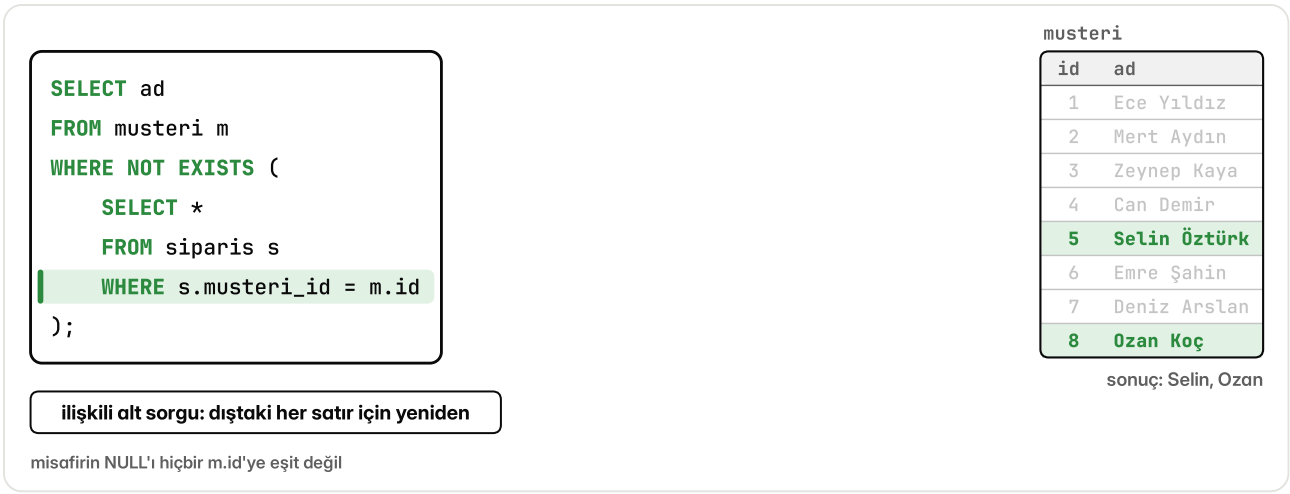


Şekil 4.1 5'in listedeki her değerle karşılaştırılması: altısı doğru, NULL'la karşılaştırma bilinmiyor.

Birinci çare alt sorguya `WHERE musteri_id IS NOT NULL` eklemektir. Daha sağlam yol NOT EXISTS'tir: alt sorgu değer listelemez, yalnız "bu müşterinin hiç siparişi var mı?" diye sorar.

```
SELECT ad
FROM musteri m
WHERE NOT EXISTS (
    SELECT *
    FROM siparis s
    WHERE s.musteri_id = m.id
);
```

İçteki sorgu dıştaki müşterinin numarasını (`m.id`) kullandığı için her müşteri için yeniden çalışır; buna **ilişkili alt sorgu** (correlated subquery) denir. Ece için 2 sipariş (1001, 1003) bulunur, Ece elenir; Selin için hiç satır bulunmaz, Selin kalır. Misafirin NULL numarası hiçbir müştereninkine eşit olmadığı için soruyu bozmaz. Sonuç: Selin Öztürk ve Ozan Koç. Mantık böyledir; veritabanı bunu çoğu zaman daha hızlı bir yola çevirir.

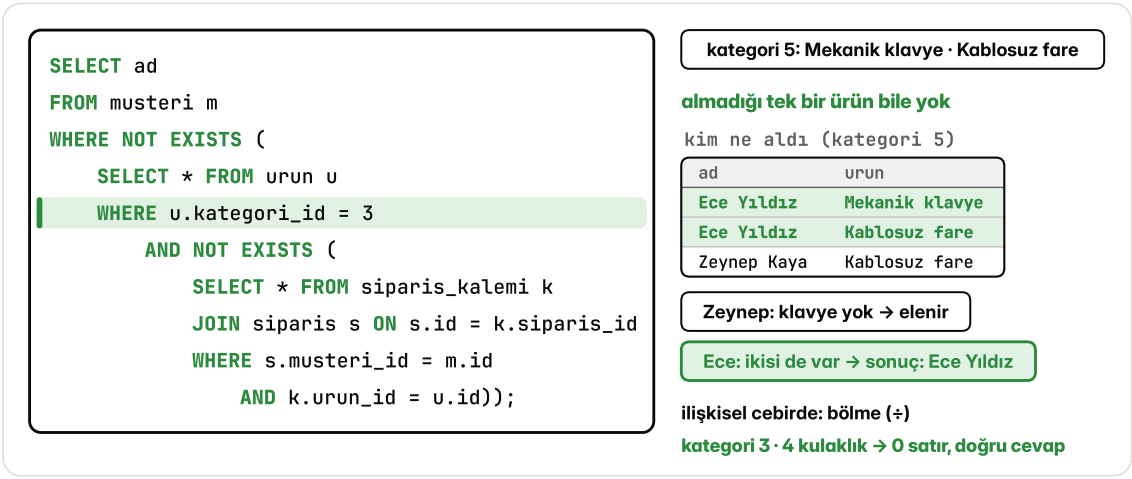


Şekil 4.2 NOT EXISTS: iç sorgu her müşteri için yeniden çalışır; yalnız Selin ve Ozan kalır.

NOT EXISTS'in bir ustalığı daha var. Bilgisayar aksesuarı kategorisinde (5) 2 ürün vardır: mekanik klavye ve kablosuz fare. İkisini de almış müşteri kim? "Bütün ürünleri aldı" demek "almadığı tek bir ürün bile yok" demektir; iç içe iki NOT EXISTS tam bunu söyler. Dıştaki, bu müşterinin almadığı bir aksesuar var mı diye sorar; içteki, bu müşteri bu ürünü aldı mı diye.

```
SELECT ad
FROM musteri m
WHERE NOT EXISTS (
    SELECT * FROM urun u
    WHERE u.kategori_id = 5
        AND NOT EXISTS (
            SELECT * FROM siparis_kalemi k
            JOIN siparis s ON s.id = k.siparis_id
            WHERE s.musteri_id = m.id
                AND k.urun_id = u.id));
```

Zeynep fareyi aldı, klavyeyi almadı; almadığı bir ürün olduğu için elenir. Ece ikisini de aldı; sonuç yalnız Ece Yıldız. İlişkisel cebirde bu işleme **bölme** (÷) denir; SQL'de ayrı bir komutu yoktur. Aynı sorgu 4 kulaklık için (kategori 3) 0 satır döner: dördünü birden alan müşteri yoktur ve bu kez boş liste doğru cevaptır.



Şekil 4.3 İç içe iki NOT EXISTS ile bölme: Zeynep elenir, sonuç Ece Yıldız.

Sıfır ürün

videoda 32:18

Mağazanın kategori tablosu iç içedir: her kategori, bağlı olduğu üst kategorinin numarasını (`ust_id`) tutar. Elektronik'in altında ses, telefon ve bilgisayar aksesuarı var; sesin altında da kulaklık. Her ürün en alttaki kategorisine bağlıdır. `WHERE kategori_id = 1` ile Elektronik'teki ürünleri isteyince sonuç 0 satırdır, çünkü hiçbir ürün doğrudan Elektronik'e bağlı değildir. Bütün elektronik ürünleri tek sorguyla nasıl isteriz?

İlk akla gelen bir alt sorgudur: içteki sorgu üst kategorisi Elektronik olan kategorileri bulur, dıştaki bu kategorilerdeki ürünleri getirir. 5 ürün gelir: hoparlör, telefon, şarj aleti, klavye ve fare. Kulaklıklar yoktur, çünkü 2 kat aşağıdadır. Bir alt sorgu daha eklesek 2 kat ineriz, ama ağacın kaç kat olduğunu bilmiyoruz; yarın kulaklığın altına bir kat daha açılabilir.

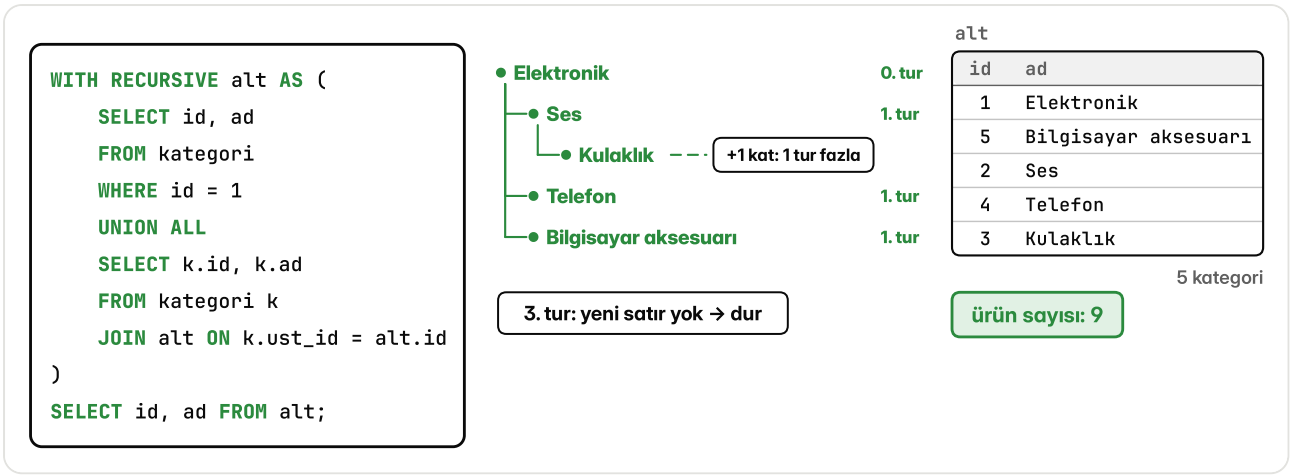
Önce bir araç: WITH, sorgunun en başında bir alt sorguya ad verir; adı verilen sorgu aşağıda bir tablo gibi kullanılır. Buna **ortak tablo ifadesi** (common table expression, CTE) denir. Sonuç aynı 5 ürün, sorgu daha okunur. Asıl güç şurada: adı olan bir sorgu kendi adını kullanabilir.

```
WITH RECURSIVE alt AS (
    SELECT id, ad
    FROM kategori
    WHERE id = 1
    UNION ALL
    SELECT k.id, k.ad
    FROM kategori k
    JOIN alt ON k.ust_id = alt.id
)
SELECT id, ad FROM alt;
```

Özyinelemeli WITH iki parçadan oluşur, arada ikisini alt alta ekleyen UNION ALL durur. İlk parça başlangıçtır, yalnız Elektronik'i getirir. İkinci parça kategori tablosunu o ana kadar bulunanlarla JOIN ile eşleştirir ve onların çocuklarını getirir. Veritabanı ikinci parçayı tur tur çalıştırır: 1. turda ses, telefon ve bilgisayar aksesuarı gelir, 2. turda kulaklık; 3. turda yeni satır çıkmaz ve iş biter.

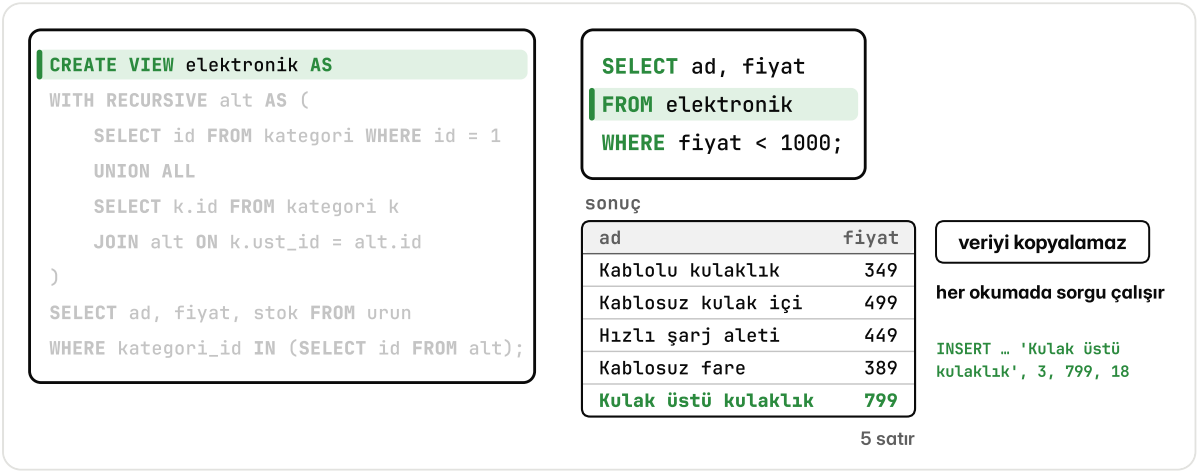
id	ad
1	Elektronik
5	Bilgisayar aksesuarı
2	Ses
4	Telefon
3	Kulaklık

5 kategori bulundu; bunlardaki ürünler sayılınca 9 çıkar. Ağaç bir kat daha derinleşse sorguya dokunmak gerekmez, yalnız bir tur fazla döner.



Şekil 4.4 Özyinelemeli WITH kategori ağacını tur tur dolaşır; 5 kategori, 9 ürün.

Bu sorguyu her seferinde yazmak zahmetli. **Görünüm** (view), bir sorguya kalıcı bir ad verir: `CREATE VIEW elektronik AS` ile yukarıdaki sorgu `ad, fiyat, stok` sütunlarıyla kaydedilir ve `elektronik` artık bir tablo gibi sorgulanır. Fiyatı 1000 liranın altındaki elektronik ürünleri isteyince 4 ürün gelir: kablolu kulaklık (349), kablosuz kulak içi (499), hızlı şarj aleti (449), kablosuz fare (389). Görünüm veriyi kopyalamaz, her kullanıldığında altındaki sorgu yeniden çalışır: ürün tablosuna 799 liralık yeni bir kulaklık eklenince aynı sorgu 5 ürün döndürür.



Şekil 4.5 elektronik görünümü: yeni kulaklık eklenince aynı sorgu 5 ürün döndürür.

Bazı görünümler milyonlarca satırı toplayan bir rapor gibi pahalı bir sorguya dayanır. Bunlar için **somutlaştırılmış görünüm** (materialized view) vardır: sonuç bir kez hesaplanıp diskte saklanır, okumak hızlıdır. Bedeli, veri değişince sonucun eskimesidir; elle ya da belirli aralıklarla yenilenir. PostgreSQL'de vardır, SQLite'ta yoktur. PostgreSQL 19'da aynı sorguyla kurulan `elektronik_ozet`, yeni kulaklık eklendikten sonra da 4 satır gösterdi; `REFRESH MATERIALIZED VIEW elektronik_ozet;` komutundan sonra 5 satır oldu.

Görünüm bir şeyi gizlemek için de kullanılır. Sitenin vitrini stokta tam kaç ürün kaldığını göstermesin, yalnız bir durum yazsın. Bunu CASE ifadesi yapar; SQL'in eğer-ise yapısıdır.

```
CREATE VIEW vitrin AS
SELECT ad, fiyat,
    CASE
        WHEN stok < 5 THEN 'tükeniyor'
        WHEN stok < 10 THEN 'az kaldı'
        ELSE 'stokta'
    END AS durum
FROM elektronik;
```

Koşullar yukarıdan aşağı denenir, ilk tutan kazanır: 4 tane kalan akıllı telefon ilk koşulda durur (tükeniyor), 6 tane kalan gürültü önleyici kulaklık ikincide (az kaldı). Vitrinde ad, fiyat ve durum vardır, stok sütunu hiç yoktur; `SELECT stok FROM vitrin;` hata verir:

```
no such column: stok
```



Şekil 4.6 vitrin görünümü stok sayısı yerine CASE ile yazılan durumu gösterir.

GROUP BY yetmez

videoda 35:11

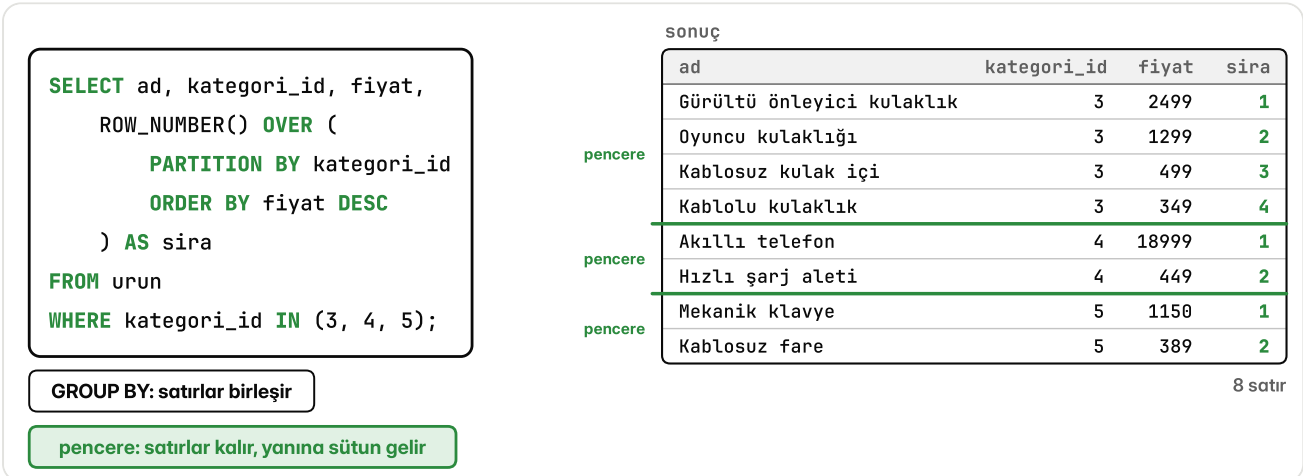
Bir mağaza her kategorinin en pahalı 3 ürününü vitrinin başına koymak istiyor. Tablo kısa kalsın diye 3 kategoriye bakalım: kulaklık (3), telefon (4), bilgisayar aksesuarı (5). GROUP BY ile kategori başına en yüksek fiyatı buluruz: 2499, 18999, 1150. Ama GROUP BY her kategoriyi tek satıra indirir; elimizde 3 sayı kalır, ürünlerin kendisi kaybolur.

Bunun aracı **pencere fonksiyonudur** (window function). Pencere, bir satırın hesap yaparken baktığı satırlardır ve OVER ile tarif edilir. PARTITION BY satırları kategorilerine göre ayrı pencerelere böler; içindeki ORDER BY her pencereyi pahalıdan ucuza dizer; ROW_NUMBER her penceredeki satırlara 1, 2, 3 diye sıra numarası verir.

```
SELECT ad, kategori_id, fiyat,
    ROW_NUMBER() OVER (
        PARTITION BY kategori_id
        ORDER BY fiyat DESC
    ) AS sira
FROM urun
WHERE kategori_id IN (3, 4, 5);
```

ad	kategori_id	fiyat	sira
Gürültü önleyici kulaklık	3	2499	1
Oyuncu kulaklığı	3	1299	2
Kablosuz kulak içi	3	499	3
Kablolu kulaklık	3	349	4
Akıllı telefon	4	18999	1
Hızlı şarj aleti	4	449	2
Mekanik klavye	5	1150	1
Kablosuz fare	5	389	2

GROUP BY'dan farkı: satırlar yerinde kalır, sonuç yanlarına yeni bir sütun olarak eklenir. 8 ürünün 8'i de sonuçtadır.



Şekil 4.7 ROW_NUMBER her kategori penceresinde fiyata göre sıra verir; satırlar yerinde kalır.

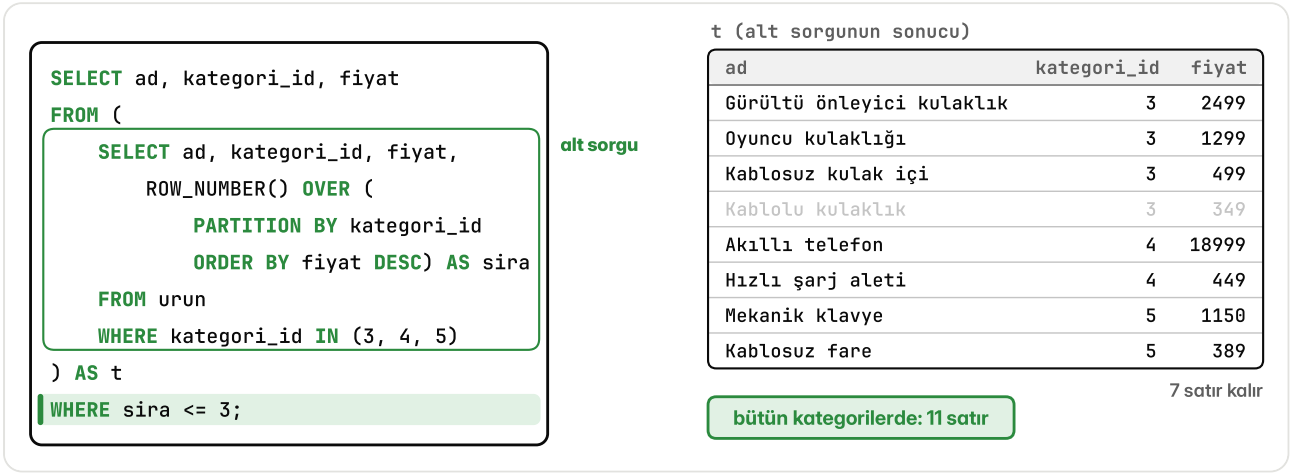
Sırası 3'ü geçmeyenleri istemek için sorgunun sonuna `WHERE sira <= 3` eklemek akla gelir, ama sorgu hata verir:

```
misuse of aliased window function sira
```

WHERE satırları pencere hesaplanmadan önce süzer; sıra sütunu o anda henüz yoktur. Çözüm, bütün sorguyu bir alt sorgu yapıp FROM'un içine koymaktır: içteki sorgunun sonucu bir tablo gibi kullanılır, dıştaki WHERE bu tabloyu süzer.

```
SELECT ad, kategori_id, fiyat
FROM (
    SELECT ad, kategori_id, fiyat,
        ROW_NUMBER() OVER (
            PARTITION BY kategori_id
            ORDER BY fiyat DESC) AS sira
    FROM urun
    WHERE kategori_id IN (3, 4, 5)
) AS t
WHERE sira <= 3;
```

7 ürün kalır; kulaklıklardan en ucuzu, 349 liralık kablolu kulaklık düşer. Bütün kategorilerde aynı sorgu 11 satır döndürür.



Şekil 4.8 Pencere sorgusu FROM'un içinde alt sorgu olur; dıştaki WHERE kablolu kulaklığı eler.

ROW_NUMBER'ın bir zayıflığı var. Kablosuz fare indirime girsin: 389 liradan 349 liraya; artık kablolu kulaklıkla aynı fiyatta. 500 liranın altındaki ürünleri ucuzdan pahalıya sıralayınca ROW_NUMBER birini birinci, öbürünü ikinci yapar; hangisinin önde olacağını kendi seçer. RANK ise eşitlere aynı dereceyi verir:

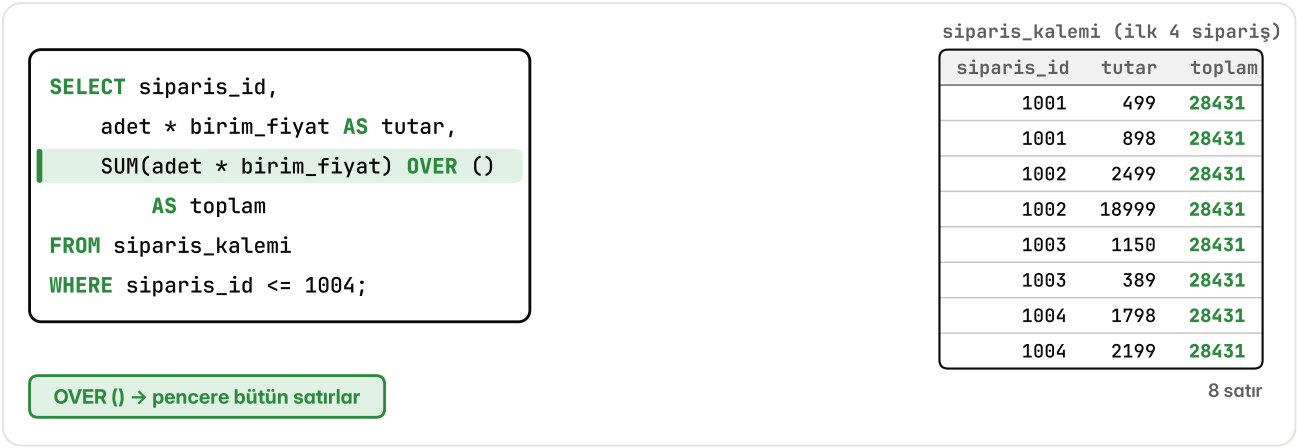
ad	fiyat	sira	derece
Kablolu kulaklık	349	1	1
Kablosuz fare	349	2	1
Hızlı şarj aleti	449	3	3
Kablosuz kulak içi	499	4	4

İkisi de birinci olur, sonraki ürün üçüncü; ikinci derece atlanır, çünkü önünde 2 ürün vardır.

Pencere yalnız sıra vermez; toplama fonksiyonları da pencerede çalışır. Sipariş kalemi tablosu her siparişte hangi üründen kaç tane, kaç liradan alındığını tutar. İlk 4 siparişin kalemlerine bakalım:

```
SELECT siparis_id,
    adet * birim_fiyat AS tutar,
    SUM(adet * birim_fiyat) OVER (
        ORDER BY siparis_id, urun_id
    ) AS kasa
FROM siparis_kalemi
WHERE siparis_id <= 1004;
```

SUM() OVER, pencereyi sipariş sırasına göre dizince her satıra o satıra kadarki toplamı yazar; buna **yürüyen toplam** (running total) denir. Birinci kalem 499 lira, kasa da 499; ikincide 898 eklenir, kasa 1.397; telefonla kasa 22.895'e sıçrar; dördüncü siparişin sonunda kasada 28.431 lira vardır. OVER'ın içi boş bırakılırsa (`OVER ()`) pencere bütün satırlardır ve her satırda aynı genel toplam, 28431 yazar.



Şekil 4.9 SUM() OVER ile yürüyen toplam: kasa kalem kalem 28.431 liraya çıkar.

Pencere fonksiyonları satırları gruplara indirmez; her satırın yanına sıra, derece ya da toplam yazar.

Özet

- Alt sorgu önce çalışır, sonucu dıştaki sorguda tek değer, liste ya da (FROM içinde) tablo olarak kullanılır.
- Alt sorgunun listesinde tek bir NULL varsa NOT IN hiçbir satır döndürmez; NOT EXISTS bu tuzağa düşmez.
- İlişkili alt sorgu dıştaki her satır için yeniden çalışır; iç içe iki NOT EXISTS bölme işlemini yazar.
- WITH alt sorguya ad verir; özyinelemeli WITH iç içe yapıyı tur tur dolaşır.
- Görünüm sorguya kalıcı bir ad verir ve veriyi kopyalamaz; somutlaştırılmış görünüm sonucu saklar, yenilenmesi gerekir.
- Pencere fonksiyonları (ROW_NUMBER, RANK, SUM() OVER) satırları kaybetmeden sıra, derece ve yürüyen toplam yazar.

Sorular

1 Haziran 2026'dan beri sipariş vermemiş müşterileri bulmak için şu sorgu yazılıyor: SELECT ad FROM musteri WHERE id NOT IN (SELECT musteri_id FROM siparis WHERE tarih >= '2026-06-01'); Bu tarihten sonraki siparişlerden biri misafir siparişi (musteri_id NULL). Sorgu kaç satır döndürür?

A 4
B 0
C 3
D 8

2 Aynı soruyu NOT EXISTS ile soruyoruz: SELECT ad FROM musteri m WHERE NOT EXISTS (SELECT * FROM siparis s WHERE s.musteri_id = m.id AND s.tarih >= '2026-06-01'); Sorgu kaç satır döndürür? Yalnız sayıyı yaz.

Cevap:

3 Kulaklıkların (kategori 3) fiyatları 349, 499, 1299 ve 2499 TL. SELECT COUNT(*) FROM urun WHERE fiyat > (SELECT AVG(fiyat) FROM urun WHERE kategori_id = 3); sorgusu kaç döndürür?

A 2
B 4
C 6
D 12

4 Üst giyim kategorisinde (7) pamuklu gömlek ve kışlık mont var. Mert Aydın ikisini de aldı, Zeynep Kaya yalnız gömleği aldı. Videodaki iç içe iki NOT EXISTS sorgusu kategori 7 için çalıştırılırsa sonuç ne olur?

A Zeynep Kaya
B Mert Aydın ve Zeynep Kaya
C Hiç satır gelmez
D Mert Aydın

5 Videodaki özyinelemeli WITH sorgusunda başlangıç Elektronik (id = 1) yerine Giyim (id = 6) yapılıyor ve bulunan kategorilerdeki ürünler sayılıyor. Sonuç kaçtır? Yalnız sayıyı yaz.

Cevap:

6 Vitrin görünümündeki CASE ifadesinin koşulları yer değiştiriyor: önce WHEN stok < 10 THEN 'az kaldı', sonra WHEN stok < 5 THEN 'tükeniyor', ELSE 'stokta'. Stokta 4 tane kalan akıllı telefonun durumu ne olur?

A az kaldı
B tükeniyor
C stokta
D NULL

7 Kulaklıkların stokları: kablolu kulaklık 40, kablosuz kulak içi 25, oyuncu kulaklığı 12, gürültü önleyici kulaklık 6. ROW_NUMBER() OVER (PARTITION BY kategori_id ORDER BY stok DESC) oyuncu kulaklığına hangi sırayı verir?

A 1
B 2
C 4
D 3

8 SUM(adet * birim_fiyat) OVER (ORDER BY siparis_id, urun_id) ile 1005, 1006 ve 1007 numaralı siparişlerin kalemlerinin yürüyen toplamı yazılıyor. Kalemlerin tutarları sırayla 1047, 449, 998, 1299, 899 ve 3250 TL. 1006 numaralı siparişin son kalemine gelindiğinde kasa kaç liradır? Yalnız sayıyı yaz.

Cevap: _____

VERİTABANI SİSTEMLERİ

5 Veritabanı programlama ve saklı yordamlar

Bu bölümde SQL'i bir programın içinden kullanıyoruz. Önce bir Python programının veritabanıyla nasıl konuştuğunu, değeri sorguya neden ayrı gönderdiğimizi ve satır satır döngünün neden pahalı olduğunu göreceğiz. Sonra kuralı programlardan alıp veritabanının içine yazacağız: kullanıcı tanımlı fonksiyon ve saklı yordam. Python örnekleri SQLite'ta, fonksiyon ve yordam PostgreSQL 19'da çalıştırıldı; örnekler mağaza veritabanında.

Düğmenin arkası

videoda 37:49

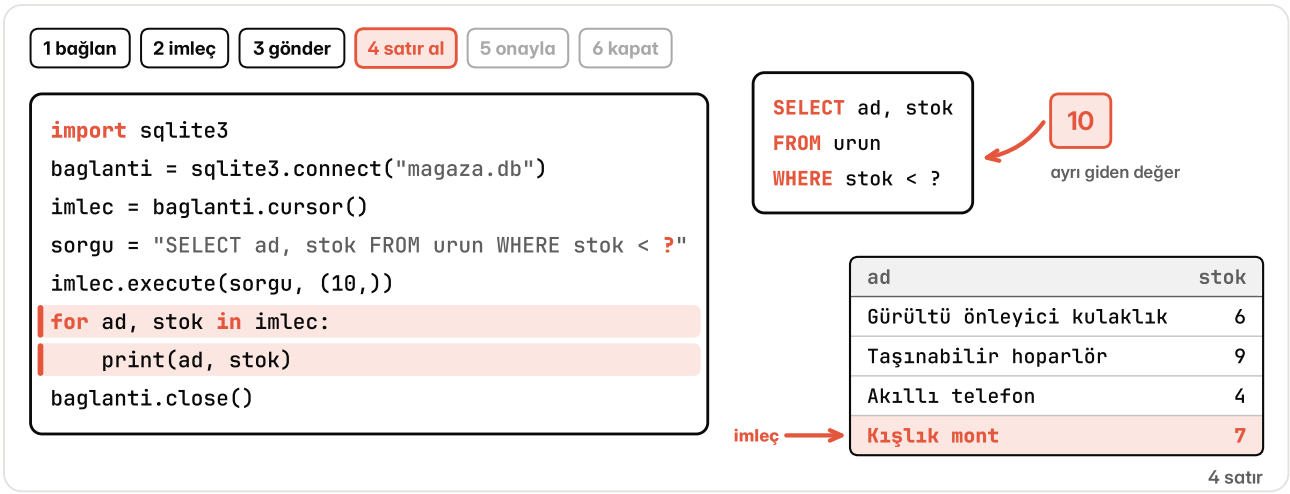
Mağazanın deposunda her sabah aynı soru sorulur: stoğu azalan ürünler hangileri? Cevap, 12 ürünün adını, fiyatını ve depoda kaç tane kaldığını tutan ürün tablosundadır. Depo görevlisi SQL yazmaz, bir düğmeye basar. Düğmenin arkasındaki program Python'da yazılmıştır, veritabanı ise yalnız SQL anlar. Program ile veritabanı arasındaki konuşma her dilde aynı altı adımla yürür: bağlan, imleç aç, sorguyu gönder, sonucu satır satır al, değişiklik varsa onayla, kapat. Java'da da adımlar aynıdır (JDBC: Connection, PreparedStatement, ResultSet), yalnız adları değişir.

Bizim veritabanımız SQLite. SQLite ayrı bir sunucu değil, programın içinde çalışan bir kütüphanedir; bütün veritabanı diskteki tek bir dosyadır. Bağlanmak bu dosyayı açmaktır. PostgreSQL gibi bir sunucuya bağlanırken dosya adı yerine adres, kullanıcı adı ve şifre verilir, gerisi aynıdır. Sonra **imleç** (cursor) açılır: sorguyu gönderen ve gelen sonucun üstünde satır satır ilerleyen işaretçi.

```
import sqlite3
baglanti = sqlite3.connect("magaza.db")
imlec = baglanti.cursor()
sorgu = "SELECT ad, stok FROM urun WHERE stok < ?"
imlec.execute(sorgu, (10,))
for ad, stok in imlec:
    print(ad, stok)
baglanti.close()
```

Sorgunun içinde 10 yazmaz, yerinde bir soru işareti vardır; 10 sorgunun yanında ayrı bir değer olarak gönderilir. Programdaki döngü imleçten her turda bir satır alır ve ekrana yazar. 4 satır gelir:

ad	stok
Gürültü önleyici kulaklık	6
Taşınabilir hoparlör	9
Akıllı telefon	4
Kışlık mont	7



Şekil 5.1 Sorgu metni ve ayrı giden 10 değeri; imleç sonuç tablosunun satırlarında birer birer ilerler.

Soru işaretli bu yazıma **parametreli sorgu** denir. Değeri sorgu metninin içine yapıştırırsak değer komutun bir parçası olur, veritabanı onu da komut gibi okur. Ayrı gönderilen değer ise ne olursa olsun yalnız değer kalır, komutu değiştiremez. Bir kazancı daha vardır: sorgu metni hiç değişmez. Eşik 5 olursa aynı metin 5'le gider ve tek satır gelir (akıllı telefon); 20 olursa 7 satır. Metin aynı kaldığı için veritabanı sorguyu her seferinde baştan çözümlemez, hazırladığını yeniden kullanabilir. Bu yüzden değer, sayı da olsa isim de olsa, soru işaretiyle gönderilir.

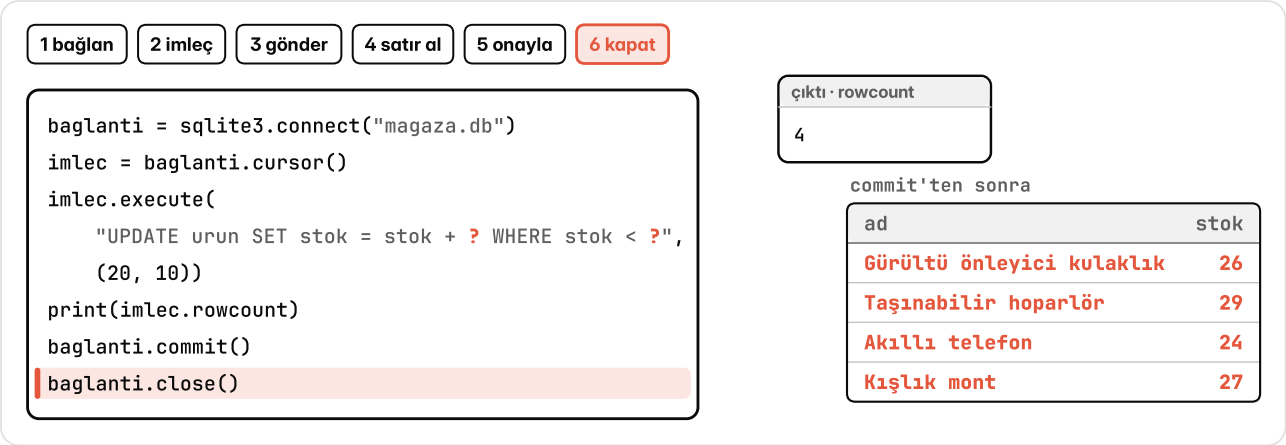


Şekil 5.2 Yapıştırılmış değer üstü çizili; sorgu metni aynı kalır, eşik 20 olunca 7 satır gelir.

Şimdi stoğu 10'dan az olan her ürüne 20 adet ekleyelim:

```
baglanti = sqlite3.connect("magaza.db")
imlec = baglanti.cursor()
imlec.execute(
    "UPDATE urun SET stok = stok + ? WHERE stok < ?",
    (20, 10))
print(imlec.rowcount)
baglanti.commit()
baglanti.close()
```

Veritabanı 4 satırın değiştiğini söyler (`rowcount` 4). `commit()` satırı olmadan bağlantı kapatılıp dosya yeniden açılırsa gürültü önleyici kulaklıkta hâlâ 6 tane görünür: değişiklik onaylanmadığı için kaybolmuştur. Bağlantının commit komutu değişikliği kalıcı yapar; commit'ten sonra stoklar 26, 29, 24 ve 27 olur. Son adımda bağlantı kapatılır.



Şekil 5.3 commit satırı eklenince yeniden açılan dosyada yeni stoklar: 26, 29, 24, 27.

Aynı iş döngüyle de yazılabilir: önce stoğu az olan ürünlerin numaralarını al (103, 105, 106, 111), sonra her ürün için ayrı bir UPDATE gönder. Sonuç aynıdır ama veritabanına 5 komut gider: 1 okuma, 4 güncelleme. Tek UPDATE ise WHERE koşuluna uyan bütün satırları bir kerede değiştirir; iş 1 komutla biter. SQL kümelerle çalışır, program satır satır düşünür. 4 üründe fark görünmez; 10.000 ürünlük bir tabloda döngü 10.001 komut demektir ve veritabanı başka bir makinedeyse her biri ağda gidip gelir. İş tek bir SQL komutuyla yapılabiliyorsa öyle yapılır. Programcılar sorguları çoğu zaman elle de yazmaz: tablodaki satırları programdaki nesnelere çeviren **nesne ilişkisel eşleme** (ORM) kütüphaneleri (Python'da SQLAlchemy ve Django, Java'da Hibernate) onların yerine yazar. Arkada yine aynı SQL komutları gider; döngü tuzağı orada da saklanabilir.

Üç program, bir kural

videoda 40:55

Mağazada aynı siparişin tutarını üç program hesaplar: web sitesi, telefon uygulaması ve fatura programı. Kural basittir: 1500 liranın altındaki siparişe 99 lira kargo eklenir. Kural üç programa ayrı ayrı yazılmışsa, eşik 2000'e çıktığı gün biri unutulur ve 1003 numaralı sipariş sitede 1638, faturada 1539 lira çıkar. Kural, üç programın da konuştuğu yere, veritabanına yazılmalıdır; PostgreSQL gibi sunucu veritabanları kod da saklar.

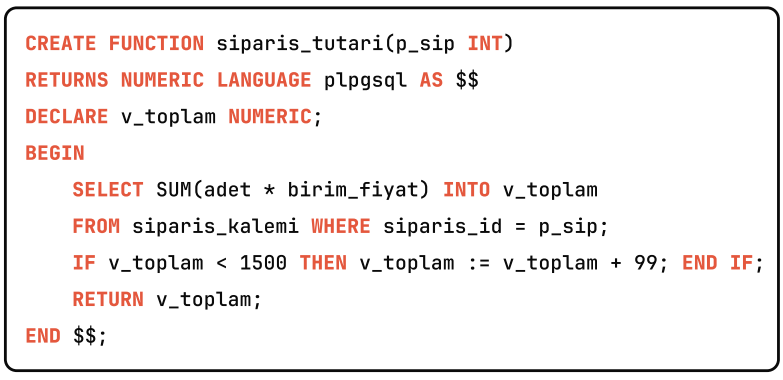
İlk araç **kullanıcı tanımlı fonksiyon**dur. Adı `siparis_tutari`, parametresi sipariş numarası, içinde bir değişken var: toplam. Fonksiyon, her siparişte hangi üründen kaç tane ve kaça satıldığını tutan sipariş kalemi tablosunda adetle birim fiyatı çarpıp toplar, değişkene yazar; toplam 1500'ün altındaysa 99 ekler ve sonucu döndürür. Kod PostgreSQL'in kendi dili PL/pgSQL ile yazılır:

```
CREATE FUNCTION siparis_tutari(p_sip INT)
RETURNS NUMERIC LANGUAGE plpgsql AS $$
DECLARE v_toplam NUMERIC;
BEGIN
    SELECT SUM(adet * birim_fiyat) INTO v_toplam
    FROM siparis_kalemi WHERE siparis_id = p_sip;
    IF v_toplam < 1500 THEN v_toplam := v_toplam + 99; END IF;
    RETURN v_toplam;
END $$;
```

Fonksiyon veritabanında saklanır ve sorgunun içinde çağrılır. 10 siparişin her biri için çağıralım (`SELECT id, siparis_tutari(id) FROM siparis`; yanında kalemlerin ham toplamı):

id	ham	tutar
1001	1397	1496
1002	21498	21498
1003	1539	1539
1004	3997	3997
1005	1496	1595
1006	2297	2297
1007	4149	4149
1008	2099	2099

(10 satırın ilk 8'i.) 1001 numaralı sipariş 1397 liraydı, kargoyla 1496 oldu; 21.000 lirayı aşan 1002 olduğu gibi kaldı. Üç program da artık hesap yapmaz, bu fonksiyonu çağırır.



SELECT id, siparis_tutari(id)

id	ham	tutar
1001	1397	1496
1002	21498	21498
1003	1539	1539
1004	3997	3997
1005	1496	1595
1006	2297	2297
1007	4149	4149
1008	2099	2099

10 satır

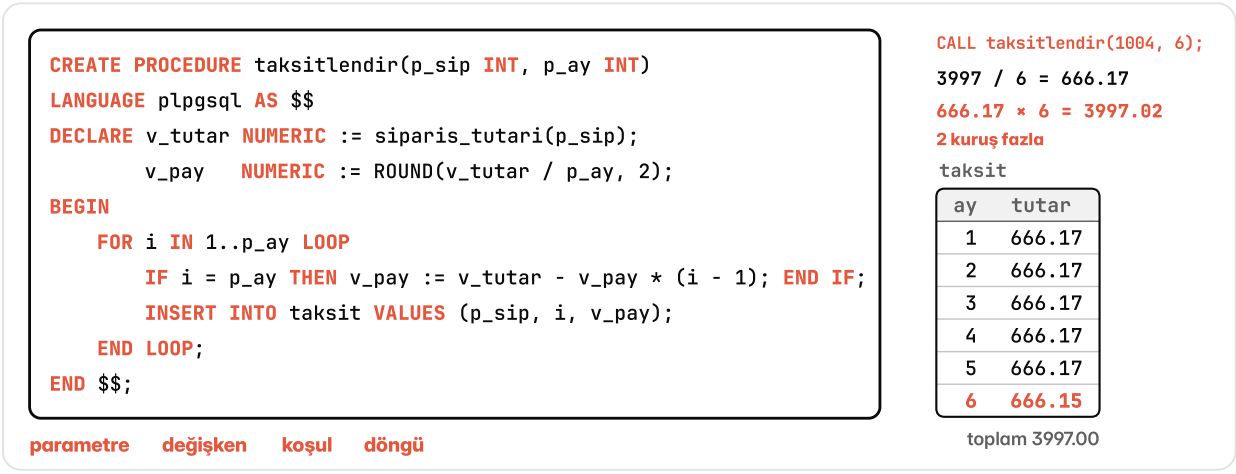
Şekil 5.4 siparis_tutari fonksiyonu ve sonucu: kargo eklenen 1001 ve 1005 vurgulu.

İkinci araç **saklı yordam**dır (stored procedure). Fonksiyon değer döndürür ve sorgunun içinde kullanılır; saklı yordam ise CALL ile çağrılan bir iştir, çoğu zaman veriyi değiştirir. `taksitlendir` yordamı iki parametre alır: sipariş numarası ve ay sayısı. Tutarı fonksiyondan alır, ay sayısına bölüp kuruşa yuvarlar, sonra bir döngüyle taksit tablosuna her ay için bir satır yazar:

```
CREATE PROCEDURE taksitlendir(p_sip INT, p_ay INT)
LANGUAGE plpgsql AS $$
DECLARE v_tutar NUMERIC := siparis_tutari(p_sip);
        v_pay   NUMERIC := ROUND(v_tutar / p_ay, 2);
BEGIN
    FOR i IN 1..p_ay LOOP
        IF i = p_ay THEN v_pay := v_tutar - v_pay * (i - 1); END IF;
        INSERT INTO taksit VALUES (p_sip, i, v_pay);
    END LOOP;
END $$;

CALL taksitlendir(1004, 6);
```

3997 lira 6 ayda 666,17 lira eder. Altı taksitin hepsi 666,17 olsaydı toplam 3997,02 çıkardı, 2 kuruş fazla. Bu yüzden döngüde bir koşul var: son ayda taksit, tutardan önceki 5 taksit çıkarılarak bulunur (3997 − 3330,85 = 666,15). Toplam kuruşu kuruşuna 3997,00 tutar. Parametre, değişken, koşul ve döngü: bir programlama dilinin temel parçaları buradadır.

ay	tutar
1	666.17
2	666.17
3	666.17
4	666.17
5	666.17
6	666.15



Şekil 5.5 taksitlendir yordamı ve taksit tablosu: son ay 666,15, toplam 3997,00.

Bunun iki kazancı var. Birincisi ağ trafiği: taksit planını program kendisi kursaydı 7 komut gönderirdi (tutarı okumak için 1, her taksit için 1); saklı yordamla tek çağrı gider, iş veritabanının içinde biter. İkincisi kural tek yerde durur: kargo eşiği 2000 liraya çıkınca fonksiyon `CREATE OR REPLACE FUNCTION` ile bir kez değişir ve 1539 liralık 1003 numaralı sipariş o andan itibaren her programda 1638 lira görünür.

Bedeli de var. Uygulamanın kodu bir sürüm takip sisteminde (Git gibi) durur, kim neyi ne zaman değiştirdi bellidir. Veritabanındaki fonksiyon ise veritabanının içinde yaşar; onu da kodla birlikte takip etmek ayrıca özen ister. Bir de dil sorunu var: bu kodlar PL/pgSQL ile yazıldı; SQL Server'ın dili T-SQL, Oracle'ınki PL/SQL. Veritabanı değişirse bu kodlar yeniden yazılır.

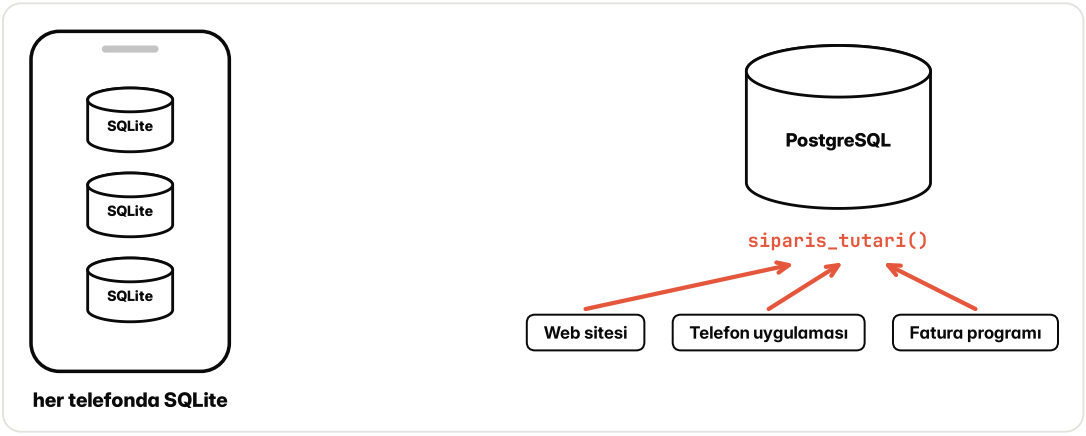
Aynı fonksiyonu SQLite'ta yazmaya kalkınca sözdizimi hatası alınır:

```
CREATE PROCEDURE taksitlendir(p_sip INT, p_ay INT);
HATA: OperationalError: near "PROCEDURE": syntax error
```

SQLite'ta saklı yordam yoktur, çünkü SQLite ayrı bir sunucu değil, programın içinde çalışan bir kütüphanedir; arada ağda gidip gelen bir şey yoktur. Kural gerekiyorsa programın kendi dilinde yazılıp bağlantıya tanıtılır ve sorguda aynı adla çağrılır:

```
def kargolu(toplam):
    return toplam + 99 if toplam < 1500 else toplam

baglanti.create_function("kargolu", 1, kargolu)
```

Python'da yazılan bu kargo fonksiyonu, kalemleri siparişe göre toplayan sorguda PostgreSQL'dekiyle aynı 10 tutarı verir. Her akıllı telefonda SQLite vardır, uygulamalar verisini böyle dosyalarda tutar. Ama birçok programın paylaştığı kargo kuralı gibi bir kural, hepsinin bağlandığı sunucuda durur.



Şekil 5.6 Telefonda uygulamaların SQLite dosyaları; üç programın bağlandığı sunucuda tek siparis_tutari fonksiyonu.

Özet

- Program veritabanıyla altı adımla konuşur: bağlan, imleç aç, sorguyu gönder, sonucu satır satır al, onayla (commit), kapat.
- Parametreli sorguda değer soru işaretiyle ayrı gider: komutun parçası olamaz ve aynı sorgu metni yeniden kullanılır.
- commit çağrılmadan kapatılan bağlantıdaki değişiklik kaybolur.
- Satır satır döngü her satır için ayrı komut gönderir; iş tek bir SQL komutuyla yapılabiliyorsa öyle yapılır.
- Kullanıcı tanımlı fonksiyon değer döndürür ve sorgunun içinde çağrılır; saklı yordam CALL ile çağrılan, veriyi değiştiren bir iştir.
- Veritabanındaki kod ağ trafiğini azaltır ve kuralı tek yerde tutar; bedeli sürüm takibi ve başka veritabanına taşımanın zorluğudur.

Sorular

1 **Bir Python programı UPDATE komutunu çalıştırıyor, rowcount 4 dönüyor, sonra commit() çağırmadan bağlantıyı kapatıyor. Dosya yeniden açılınca ne görülür?**

A 4 satır değişmiş olarak görülür

B Yalnız ilk satır değişmiş görülür

C Hiçbir satır değişmemiş görülür

D Dosya bozulur, açılmaz

2 **Depo programı aynı sorgu metnini (SELECT ad, stok FROM urun WHERE stok < ?) bu kez 15 değeriyle gönderiyor. Mağaza veritabanında kaç satır gelir? Yalnız sayıyı yaz.**

Cevap: _____

3 **Sorguya değeri soru işaretiyle (imlec.execute(sorgu, (esik,))) göndermenin, değeri sorgu metnine yapıştırmaya göre üstünlüğü nedir?**

A Sorgu daha az satır döndürür

B Değer komutun parçası olamaz ve aynı sorgu metni yeniden kullanılır

C commit çağırmaya gerek kalmaz

D Veritabanı değeri kendiliğinden yuvarlar

4 **Stoğu 30'un altındaki her ürüne 20 adet eklenecek. Program önce bu ürünlerin numaralarını okuyup sonra her biri için ayrı bir UPDATE gönderirse veritabanına toplam kaç komut gider? (Mağaza veritabanında stoğu 30'un altında 9 ürün var.)**

A 1
B 9
C 11
D 10

5 **Videodaki siparis_tutari fonksiyonu 1500 liranın altındaki siparişe 99 lira kargo ekliyor. 1010 numaralı siparişin kalemlerinin toplamı 798 lira. SELECT siparis_tutari(1010); kaç döndürür? Yalnız sayıyı yaz.**

Cevap: _____

6 **1001 numaralı siparişin tutarı (kargo dahil) 1496 lira. CALL taksitlendir(1001, 3); çağrısından sonra 3. ayın taksiti kaç lira olur?**

Cevap: _____

7 **Kullanıcı tanımlı fonksiyon ile saklı yordam arasındaki fark hangisinde doğru verilmiştir?**

A Fonksiyon değer döndürür ve SELECT içinde kullanılır; yordam CALL ile çağrılır ve çoğu zaman veriyi değiştirir
B Fonksiyon yalnız SQLite'ta, yordam yalnız PostgreSQL'de vardır
C Yordam SELECT içinde kullanılır, fonksiyon CALL ile çağrılır
D Fonksiyonda döngü yazılamaz, yordamda koşul yazılamaz

8 **SQLite'ta CREATE PROCEDURE sözdizimi hatası verir. Bunun nedeni en iyi hangisiyle açıklanır?**

A SQLite eski bir SQL standardını kullanır
B SQLite programın içinde çalışan bir kütüphanedir; program ile veritabanı arasında kazanılacak ağ trafiği yoktur, kural programın kendi dilinde yazılır
C SQLite'ta veriler kalıcı değildir
D SQLite yalnız tek tablolu veritabanlarını destekler

VERİTABANI SİSTEMLERİ

6 Veri bütünlüğü ve güvenlik

Bu bölümde veritabanının kendini nasıl koruduğuna bakıyoruz. Önce yanlış bir değeri tablonun kapısında durduran kısıtları, sonra bir satır silinince ona bağlı satırlara ne olacağını söyleyen yabancı anahtar kurallarını ve kendiliğinden çalışan tetikleyicileri göreceğiz. En sonda veritabanının dışarıdan gelen saldırıya karşı üç duvarını kuracağız: SQL enjeksiyonuna karşı parametreli sorgu, en az yetki ve şifrelerin tek yönlü özetle saklanması. Örnekler mağaza veritabanında; SQLite'ta, yetki örnekleri PostgreSQL 19'da çalıştırıldı.

Eksi 349 lira

videoda 44:02

Mağazada kablolu kulaklık 349 lira. Fiyatı güncelleyen bir çalışan, sayının başına yanlışlıkla bir eksi koyuyor:

```
UPDATE urun
SET fiyat = -349
WHERE id = 101;
```

Veritabanı itiraz etmez: 1 satır güncellendi. Artık bu kulaklıktan 3 tane alan müşterinin sepeti −1.047 lira tutar. Oysa ürün tablosu kurulurken birkaç kural yazılmıştı. Tablonun tanımına bakalım:

```
CREATE TABLE urun (
    id          INTEGER PRIMARY KEY,
    ad          TEXT NOT NULL,
    kategori_id INTEGER NOT NULL
        REFERENCES kategori(id),
    fiyat       NUMERIC NOT NULL,
    stok        INTEGER NOT NULL DEFAULT 0
);
```

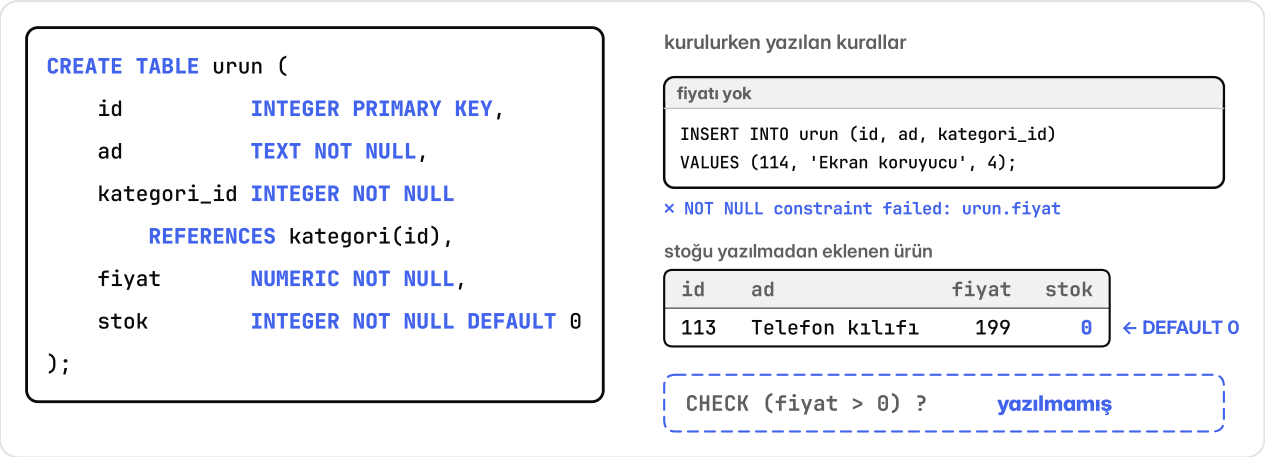
Ad ve fiyatın yanında NOT NULL var: bu sütunlar boş bırakılamaz. Fiyatı yazılmamış bir ekran koruyucu eklenmek istenince veritabanı reddeder:

```
INSERT INTO urun (id, ad, kategori_id)
VALUES (114, 'Ekran koruyucu', 4);
HATA: IntegrityError: NOT NULL constraint failed: urun.fiyat
```

Stoğun yanında DEFAULT 0 var: stoğu yazılmadan eklenen telefon kılıfı 0 stokla girer.

id	ad	fiyat	stok
113	Telefon kılıfı	199	0

Ama fiyatın sıfırdan büyük olacağı hiçbir yerde yazmıyor. Veritabanı yalnız kendisine söylenen kuralı bilir.



Şekil 6.1 Ürün tablosunun tanımı: NOT NULL ve DEFAULT var, fiyatın sıfırdan büyük olması gerektiği yazılmamış.

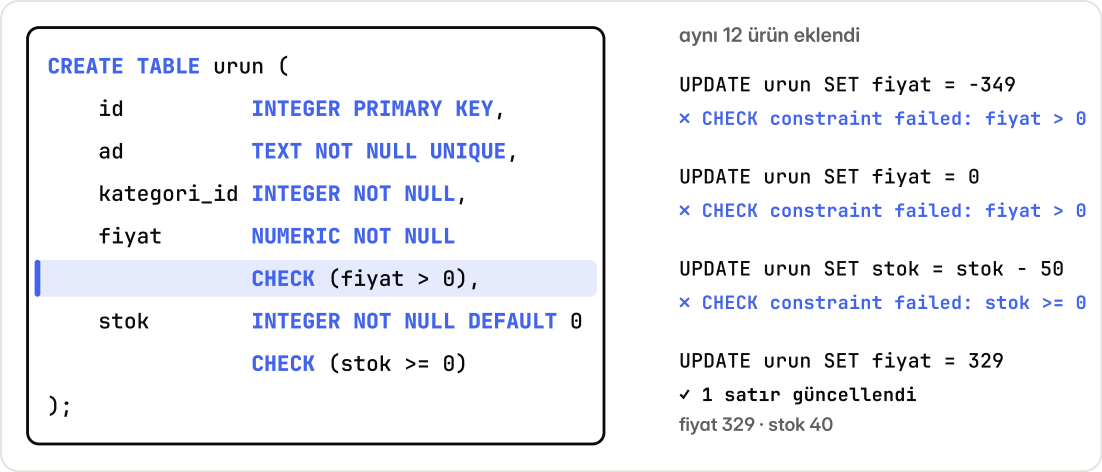
Bu denetimi programa koymak yetmez mi? Aynı tabloya mağazanın sitesi, telefon uygulaması, depo programı ve elle komut yazan çalışan yazar. Kural her birinde ayrı ayrı yazılırsa birinde unutulması yeter. Veritabanına yazılırsa, hangi program yazarsa yazsın her satır aynı kapıdan geçer. Sütunlara yazılan bu kurallara **kısıt** (constraint) denir; veritabanı her eklemede ve her güncellemede kısıtları sınar, tutmayan satırı yazmaz.

Ürün tablosunu bu kez kurallarıyla kuralım. Fiyatın yanına bir **CHECK** kısıtı gelir: fiyat sıfırdan büyük olmalı. Stoğun yanına bir tane daha: stok sıfırdan küçük olamaz. İçine aynı 12 ürün girer.

```
CREATE TABLE urun (
    id          INTEGER PRIMARY KEY,
    ad          TEXT NOT NULL UNIQUE,
    kategori_id INTEGER NOT NULL,
    fiyat       NUMERIC NOT NULL
                CHECK (fiyat > 0),
    stok        INTEGER NOT NULL DEFAULT 0
                CHECK (stok >= 0)
);
```

Aynı eksi fiyat denemesi artık reddedilir ve hata mesajı çiğnenen kuralı yazar. 0 lira da geçmez; 40 tane kalan kulaklıktan 50 tane düşmek isteyen komut da reddedilir, çünkü stok −10'a inerdi. 329 liralık indirimli fiyat ise kuralı tutar ve yazılır.

```
UPDATE urun SET fiyat = -349 WHERE id = 101;
HATA: IntegrityError: CHECK constraint failed: fiyat > 0

UPDATE urun SET stok = stok - 50 WHERE id = 101;
HATA: IntegrityError: CHECK constraint failed: stok >= 0

UPDATE urun SET fiyat = 329 WHERE id = 101;
-> 1 satır etkilendi
```



Şekil 6.2 CHECK kısıtları: eksi fiyat, 0 lira ve eksiye inen stok reddedilir; 329 liralık fiyat yazılır.

Ürünün numarası birincil anahtar olduğu için zaten tektir. Ama tek olması gereken başka sütunlar da vardır: vitrinde aynı adı taşıyan iki ürün müşteriyi şaşırtır, bu yüzden ad sütununa **UNIQUE** kısıtı yazıldı. Kablosuz fare adıyla ikinci bir ürün eklenmek istenince veritabanı reddeder:

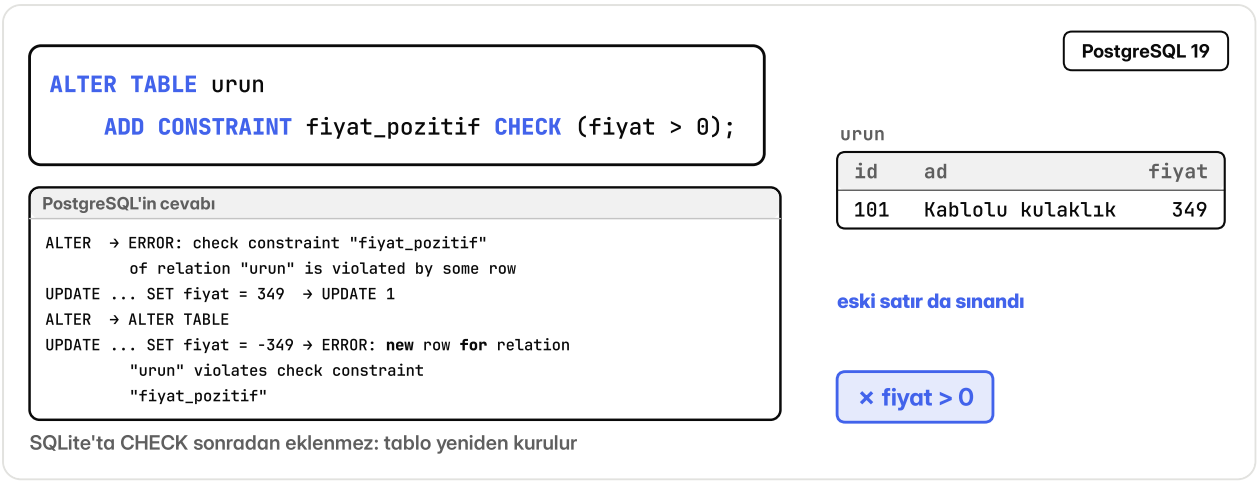
```
INSERT INTO urun (id, ad, kategori_id, fiyat)
VALUES (113, 'Kablosuz fare', 5, 299);
HATA: IntegrityError: UNIQUE constraint failed: urun.ad
```

Bir tablonun tek bir birincil anahtarı olur, UNIQUE kısıtı ise istendiği kadar sütuna yazılabilir.

Kural sonradan da eklenebilir. PostgreSQL'de ALTER TABLE ile ürün tablosuna aynı fiyat kuralı eklenir. Ama tabloda eksi fiyatlı satır dururken veritabanı kuralı kabul etmez, çünkü kuralı eklerken eski satırları da sınar. Fiyat 349'a düzeltilince kural eklenir ve o andan sonra eksi fiyat hiçbir programdan giremez (PostgreSQL 19):

```
ALTER TABLE urun
    ADD CONSTRAINT fiyat_pozitif CHECK (fiyat > 0);
ERROR:  check constraint "fiyat_pozitif" of relation "urun" is violated by some row

UPDATE urun SET fiyat = 349 WHERE id = 101;
UPDATE 1

ALTER TABLE urun
    ADD CONSTRAINT fiyat_pozitif CHECK (fiyat > 0);
ALTER TABLE

UPDATE urun SET fiyat = -349 WHERE id = 101;
ERROR:  new row for relation "urun" violates check constraint "fiyat_pozitif"
```

SQLite'ta CHECK sonradan eklenmez; tablo kurallarıyla yeniden kurulur. Açılıştaki eksi işaretini artık bir çalışanın dikkati değil, tablonun kendisi durdurur.



Şekil 6.3 PostgreSQL'de sonradan eklenen kural: eski satır da sınanır, düzeltilince kural eklenir.

Ece hesabını kapatıyor

videoda 46:26

Ece Yıldız mağazadan iki kez alışveriş yaptı, sonra hesabını kapatmak istedi. Müşteri satırı silinince Ece'nin iki siparişi ne olur?

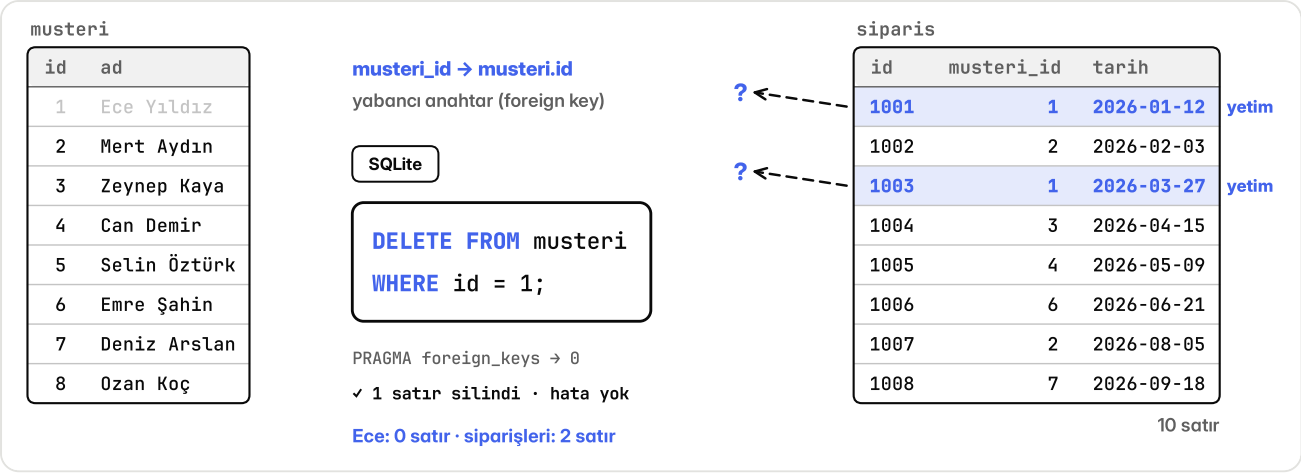
Sipariş tablosunda her siparişin yanında, veren müşterinin numarası durur. Ece'nin numarası 1; 1001 ve 1003 numaralı siparişler bu numarayı gösterir. Bu sütun bir **yabancı anahtardır** (foreign key): değeri, müşteri tablosunda var olan bir satırı göstermeli. Tabloda bu kural yazılıdır (`musteri_id INTEGER REFERENCES musteri(id)`). SQLite'ta Ece'yi silelim:

```
PRAGMA foreign_keys;
0

DELETE FROM musteri
WHERE id = 1;
-> 1 satır etkilendi
```

Bir satır silindi, hata yok. Ama iki sipariş hâlâ 1 numaralı müşteriyi gösteriyor ve böyle bir müşteri artık yok:

id	musteri_id	tarih
1001	1	2026-01-12
1003	1	2026-03-27

Bunlara **yetim satır** denir: kime ait olduğu bilinmeyen siparişler.



Şekil 6.4 Denetim kapalıyken Ece silinir; 1001 ve 1003 numaralı siparişler yetim kalır.

Kural tabloda yazılı olduğu hâlde neden işlemedi? SQLite, eski programlar bozulmasın diye yabancı anahtar denetimini varsayılan olarak kapalı tutar. Denetim PRAGMA komutuyla açılır, hem de her bağlantıda yeniden. Açınca aynı silme reddedilir; 99 numaralı, var olmayan bir müşteriye sipariş yazmak da reddedilir:

```
PRAGMA foreign_keys = ON;

DELETE FROM musteri
WHERE id = 1;
HATA: IntegrityError: FOREIGN KEY constraint failed

INSERT INTO siparis
VALUES (1011, 99, '2026-11-21');
HATA: IntegrityError: FOREIGN KEY constraint failed
```

PostgreSQL gibi sunucularda bu denetim baştan açıktır.

Peki Ece gerçekten gidecekse? Yabancı anahtar yazılırken, gösterdiği satır silinince ne olacağı da yazılabilir:

```
CREATE TABLE siparis (
    id         INTEGER PRIMARY KEY,
    musteri_id INTEGER
        REFERENCES musteri(id)
        ON DELETE RESTRICT,
    tarih      TEXT NOT NULL
);
```

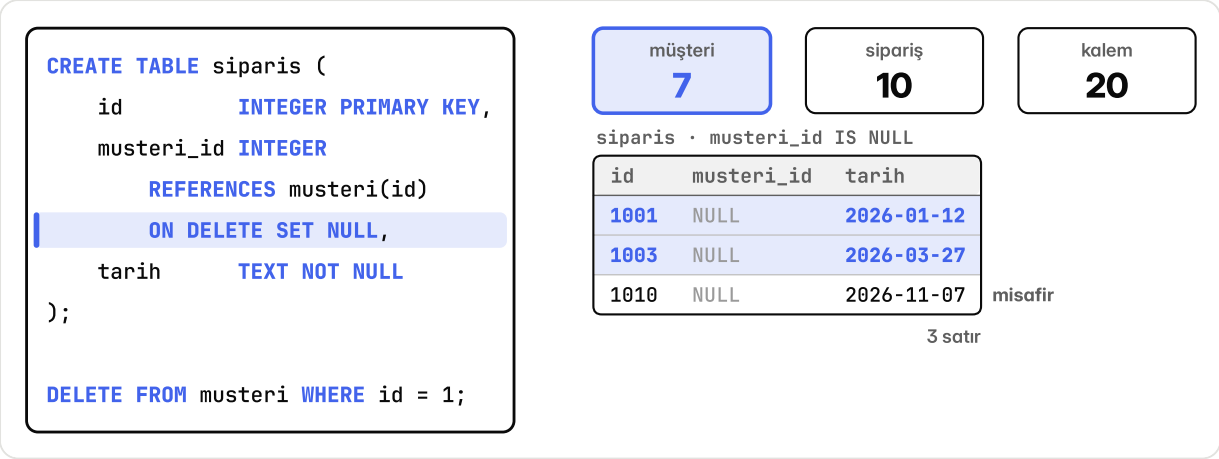
Üç seçenek var. Ece'yi her biriyle ayrı ayrı silmeyi denedik; başta 8 müşteri, 10 sipariş ve 20 kalem vardı.

- **RESTRICT**: siparişi olan müşteri silinmez (`FOREIGN KEY constraint failed`); sayılar 8, 10, 20 kalır. Hiçbir şey yazılmazsa da silme reddedilir.
- **CASCADE**: silme zincirleme ilerler. Ece'yle birlikte 2 siparişi, siparişlerle birlikte de içlerindeki 4 kalem silinir; sayılar 7, 8, 16 olur. Komut "1 satır etkilendi" der, ama 7 satır gider.

- **SET NULL**: siparişler kalır, müşteri numaraları NULL olur; sayılar 7, 10, 20.

SET NULL'dan sonra müşterisi bilinmeyen 3 sipariş vardır: Ece'nin 2 siparişi ve zaten misafir alışverişi olan 1010.

id	musteri_id	tarih
1001	NULL	2026-01-12
1003	NULL	2026-03-27
1010	NULL	2026-11-07



Şekil 6.5 Üç silme seçeneği: SET NULL'dan sonra 7 müşteri, 10 sipariş, 20 kalem ve müşterisi NULL olan 3 sipariş.

Hangisinin doğru olduğu verinin anlamına bağlıdır. Sipariş kalemi siparişsiz anlamsızdır; sipariş silinirse kalemleri de gitmeli: CASCADE. Sipariş ise kesilmiş bir faturadır; müşteri gitse de kayıt kalmalı: RESTRICT ya da SET NULL. CASCADE'i her yere yazmak tehlikelidir; tek bir DELETE fark edilmeden yüzlerce satır götürebilir.

CASCADE, bir tablodaki silmeye öteki tablonun hazır tepkisidir. Tepkiyi kendimiz de yazabiliriz: **tetikleyici** (trigger), bir tabloda ekleme, güncelleme ya da silme olunca, o işten önce ya da sonra kendiliğinden çalışan koddur. İlki işten sonra çalışır, sipariş kalemi eklenince ürünün stoğunu düşürür:

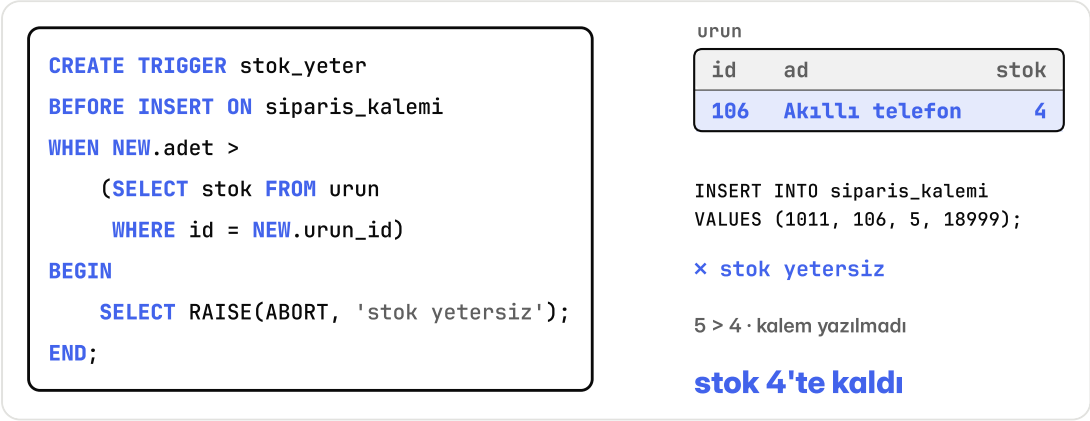
```
CREATE TRIGGER stok_azalt
AFTER INSERT ON siparis_kalemi
BEGIN
    UPDATE urun
    SET stok = stok - NEW.adet
    WHERE id = NEW.urun_id;
END;
```

`NEW` eklenen satırdır. Ozan Koç 2 kablosuz fare sipariş eder (`INSERT INTO siparis_kalemi VALUES (1011, 109, 2, 389);`); kalem yazılır yazılmaz farenin stoğu 30'dan 28'e iner. Kimse stok komutu yazmadı. İkincisi işten önce çalışır: istenen adet stoktan fazlaysa kalemi hiç yazdırmaz.

```
CREATE TRIGGER stok_yeter
BEFORE INSERT ON siparis_kalemi
WHEN NEW.adet >
    (SELECT stok FROM urun
     WHERE id = NEW.urun_id)
BEGIN
    SELECT RAISE(ABORT, 'stok yetersiz');
END;
```

4 tane kalan akıllı telefondan 5 tane istenince kalem geri döner ve stok 4'te kalır:

```
INSERT INTO siparis_kalemi
VALUES (1011, 106, 5, 18999);
HATA: IntegrityError: stok yetersiz
```



Şekil 6.6 İşten önce çalışan tetikleyici: 5 telefon istenir, stok 4; kalem yazılmaz.

Üçüncüsü kayıt tutar: bir ürünün fiyatı değişince eski fiyatı, yeni fiyatı ve tarihi fiyat geçmişi tablosuna yazar. `OLD` değişmeden önceki satırdır.

```
CREATE TRIGGER fiyat_kaydi
AFTER UPDATE OF fiyat ON urun
BEGIN
    INSERT INTO fiyat_gecmisi
        (urun_id, eski, yeni)
    VALUES (OLD.id, OLD.fiyat, NEW.fiyat);
END;
```

Oyuncu kulaklığı önce 1.199 liraya, sonra 1.099 liraya iner; arada stoğu da 20 yapılır. Geçmişte 2 satır belirir; stok değişikliği kayda düşmez, çünkü tetikleyici yalnız fiyat sütununu izler. Tarih sütunu DEFAULT CURRENT_DATE ile dolar.

urun_id	eski	yeni	tarih
104	1299	1199	2026-11-22
104	1199	1099	2026-11-22

Ece'nin siparişleri de, kulaklığın eski fiyatı da artık bir programın hatırlamasına bağlı değil; kuralı veritabanı tutar.

Bir tırnak işareti

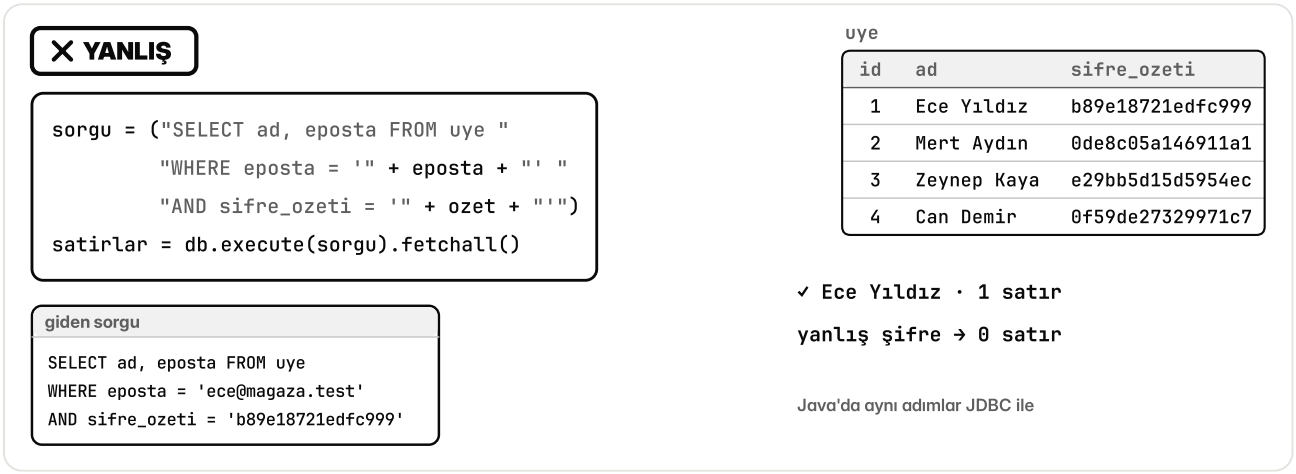
videoda 49:20

Bir mağazanın giriş ekranında iki kutu var: e-posta ve şifre. E-posta kutusuna tırnak işaretiyle başlayan kısa bir metin yazılıyor ve şifre bilinmeden bütün üyelerin listesi geliyor. Bir tırnak işareti bunu nasıl yapar?

Üyeler `uye` tablosunda durur: adı, e-postası ve şifresinin özeti. Giriş ekranının arkasında Python'la yazılmış birkaç satır var. Aşağıdaki kod YANLIŞTIR; burada yalnız saldırının nasıl işlediğini göstermek için duruyor. Program, kutulara yazılanı sorgu metninin içine yapıştırır:

```
# YANLIŞ: kullanıcının yazdığı metin sorguya yapıştırılıyor
sorgu = ("SELECT ad, eposta FROM uye "
         "WHERE eposta = '" + eposta + "' "
         "AND sifre_ozeti = '" + ozet + "'")
satirlar = db.execute(sorgu).fetchall()
```

Ece kendi e-postasını ve şifresini yazınca tek satır gelir: Ece Yıldız. Yanlış şifreyle hiç satır gelmez. Her şey çalışıyor gibi, ama bu kod yanlış.



Şekil 6.7 YANLIŞ yol: kutulardaki metin Python koduyla sorgu metnine yapıştırılır.

Kutuya içinde tek tırnak olan bir metin, örneğin `o'brien@magaza.test` yazılsın. Sorgudaki metin vaktinden önce kapanır (`'o'`), geri kalanı anlamsız kalır ve veritabanı sözdizimi hatası verir:

```
OperationalError: near "brien": syntax error
```

Saldırgan için bu hata bir işarettir: yazdığı metin sorgunun içine giriyor. Şimdi e-posta kutusuna `' OR '1'='1' --` yazılır. Yapıştırılınca giden sorgu şudur:

```
SELECT ad, eposta FROM uye WHERE eposta = '' OR '1'='1' --' AND sifre_ozeti =
'9c0073599323dc0f'
```

Sorgu "e-postası boş olan ya da 1, 1'e eşitse" diye okunur. 1 her zaman 1'e eşittir; koşul her satırda doğru çıkar. Sondaki iki tire SQL'de açıklama işaretidir; şifre denetimi açıklamaya dönüşür ve hiç çalışmaz. Sonuç 4 satır: tablodaki bütün üyeler. Kullanıcının yazdığı veri, sorgunun komutu olmuştur. Buna **SQL enjeksiyonu** (SQL injection) denir.



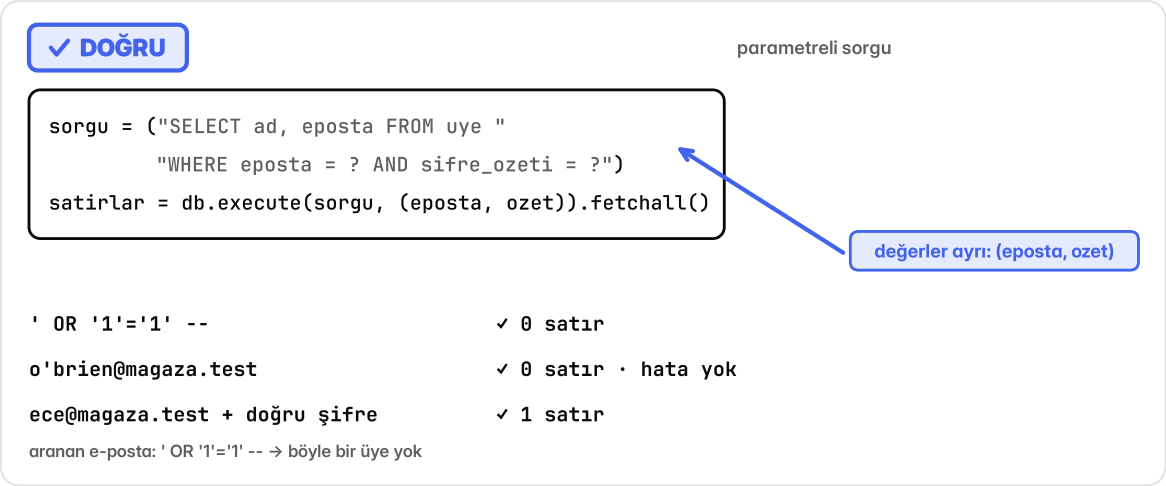
Şekil 6.8 Yapıştırılan saldırı metni: OR '1'='1' her satırda doğru, iki tireden sonrası çalışmaz; 4 üyenin hepsi gelir.

Bu yalnız bir ders örneği değil. İngiltere'nin telefon ve internet şirketlerinden TalkTalk, Ekim 2015'te SQL enjeksiyonuyla saldırıya uğradı. 156.959 müşterinin kişisel bilgileri alındı; bunların 15.656'sında banka hesap bilgisi de vardı. İngiltere'nin veri koruma kurumu (ICO) 2016'da şirkete 400.000 sterlin ceza verdi ve bu yöntemin on yılı aşkın süredir bilindiğini, önleminin de bilindiğini yazdı.

Önlem basittir: değer sorgu metnine yapıştırılmaz. Doğru yol şudur:

```
# DOĞRU: değer ? yer tutucusuyla ayrı gönderiliyor
sorgu = ("SELECT ad, eposta FROM uye "
         "WHERE eposta = ? AND sifre_ozeti = ?")
satirlar = db.execute(sorgu, (eposta, ozet)).fetchall()
```

Sorguda değerin yerine soru işareti konur, değer ayrıca gönderilir. Buna **parametreli sorgu** denir. Veritabanı sorguyu önce komut olarak okur, değeri sonra yalnız değer olarak yerleştirir. Aynı saldırı metni artık bir e-posta adresi gibi aranır; böyle bir üye yok, 0 satır. İçinde tırnak olan gerçek bir adres de hata vermez, 0 satır döner; Ece'nin doğru girişi yine 1 satır getirir.



Şekil 6.9 DOĞRU yol: soru işaretli sorgu ve ayrı gönderilen değerler; saldırı metni 0 satır getirir.

Bir duvar daha var: yetkiler. PostgreSQL'de her bağlantı bir kullanıcıyla açılır ve kullanıcı yalnız kendisine verileni yapar. Yetki kişiye değil göreve verilir; göreve **rol** denir. Depo rolüne GRANT ile iki yetki verilir: ürünleri okumak ve yalnız stok sütununu değiştirmek. Depoda çalışan Tolga bu role eklenir.

```
CREATE ROLE depo;
GRANT SELECT ON urun TO depo;
GRANT UPDATE (stok) ON urun TO depo;
CREATE ROLE tolga LOGIN;
GRANT depo TO tolga;
```

Tolga akıllı telefonun stoğunu 10 artırır (`UPDATE 1`, stok 14), fiyatına dokunamaz, müşteri tablosunu da göremez:

```
UPDATE urun SET fiyat = 1 WHERE id = 106;
ERROR:  permission denied for table urun

SELECT ad, sehir FROM musteri;
ERROR:  permission denied for table musteri
```

Destek rolündeki Pelin üyeleri bir görünümden görür; görünümde ad ve e-posta var, şifre özeti hiç yok. Üye tablosunun kendisini isteyince yetki yok hatası alır:

```
CREATE VIEW uye_destek AS
    SELECT id, ad, eposta FROM uye;
GRANT SELECT ON uye_destek TO destek;

SELECT sifre_ozeti FROM uye;
ERROR:  permission denied for table uye
```

REVOKE verileni geri alır: `REVOKE UPDATE (stok) ON urun FROM depo;` sonrasında Tolga'nın aynı stok komutu reddedilir ve stok 14'te kalır. Bunun adı **en az yetki ilkesidir**: herkes işini yapacak kadar yetki alır, fazlasını değil. Giriş ekranının bağlandığı kullanıcı da öyle olmalı; bir açık kalsa bile

saldırgan o kullanıcının göremediğini göremez.

Son duvar tablonun içinde. Şifreler tabloya düz yazılmaz, yalnız tek yönlü hash özetleri saklanır; tablo ele geçse bile şifrenin kendisi orada yoktur. Açılıştaki tırnak işaretinin gücü, metnin sorguya yapıştırılmasından geliyordu. Değer ayrı gönderilince tırnak yalnızca bir harftir.

Özet

- Kısıtlar (NOT NULL, DEFAULT, CHECK, UNIQUE) kuralı veritabanına yazar; hangi program yazarsa yazsın tutmayan satır yazılmaz.
- Yabancı anahtar var olan bir satırı göstermeli; SQLite'ta denetim varsayılan kapalıdır ve `PRAGMA foreign_keys = ON` ile her bağlantıda açılır.
- Bir satır silinince ON DELETE RESTRICT silmeyi reddeder, CASCADE bağlı satırları da siler, SET NULL bağlı satırlarda numarayı NULL yapar.
- Tetikleyici bir ekleme, güncelleme ya da silmeden önce veya sonra kendiliğinden çalışır: stoğu düşürür, yetersiz stoku reddeder, fiyat geçmişini tutar.
- Kullanıcının yazdığı metin sorguya yapıştırılırsa SQL enjeksiyonu olur; değer parametreli sorguyla ayrı gönderilir.
- Roller, GRANT ve REVOKE ile en az yetki verilir; şifreler düz değil, tek yönlü özetle saklanır.

Sorular

1 **Ürün tablosu fiyat sütununda CHECK (fiyat > 0), stok sütununda CHECK (stok >= 0) kısıtıyla kuruldu. Kablolu kulaklığın stoğu 40. Aşağıdaki komutlardan hangisi reddedilir?**

A UPDATE urun SET fiyat = 0.5 WHERE id = 101;

B UPDATE urun SET stok = stok - 40 WHERE id = 101;

C UPDATE urun SET fiyat = 0 WHERE id = 103;

D UPDATE urun SET stok = 0 WHERE id = 102;

2 **Mağaza veritabanında SQLite yabancı anahtar denetimi kapalıyken (PRAGMA foreign_keys açılmadan) Zeynep Kaya (id 3) müşteri tablosundan silinir. Sipariş tablosunda kaç sipariş yetim kalır? Yalnız sayıyı yaz.**

Cevap:

3 **Sipariş tablosundaki musteri_id yabancı anahtarı ON DELETE RESTRICT ile yazıldı ve denetim açık. Hiç sipariş vermemiş Selin Öztürk (id 5) silinmek istenirse ne olur?**

A Selin silinir; sipariş tablosu değişmez.

B Silme reddedilir: FOREIGN KEY constraint failed.

C Selin silinir; siparişlerindeki müşteri numarası NULL olur.

D Selin silinir; siparişleri de silinir.

4 **Sipariş tablosunda musteri_id, sipariş kalemi tablosunda siparis_id ON DELETE CASCADE ile yazıldı. Can Demir'in (id 4) tek siparişi 1005'tir ve içinde 2 kalem vardır. Can müşteri tablosundan silinince üç tablodan toplam kaç satır gider? Yalnız sayıyı yaz.**

Cevap: _____

5 **Sipariş tablosundaki musteri_id ON DELETE SET NULL ile yazıldı. Mağazada 1010 numaralı misafir siparişinin musteri_id'si zaten NULL. Tek siparişi 1006 olan Emre Şahin (id 6) silinince musteri_id'si NULL olan kaç sipariş olur?**

A 1

B 3

C 0

D 2

6 **Mağaza veritabanına bölümdeki stok_azalt tetikleyicisi (AFTER INSERT ON siparis_kalemi, stok = stok - NEW.adet) kuruldu. Mekanik klavyenin (id 108) stoğu 15. Yeni bir siparişe 3 mekanik klavyelik bir kalem eklenir. Klavyenin stoğu kaç olur? Yalnız sayıyı yaz.**

Cevap: _____

7 **PostgreSQL'de depo rolüne yalnız GRANT SELECT ON urun ve GRANT UPDATE (stok) ON urun verildi; Tolga bu role eklendi. Tolga olarak hangi komut çalışır?**

A UPDATE urun SET fiyat = 0 WHERE id = 101;

B UPDATE urun SET stok = 0 WHERE id = 101;

C DELETE FROM urun WHERE id = 101;

D SELECT ad, sehir FROM musteri;

8 **Giriş kodu YANLIŞ yolla yazılmış: SELECT ad, eposta FROM uye WHERE eposta = '<e-posta kutusu>' AND sifre_ozeti = '<özet>'. Saldırgan e-posta kutusuna sondaki iki tireyi unutarak x' OR '1'='1 yazar, şifre kutusuna da yanlış bir şifre. Giden sorgu WHERE eposta = 'x' OR '1'='1' AND sifre_ozeti = '...' olur. 4 üyeli tablodan kaç satır gelir?**

A 4

B 0

C 1

D Sözdizimi hatası verir

VERİTABANI SİSTEMLERİ

7 ER ve EER ile kavramsal tasarım

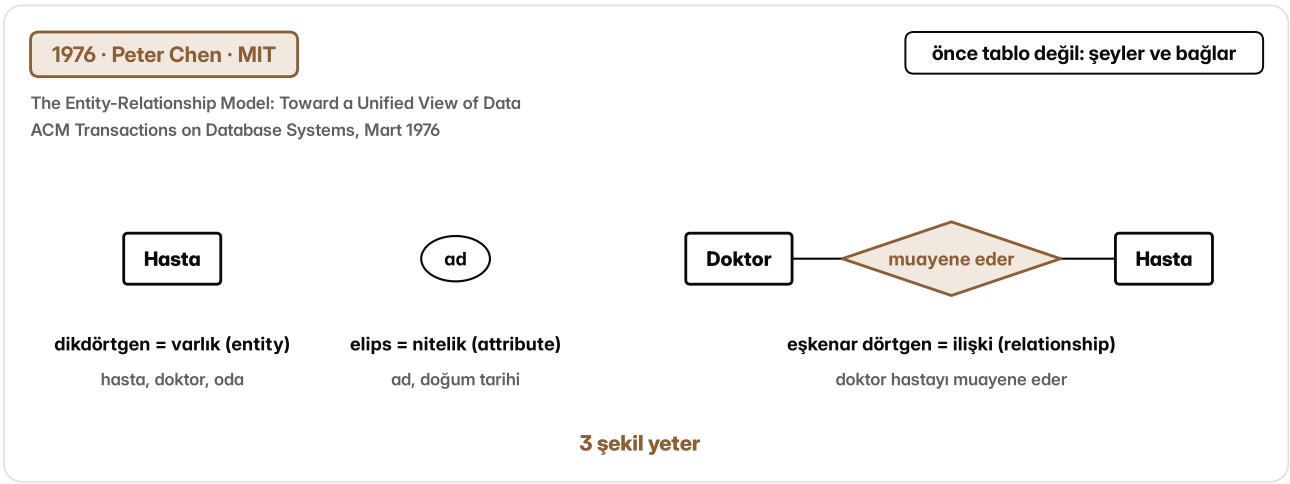
Bu bölümde bir veritabanının tablolar kurulmadan önce kâğıtta nasıl tasarlandığına bakıyoruz. Örneğimiz bir hastane: hastalar, doktorlar, odalar, yataklar ve muayeneler. Önce varlıkları ve niteliklerini çiziyoruz, sonra varlıkları birbirine bağlayan ilişkileri, en sonda doktor ve hemşire gibi alt türleri. Her adımın sonunda çizimi tablolara çeviriyoruz; tablolar SQLite'ta kuruldu.

Kâğıt

videoda 52:29

Bir hastane yazılım istiyor. Tablolar kurulmadan önce tasarım dört adımda ilerler. Önce **gereksinim**: hastaneyle konuşulur, ne tutulacağı cümle cümle yazılır ("her hastanın bir numarası var, birden çok telefonu olabilir, odalarda yataklar var"). Sonra **kavramsal tasarım**: bu cümleler bir çizime döner. Çizim tablo değildir, hastanenin dünyasını anlatır; adı **varlık ilişki diyagramı** (ER diyagramı). Üçüncü adım **mantıksal tasarım**: çizim tablolara çevrilir. En son **fiziksel tasarım**: tabloların diskte nasıl duracağı, hangi indekslerin kurulacağı. Kâğıda çizilen, ikinci adımın çizimidir.

Bu çizimi 1976'da Peter Chen yayımladı ("The Entity-Relationship Model: Toward a Unified View of Data", ACM Transactions on Database Systems, Mart 1976). Fikri, veriyi önce tablolarla değil, dünyadaki şeylerle ve aralarındaki bağlarla düşünmekti. Üç şekil yeter. Dikdörtgen **varlıktır** (entity): hasta, doktor, oda gibi ayrı ayrı tanınan şeyler. Elips **niteliktir** (attribute): varlığın bir özelliği, hastanın adı ya da doğum tarihi. Eşkenar dörtgen **ilişkidir** (relationship): iki varlığı bağlar, doktor hastayı muayene eder. Buradaki ilişki, ilişkisel modelde tabloya verilen "ilişki" (relation) adıyla karıştırılmamalı: ER'de ilişki, varlıklar arasındaki bağdır.

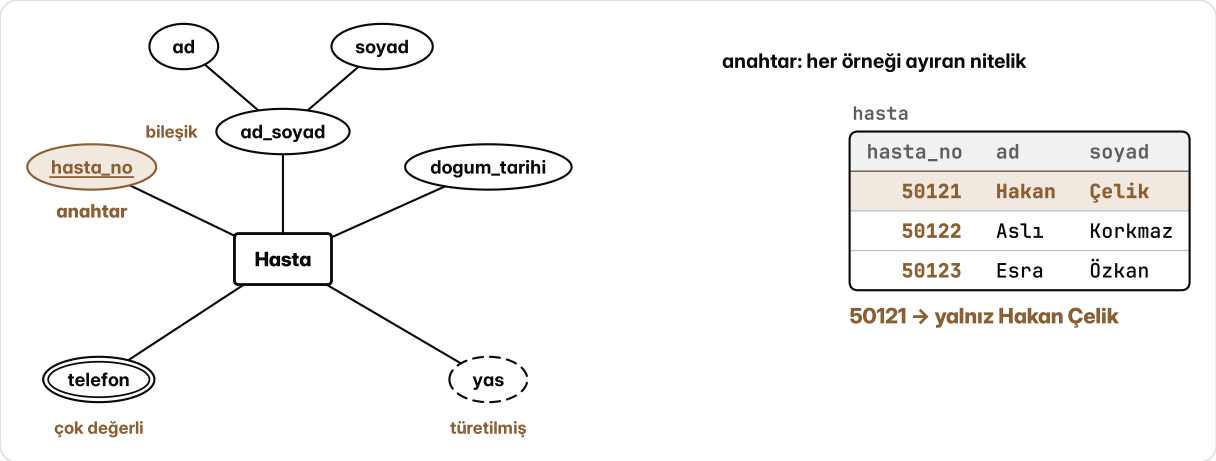


Şekil 7.1 Chen'in üç şekli: dikdörtgen varlık, elips nitelik, eşkenar dörtgen ilişki.

Hasta bir varlıktır; Hakan Çelik, Aslı Korkmaz ve Esra Özkan bu varlığın örnekleridir. Nitelikler birkaç türlüdür:

- Hasta numarası **basit** bir niteliktir, parçalanmaz.

- Ad soyad **bileşiktir**: içinde ad ve soyad var; çizimde ad soyad elipsine iki küçük elips bağlanır. Hastane hastayı soyadıyla arayacaksa soyad kendi başına durmalı.
- Telefon **çok değerlidir**: Hakan Çelik'in iki telefonu var, cebi ve evi. Çizimde çift elips.
- Yaş **türetilmiştir**: başka bir nitelikten hesaplanır, çizimde kesik çizgili elips. Hakan 14 Mart 1990'da doğdu; 22 Kasım 2026'da 36, 14 Mart 2027'de 37 yaşında. Tabloya yazılan yaş her doğum gününde eskir; bu yüzden doğum tarihi saklanır, yaş gerektiğinde hesaplanır.
- Hasta numarası aynı zamanda **anahtardır**: her örneği ötekilerden ayırır, çizimde adının altı çizilir. 50121 yalnız Hakan Çelik'tir.

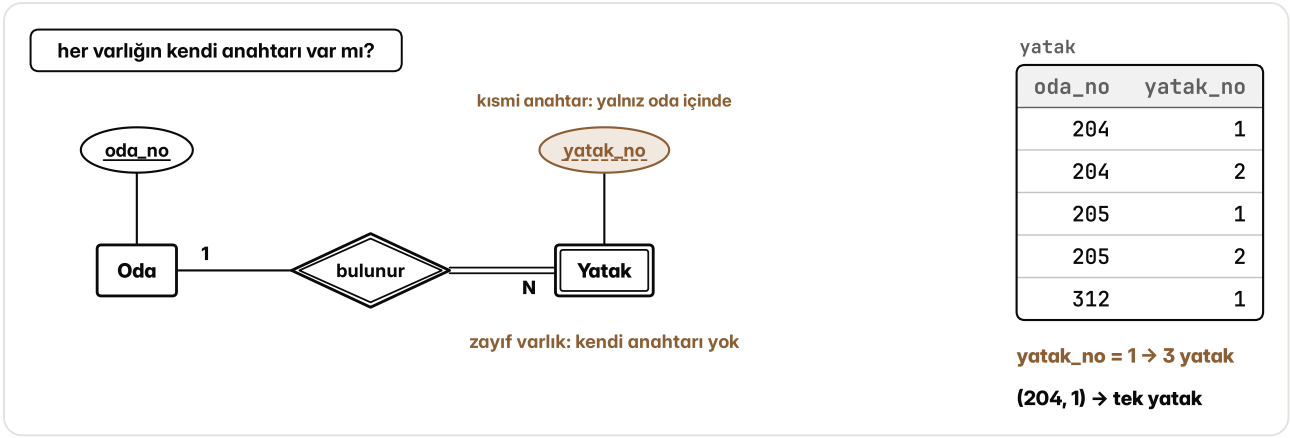


Şekil 7.2 Hasta varlığı ve nitelikleri: anahtar hasta_no altı çizili, telefon çift elips, yas kesik elips.

Her varlığın kendi anahtarı var mı? Yataklara bakalım:

oda_no	yatak_no
204	1
204	2
205	1
205	2
312	1

Yatak numarası tek başına hiçbir yatağı ayırmaz: hastanede 3 tane 1 numaralı yatak var. Yatak ancak odasıyla tanınır: 204'ün 1 numaralı yatağı. Kendi anahtarı olmayan, sahibi üzerinden tanınan varlığa **zayıf varlık** (weak entity) denir. Çizimde çift çizgili dikdörtgendir; onu odaya bağlayan ilişki de çift çizgilidir. Yatak numarası **kısmi anahtardır**, altı kesik çizilir: yalnız oda içinde ayırır.



Şekil 7.3 Zayıf varlık: Yatak çift dikdörtgen, bulunur ilişkisi çift eşkenar dörtgen, yatak_no'nun altı kesik.

Çizimi tablolara çevirirken her varlık bir tablo, nitelikleri sütun olur. Bileşik ad soyad iki sütuna ayrılır, türetilmiş yaş hiç sütun olmaz. Bir tablonun her hücresine tek değer yazıldığı için çok değerli telefon kendi tablosuna gider; hasta numarası ile telefon birlikte anahtardır. Zayıf yatağın anahtarı da odanın numarasıyla kendi numarasıdır; oda numarası aynı zamanda oda tablosunu gösteren yabancı anahtardır:

```
CREATE TABLE hasta_telefon (
    hasta_no INT NOT NULL,
    telefon TEXT NOT NULL,
    PRIMARY KEY (hasta_no, telefon),
    FOREIGN KEY (hasta_no) REFERENCES hasta
);

CREATE TABLE yatak (
    oda_no INT NOT NULL,
    yatak_no INT NOT NULL,
    PRIMARY KEY (oda_no, yatak_no),
    FOREIGN KEY (oda_no) REFERENCES oda
);
```

Veritabanı bu çizimi korur. 204'ün 1 numaralı yatağı ikinci kez eklenemez, olmayan bir odaya yatak eklenemez; ama 312 numaralı odaya 2 numaralı yatak sorunsuz girer (SQLite'ta yabancı anahtar denetimi `PRAGMA foreign_keys = ON;` ile açılır):

```
INSERT INTO yatak VALUES (204, 1);
HATA: IntegrityError: UNIQUE constraint failed: yatak.oda_no, yatak.yatak_no

INSERT INTO yatak VALUES (999, 1);
HATA: IntegrityError: FOREIGN KEY constraint failed

INSERT INTO yatak VALUES (312, 2);
-> 1 satır etkilendi
```

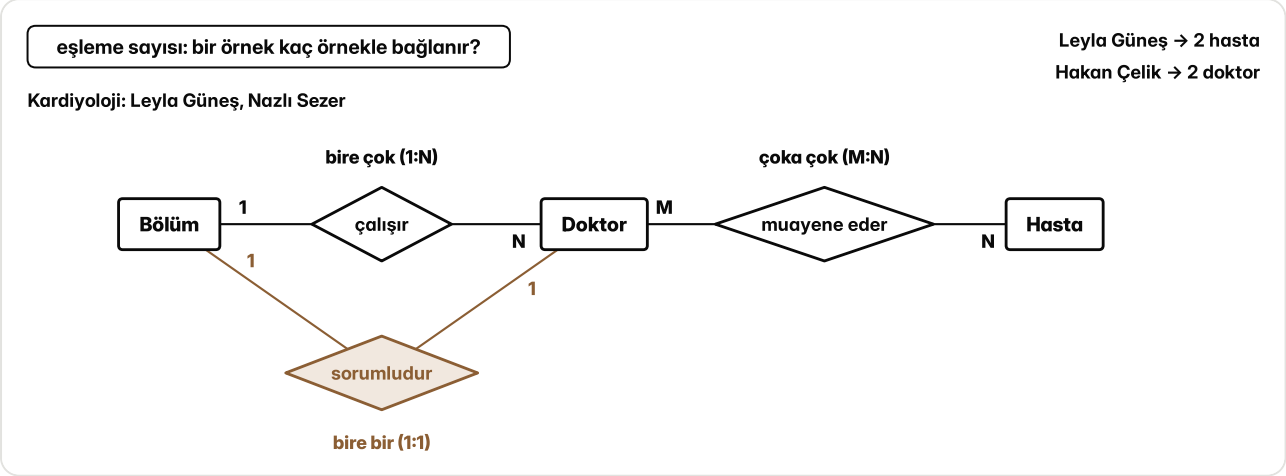
Üç soru

videoda 55:51

Bir doktor kaç hastaya bakar, bir hasta kaç doktora görünür, bir bölümde kaç doktor çalışır? Bu soruların cevabı tabloları değiştirir.

Varlıkları bağlayan ilişki eşkenar dörtgenle çizilir: doktor hastayı muayene eder. İlişkiye katılan varlık sayısına **derece** denir. Muayene iki varlığı bağlar, derecesi 2'dir; en yaygını budur. Reçete üç varlığı bağlar: hangi doktor hangi hastaya hangi ilacı yazdı (derece 3). Bir varlık kendisiyle de bağlanabilir: bir doktor, uzmanlık eğitimi alan başka bir doktorun eğiticisidir (derece 1).

İlişkinin her ucuna bir sayı yazılır: bir örnek karşı taraftaki kaç örnekle bağlanabilir. Buna **eşleme sayısı** (cardinality) denir. Bir doktor tek bir bölümde çalışır, bir bölümde birden çok doktor çalışır: **bire çok** (1:N); kardiyolojide Leyla Güneş ile Nazlı Sezer var. Doktorla hasta arasında iki uç da çoktur: Leyla Güneş 2 ayrı hastaya baktı, Hakan Çelik 2 ayrı doktora göründü; bu **çoka çok** (M:N). Her bölümün tek bir sorumlu doktoru olur, bir doktor da yalnız bir bölümün sorumlusu olabilir: **bire bir** (1:1).

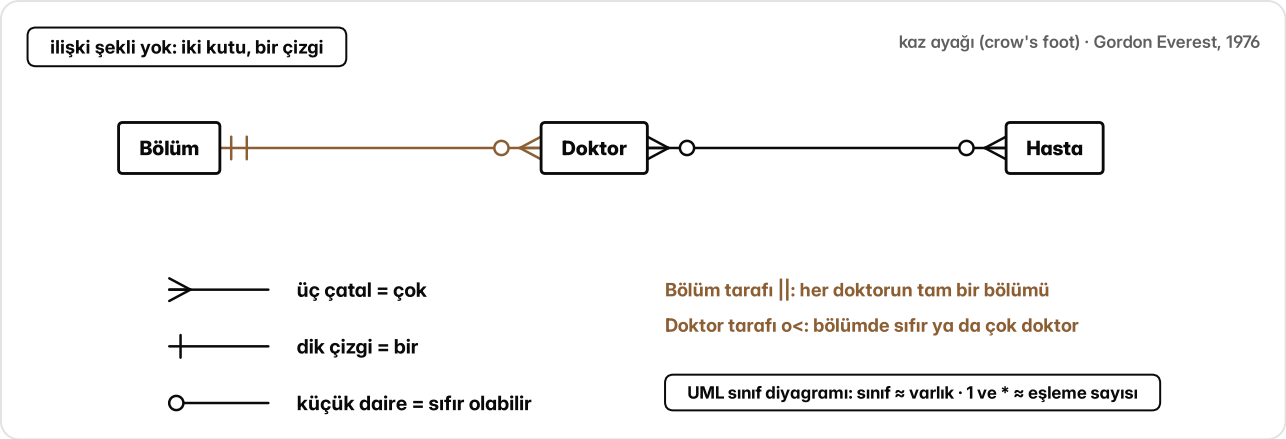


Şekil 7.4 Bölüm, doktor ve hasta arasındaki üç ilişki: çalışır 1:N, muayene eder M:N, sorumludur 1:1.

Bir soru daha: her örnek ilişkiye katılmak zorunda mı? Her doktor bir bölümde çalışmak zorunda; bölümsüz doktor olmaz. Buna **tam katılım** denir, çizimde çift çizgi. Bölümler için öyle değil: yeni açılan nörolojide henüz doktor yok. Bu **kısmi katılımdır**, tek çizgi.

Muayenenin bir tarihi var. Bu tarih doktorun niteliği değildir, çünkü bir doktorun yüzlerce muayenesi olur; hastanın da değildir. Tarih, belli bir doktorla belli bir hastanın buluşmasına aittir; bu yüzden ilişkinin elipsine yazılır. Varlıklar gibi ilişkilerin de niteliği olur.

Aynı bilgi başka bir gösterimle de çizilir. Bugünkü çizim araçlarının çoğu **kaz ayağını** (crow's foot) kullanır: ilişki ayrı bir şekil olmaz, iki kutu bir çizgiyle bağlanır. Çizginin ucundaki üç çatal "çok", dik çizgi "bir", küçük daire "sıfır olabilir" demektir. Böylece eşleme sayısı da katılım da çizginin ucunda okunur. Yazılımcıların UML sınıf diyagramı da buna benzer: kutular varlıklara, çizgi uçlarındaki sayılar eşleme sayısına karşılık gelir.



Şekil 7.5 Kaz ayağı gösterimi: Bölüm tarafında iki dik çizgi, Doktor ve Hasta uçlarında daire ve üç çatal.

Tablolara geçerken eşleme sayısı belirleyicidir. Bire çokta yabancı anahtar çok olan tarafa konur: her doktorun satırına bölümünün numarası yazılır. Tam katılım bu sütunun boş bırakılamamasıyla korunur:

```
CREATE TABLE doktor (
    doktor_id INT NOT NULL PRIMARY KEY,
    ad TEXT NOT NULL,
    bolum_id INT NOT NULL,
    FOREIGN KEY (bolum_id) REFERENCES bolum
);

INSERT INTO doktor VALUES (7004, 'Yeni Doktor', NULL);
HATA: IntegrityError: NOT NULL constraint failed: doktor.bolum_id
```

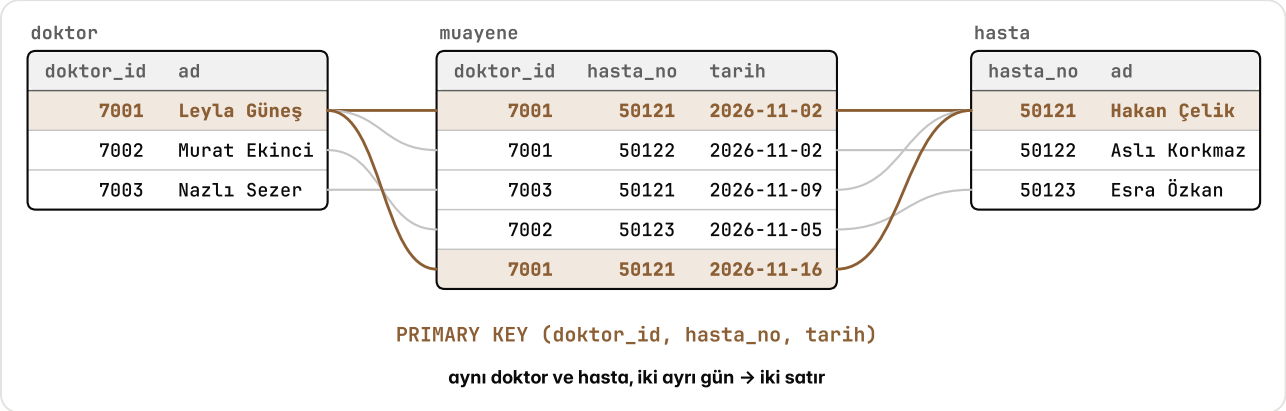
Bire birde yabancı anahtar iki taraftan birine konur ve tekrarı yasaklanır: bölüm tablosunda sorumlu doktorun numarası. Leyla Güneş'i nörolojiye de sorumlu yazmak istersek veritabanı reddeder:

```
CREATE TABLE bolum (
    bolum_id INT NOT NULL PRIMARY KEY,
    ad TEXT NOT NULL,
    sorumlu_id INT UNIQUE,
    FOREIGN KEY (sorumlu_id) REFERENCES doktor
);

UPDATE bolum SET sorumlu_id = 7001 WHERE bolum_id = 3;
HATA: IntegrityError: UNIQUE constraint failed: bolum.sorumlu_id
```

Çoka çokta yabancı anahtar iki tarafa da sığmaz, çünkü iki uçta da birden çok değer var. Bunun için **ara tablo** kurulur: muayene tablosu. Her satırında bir doktor, bir hasta ve ilişkinin niteliği olan tarih durur. Hakan Çelik aynı doktora başka bir gün yine gelebilir, bu yüzden tarih de anahtara girer:

```
CREATE TABLE muayene (
    doktor_id INT NOT NULL,
    hasta_no INT NOT NULL,
    tarih DATE NOT NULL,
    PRIMARY KEY (doktor_id, hasta_no, tarih),
    FOREIGN KEY (doktor_id) REFERENCES doktor,
    FOREIGN KEY (hasta_no) REFERENCES hasta
);
```

doktor_id	hasta_no	tarih
7001	50121	2026-11-02
7001	50122	2026-11-02
7003	50121	2026-11-09
7002	50123	2026-11-05
7001	50121	2026-11-16



Şekil 7.6 Ara tablo: muayene satırları doktor ve hasta tablolarına bağlı; Leyla Güneş ile Hakan Çelik'in iki ayrı günkü muayenesi vurgulu.

Bölümle doktor arasında ara tablo gerekmez; doktorun satırındaki tek bir bölüm numarası yeter. Çizginin ucuna yazılan bir ya da çok, tabloların kalıbını belirler.

Aynı tablo mu?

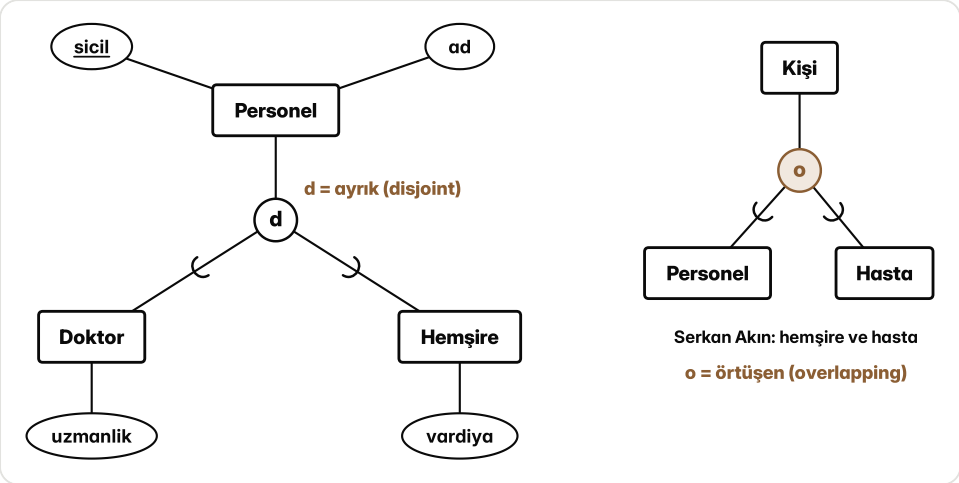
videoda 58:46

Bir hastanede doktorlar da hemşireler de personeldir; hepsinin sicil numarası ve adı var. Ama doktorun bir uzmanlığı, hemşirenin bir vardiyası var. Bunlar aynı tabloda mı durmalı, ayrı tablolarda mı?

Bu durumu anlatmak için **genişletilmiş varlık ilişki diyagramı** (EER diyagramı) kullanılır. Personel **üst sınıftır** (superclass); doktor ve hemşire onun **alt sınıflarıdır** (subclass). Alt sınıf üst sınıfın her şeyini alır: her doktorun bir sicili ve adı vardır. Üstüne kendi niteliklerini ekler: doktorun uzmanlığı, hemşirenin vardiyası. Kendi ilişkileri de olur: hastayı doktor muayene eder, hemşire etmez. Çizimde personel ile alt sınıflar arasına bir daire konur. Bu yapıya iki yönden varılır: personelden başlayıp

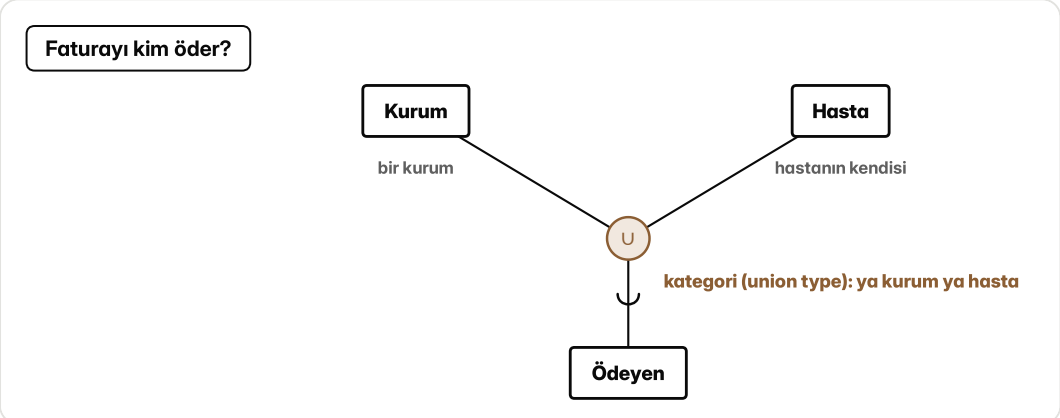
onu doktor ve hemşire diye ayırmak **özelleştirmedir** (yukarıdan aşağı); doktor ile hemşireyi ayrı ayrı çizip ortak sicili ve adı bir üst sınıfta toplamak **genellemedir** (aşağıdan yukarı). Sonunda aynı çizim çıkar.

İki soru çizimi tamamlar. Birincisi: bir kişi iki alt sınıfta birden olabilir mi? Bu hastanede biri hem doktor hem hemşire olamaz; buna **ayrık** (disjoint) denir, dairenin içine d yazılır. Kişileri personel ve hasta diye ayırırsak durum değişir: bir hemşire de hasta olarak muayene olabilir. Bu **örtüşen** (overlapping) bir ayrımdır, dairede o yazılır. İkincisi: her personel bir alt sınıfa girmek zorunda mı? Radyoloji teknikeri Volkan Bulut ne doktor ne hemşire; öyleyse özelleştirme **kısmidir** ve personelden daireye tek çizgi çekilir. Her personel ya doktor ya hemşire olsaydı **tam** olurdu, çizgi çift çekilirdi.



Şekil 7.7 EER: Personel üst sınıf, dairede d (ayrık); yanda Kişi'nin Personel ve Hasta diye örtüşen (o) ayrımı.

Bir de ters durum var: faturayı bazen bir kurum, bazen hastanın kendisi öder. Ödeyen, iki ayrı üst sınıftan gelebilen bir alt sınıftır: ya bir kurumdur ya bir hasta. Buna **kategori** (union type) denir; dairenin içine birleşim işareti ∪ yazılır.



Şekil 7.8 Kategori: Kurum ve Hasta birleşim dairesinde buluşur, Ödeyen ikisinden birinden gelir.

Alt sınıflar tablolara üç yolla çevrilir.

Birinci yol: bütün personel tek tabloda, yanında bir tür sütunu. Doktorları listelemek kolaydır. Ama doktorun vardiyası, hemşirenin uzmanlığı boş kalır: 5 personelde bu iki sütunun 10 hücresinden 6'sı NULL.

ad	tur	uzmanlik	vardiya
Leyla Güneş	doktor	Kardiyoloji	NULL
Murat Ekinci	doktor	Ortopedi	NULL
Serkan Akın	hemşire	NULL	gece
Gülay Tunç	hemşire	NULL	gündüz
Volkan Bulut	tekniker	NULL	NULL

Üstelik bir hemşireye uzmanlık yazılsa veritabanı bunu engellemez:

```
INSERT INTO personel VALUES
(7013, 'Deneme Hemşire', 'hemşire',
 'Kardiyoloji', 'gece');
-> 1 satır etkilendi
```

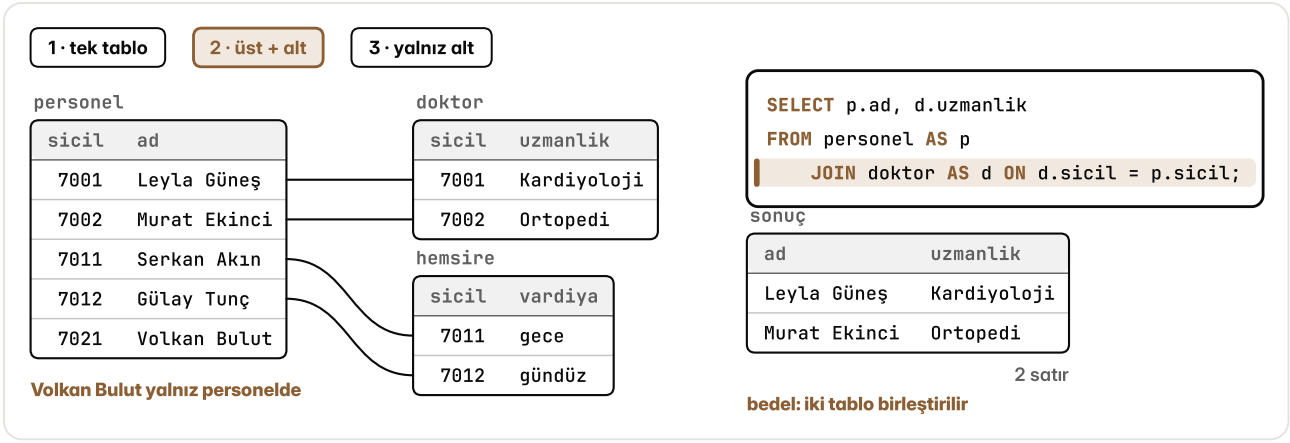
İkinci yol: üst sınıfa bir tablo, her alt sınıfa bir tablo. Personel tablosunda sicil ve ad, alt tablolarda sicille birlikte kendi nitelikleri durur. Alt tablonun anahtarı aynı zamanda personeli gösteren yabancı anahtardır:

```
CREATE TABLE doktor (
    sicil INT NOT NULL PRIMARY KEY,
    uzmanlik TEXT NOT NULL,
    FOREIGN KEY (sicil) REFERENCES personel
);
```

Boş hücre kalmaz; uzmanlık artık zorunlu ve yalnız doktorda. Tekniker de sorun olmaz, yalnız personel tablosunda durur. Bedeli şu: bir doktorun adını görmek için iki tablo birleştirilir.

```
SELECT p.ad, d.uzmanlik
FROM personel AS p
    JOIN doktor AS d ON d.sicil = p.sicil;
```

ad	uzmanlik
Leyla Güneş	Kardiyoloji
Murat Ekinci	Ortopedi



Şekil 7.9 İkinci yol: doktor ve hemşire satırları personel satırlarına bağlı, Volkan Bulut yalnız personelde; birleştirme sorgusu ve sonucu.

Üçüncü yol: yalnız alt sınıflara tablo. Sicil ve ad her alt tabloda tekrarlanır; doktor tek tablodan okunur. Ama Volkan Bulut'a yer yoktur: bütün personel listelenince 4 kişi çıkar, tekniker görünmez.

```
SELECT sicil, ad FROM doktor
UNION ALL
SELECT sicil, ad FROM hemsire;
```

sicil	ad
7001	Leyla Güneş
7002	Murat Ekinci
7011	Serkan Akın
7012	Gülay Tunç

Üstelik aynı sicil iki tabloya birden yazılabilir; veritabanı ayrıklığı kendiliğinden korumaz. Hemşire Serkan Akın doktor tablosuna da girer; aynı sorgu artık 5 satır döndürür:

```
INSERT INTO doktor VALUES (7011, 'Serkan Akın', 'Nöroloji');
-> 1 satır etkilendi
```

Bu yol ancak özelleştirme hem tam hem ayrıksa işe yarar.

Seçimi çizimdeki iki işaret verir: bu hastanedeki gibi kısmi bir ayrımda ikinci yol güvenlidir; alt sınıfların kendine özgü niteliği azsa birinci yol, ayrım hem tam hem ayrıksa üçüncü yol da olur.

Özet

- Tasarım dört adımdır: gereksinim, kavramsal tasarım (ER diyagramı), mantıksal tasarım (tablolar), fiziksel tasarım.
- Chen'in gösteriminde dikdörtgen varlık, elips nitelik, eşkenar dörtgen ilişkidir; anahtar altı çizili, çok değerli nitelik çift elips, türetilmiş nitelik kesik elips, zayıf varlık çift dikdörtgendir.
- Her varlık bir tablo olur; çok değerli nitelik ayrı tabloya gider, türetilmiş nitelik saklanmaz, zayıf varlığın anahtarı sahibinin anahtarıyla kurulur.
- Bire çokta yabancı anahtar çok tarafa, bire birde tekrarı yasak olarak bir tarafa konur; çoka çok ilişki ara tablo ister ve ilişkinin niteliği orada durur.
- EER'de alt sınıflar ayrık (d) ya da örtüşen (o), tam (çift çizgi) ya da kısmi (tek çizgi) olur; kategori ise birden çok üst sınıftan gelen alt sınıftır.
- Alt sınıflar için üç yol vardır: tek tablo, üst ve alt tablolar, yalnız alt tablolar; hangisinin uygun olduğunu tam/kısmi ve ayrık/örtüşen işaretleri söyler.

Sorular

1 **Hastane tasarımında her hastanın birden çok e-posta adresi tutulacak. Chen gösteriminde bu nitelik nasıl çizilir?**

A Kesik çizgili elips

B Çift elips

C Adı altı çizili elips

D Çift çizgili dikdörtgen

2 **Yaş türetilmiş bir nitelik olduğu için saklanmıyor, doğum tarihinden hesaplanıyor. Doğum tarihi 10 Mayıs 1985 olan hasta 9 Mayıs 2027'de kaç yaşındadır? Yalnız sayıyı yaz.**

Cevap:

3 **Yatak tablosunun anahtarı (oda_no, yatak_no), oda_no da oda tablosunu gösteren yabancı anahtar. Tabloda (204, 1), (204, 2), (205, 1), (205, 2), (312, 1) satırları var; oda tablosunda yalnız 204, 205 ve 312 numaralı odalar var. Sırayla (205, 1), (205, 3), (312, 1), (999, 2) ve (312, 2) eklenmek isteniyor. Bu beş eklemeden kaçı kabul edilir? Yalnız sayıyı yaz.**

Cevap:

4 **Her hastanın tek bir aile hekimi var, bir aile hekiminin ise birçok hastası var. Bu bire çok ilişki tablolara nasıl çevrilir?**

A Aile hekimi tablosuna bir hasta numarası sütunu eklenir.

B Hasta ile aile hekimi arasına ara tablo kurulur.

C Hasta tablosuna aile hekiminin numarası yabancı anahtar olarak eklenir.

D Hekim ile hastalar tek bir tabloda tutulur.

5 **Muayene ara tablosunda (doktor_id, hasta_no, tarih) satırları şunlar: (7001, 50121, 2026-11-02), (7001, 50122, 2026-11-02), (7003, 50121, 2026-11-09), (7002, 50123, 2026-11-05), (7001, 50121, 2026-11-16). Hakan Çelik (50121) kaç satırda geçer ve kaç farklı doktora görünmüştür?**

A 2 satır, 2 doktor

B 3 satır, 3 doktor

C 2 satır, 1 doktor

D 3 satır, 2 doktor

6 **Kaz ayağı gösteriminde Bölüm ile Doktor arasındaki çizginin Bölüm ucunda iki dik çizgi, Doktor ucunda küçük bir daire ve üç çatal var. Bu çizim ne söyler?**

A Her doktorun tam bir bölümü var; bir bölümde sıfır ya da çok doktor olabilir.

B Her bölümün tam bir doktoru var; bir doktor sıfır ya da çok bölümde çalışabilir.

C Her doktor en az bir bölümde çalışır; bir bölümde en çok bir doktor olur.

D Bölüm ile doktor arasında çoka çok ilişki var.

7 **Personel üst sınıfının alt sınıfları doktor ve hemşire. Hangi durumda yalnız alt sınıflara tablo kurmak (personel tablosu olmadan) güvenle kullanılabilir?**

A Özelleştirme kısmi ve ayrıksa

B Özelleştirme tam ve örtüşense

C Özelleştirme hem tam hem ayrıksa

D Özelleştirme kısmi ve örtüşense

8 **Tek tablo yolunda personel(sicil, ad, tur, uzmanlik, vardiya) tablosuna 2 doktor, 3 hemşire ve 1 tekniker konuyor. Doktorların vardiyası, hemşirelerin uzmanlığı ve teknikerin ikisi de boş. Uzmanlik ve vardiya sütunlarında toplam kaç NULL hücre olur? Yalnız sayıyı yaz.**

Cevap: _____

VERİTABANI SİSTEMLERİ

8 Normalizasyon

Bu bölümde tek büyük bir tablonun neden dert çıkardığını ve tablonun hangi kurala göre bölüneceğini görüyoruz. Örneğimiz bir ortaokulun haftalık ders programı. Fonksiyonel bağımlılık, kapanış ve aday anahtardan sonra birinci, ikinci ve üçüncü normal formu, Boyce-Codd normal formunu, dördüncü normal formu ve bilerek normalden çıkmayı ele alacağız.

Bir telefon, kaç satır?

videoda 1:01:58

Bir ortaokul bütün haftalık ders programını tek bir tabloda tutuyor. Okulda 4 şube var: 6-A, 6-B, 7-A ve 7-B. Her şubenin haftada 30 ders saati var; her ders saati bir satır.

tablo	sütunlar	satır
ders_programi	sinif, gun, saat, ders, ogretmen, telefon, derslik	120

Satırı tanıyan anahtar sınıf, gün ve saattir. 6-A'nın pazartesisi şöyle:

saat	ders	ogretmen	telefon
1	Türkçe	Gönül Ersoy	0505 020 41 17
2	Türkçe	Gönül Ersoy	0505 020 41 17
3	Matematik	Kemal Uçar	0505 020 12 40
4	Matematik	Kemal Uçar	0505 020 12 40
5	Fen Bilimleri	Orhan Yazıcı	0505 020 27 05
6	Fen Bilimleri	Orhan Yazıcı	0505 020 27 05

Türkçe öğretmeni Gönül Ersoy 4 şubenin de Türkçesine giriyor, haftada 24 saat. Her ders saatinin satırına telefonu da yazılmış; aynı numara tabloda 24 kez duruyor. Telefonu değişince 24 satırı güncellemek gerekir. Bu tekrar üç dert doğurur.

Güncelleme anomalisi (update anomaly): numara 24 satırın hepsinde değişmeli. Diyelim 6-A'nın sınıf öğretmeni yalnız kendi sınıfının satırlarını düzeltti:

```
UPDATE ders_programi
SET telefon = '0505 020 88 03'
WHERE ogretmen = 'Gönül Ersoy'
    AND sinif = '6-A';
```

Komut 6 satırı değiştirir, 18 satır eski numarada kalır. Tablo artık Gönül Hanım için iki ayrı telefon söylüyor ve hangisinin doğru olduğunu bilmiyor:

telefon	satir
0505 020 41 17	18
0505 020 88 03	6



Şekil 8.1 Yarım kalan güncelleme: 6 satır yeni numarada, 18 satır eski numarada; tablo iki telefon söylüyor.

Ekleme anomalisi (insertion anomaly): okula derse girmeyen bir rehber öğretmen, Zerrin Kahraman atandı. Telefonu kayıtlı olmalı, ama satırın anahtarı sınıf, gün ve saat boş bırakılamaz:

```
INSERT INTO ders_programi
    (ogretmen, telefon)
VALUES ('Zerrin Kahraman',
    '0505 020 15 26');
HATA: IntegrityError: NOT NULL constraint failed: ders_programi.sinif
```

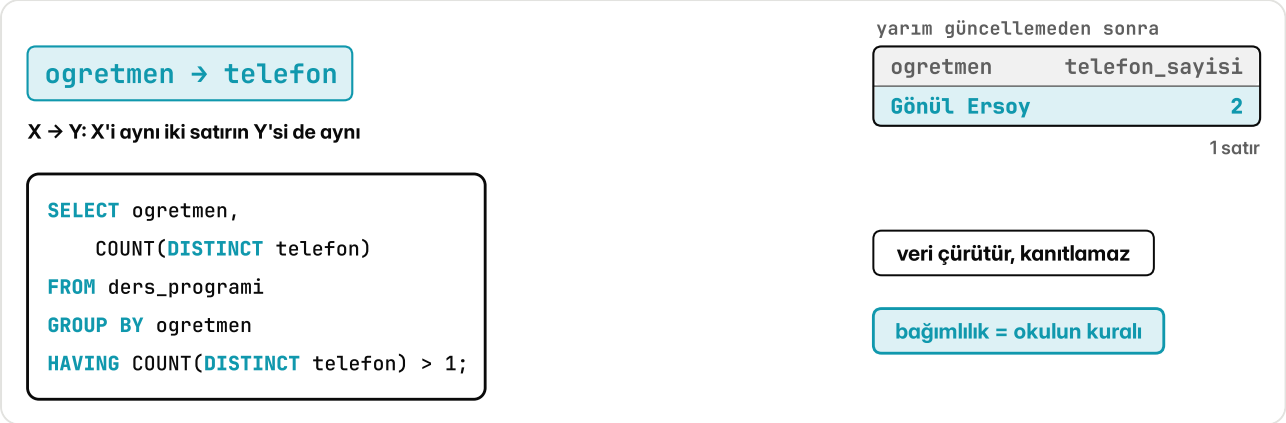
Silme anomalisi (deletion anomaly): 6-B seçmeli satranç dersini bıraktı. Satranç öğretmeni Erhan Bozkurt yalnız bu sınıfa, haftada 2 saat giriyordu. DELETE komutu bu 2 satırı siler; Erhan Bey'in telefonunu soran sorgu artık 0 satır döndürür.

```
DELETE FROM ders_programi
WHERE sinif = '6-B'
    AND ders = 'Satranç';
```

Üç derdin kökü aynı: telefon satırın tamamına değil, yalnız öğretmene bağlı. Öğretmeni bilen telefonu bilir. Buna **fonksiyonel bağımlılık** (functional dependency) denir ve `ogretmen → telefon` diye yazılır. Genel kural: X → Y ise X'i aynı olan iki satırın Y'si de aynıdır. Bunu tabloda sınayabiliriz:

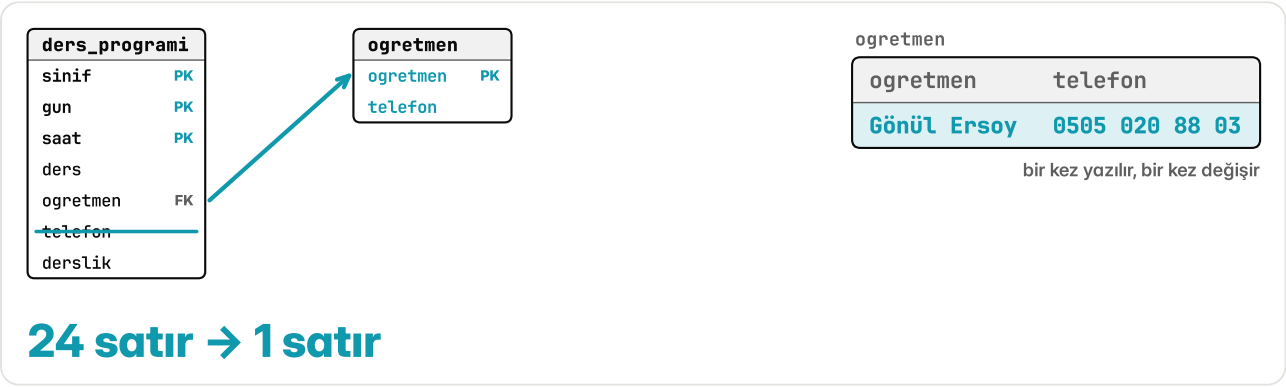
```
SELECT ogretmen,
    COUNT(DISTINCT telefon)
FROM ders_programi
GROUP BY ogretmen
HAVING COUNT(DISTINCT telefon) > 1;
```

Temiz tabloda sonuç 0 satırdır; yarım güncellemeden sonra Gönül Ersoy 2 telefonla listeye düşer. Yine de veri bir bağımlılığı yalnız çürütebilir, kanıtlayamaz. Bağımlılık okulun kuralıdır; satırlar ona ya uyar ya uymaz.



Şekil 8.2 ogretmen → telefon bağımlılığı ve sınama sorgusu; yarım güncellemeden sonra Gönül Ersoy 2 telefonla listede.

Çözüm, bağımlılığı kendi tablosuna taşımaktır: `ogretmen(ogretmen, telefon)` tablosunda her öğretmen bir satır, ders programında yalnız öğretmenin adı kalır. Bir tabloyu bağımlılıklarına göre böyle bölmeye **normalizasyon** denir. Şimdi telefon tek satırda değişir (1 satır), Zerrin Hanım sınıfsız saatsiz eklenir (tablo 13 satır), satranç dersleri silinse de Erhan Bey öğretmen tablosunda kalır. Bedeli küçük: bir dersin öğretmeninin telefonunu görmek için iki tablo birleştirilir.



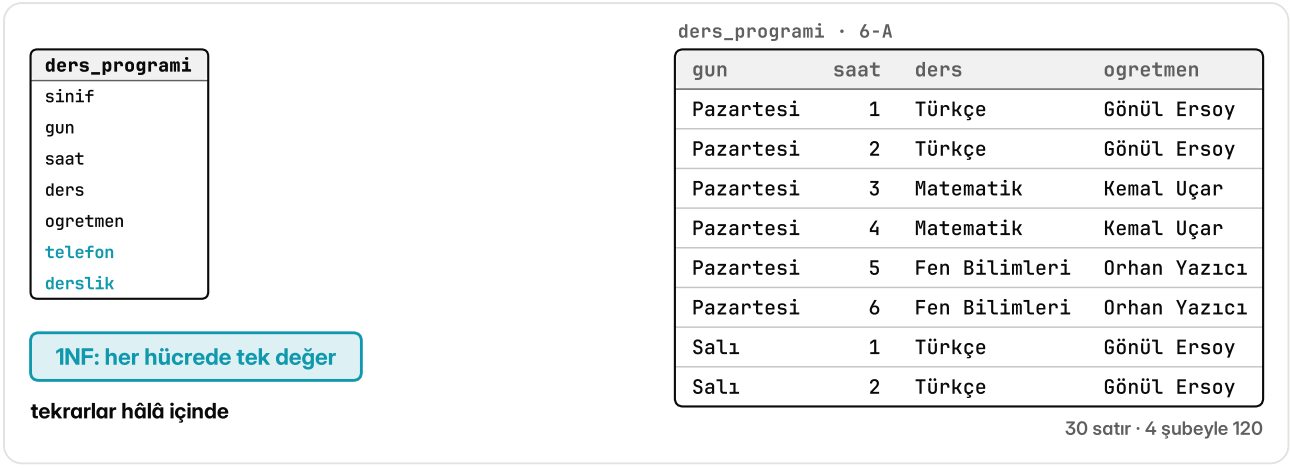
Şekil 8.3 ders_programi ile ogretmen tablosu; Gönül Ersoy'un numarası tek satırda: 24 satır yerine 1 satır.

Duvardaki program

videoda 1:04:59

Okulun duvarındaki ders programı bir ızgaradır: satırlarda günler, sütunlarda saatler, her kutuda ders ve öğretmen. Bu bilgi veritabanına girerken hangi tablolara, nereden bölünür?

İlişkisel bir tabloda her hücrede tek değer olur ve aynı bilgi sütun sütun tekrar etmez. Izgarada ise bir kutuda iki değer var, saatler de bilgi taşıyan sütun başlıkları olmuş. Bu yüzden ızgara açılır: her ders saati bir satır olur, gün ve saat kendi sütunlarına, ders ve öğretmen ayrı sütunlara geçer. 6-A'nın haftası 30 satır, okulun 4 şubesiyle 120. Her hücrede tek değer kuralına **birinci normal form** (1NF) denir. Tablo artık bu formda, ama tekrarlar hâlâ içinde.



Şekil 8.4 Izgaradan satırlara: ders_programi şeması, 6-A'nın ilk satırları ve 1NF etiketi.

Nereden bölüneceğini okulun dört kuralı söyler:

```
sinif, gun, saat → ders
sinif, ders → ogretmen
ogretmen → telefon
sinif → derslik
```

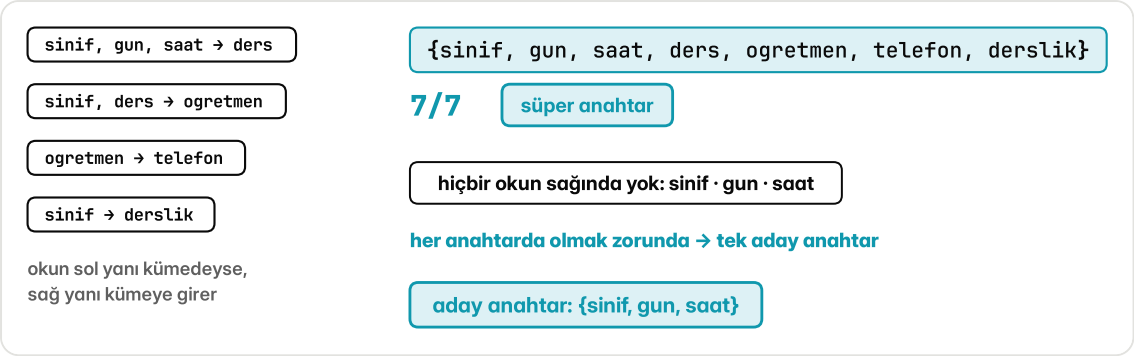
Ders tek başına öğretmeni belirlemez: matematiğin iki öğretmeni var, Kemal Uçar 6. sınıflara, Filiz Tekin 7. sınıflara giriyor.

Verilen kurallardan yenileri türetilebilir; bunun için üç **Armstrong kuralı** yeter (W. W. Armstrong, 1974). Yansıma: bir sütun kümesi kendi parçasını belirler (`sinif, gun → sinif`). Artırma: okun iki yanına aynı sütun eklenince ok bozulmaz (`ogretmen, gun → telefon, gun`). Geçişme: oklar art arda bağlanır; `sinif, ders → ogretmen` ve `ogretmen → telefon` olduğundan `sinif, ders → telefon`.

Satırı hangi sütunlar tanır? Bunu **nitelik kapanışı** (closure) bulur: bir sütun kümesiyle başlanır, sol yanı kümenin içinde kalan her okun sağ yanı kümeye eklenir, yeni bir şey eklenmeyene dek sürer.

```
{sinif, gun, saat}
+ ders       (sinif, gun, saat → ders)
+ ogretmen   (sinif, ders → ogretmen)
+ telefon    (ogretmen → telefon)
+ derslik    (sinif → derslik)
{sinif, gun, saat}⁺ = 7 sütunun 7'si
```

Bütün satırı belirleyen küme **süper anahtardır**. Birini çıkaralım: `{sinif, gun}⁺ = {sinif, gun, derslik}`, 7 sütunun 3'ü; saati ya da sınıfı çıkarınca da eksik kalır. Hiçbir parçası çıkarılamayan süper anahtara **aday anahtar** denir. Üstelik sınıf, gün ve saat hiçbir okun sağında yok; onları başka hiçbir şey getiremez, her anahtarda bulunmak zorundalar. Bu tablonun tek aday anahtarı `{sinif, gun, saat}`.



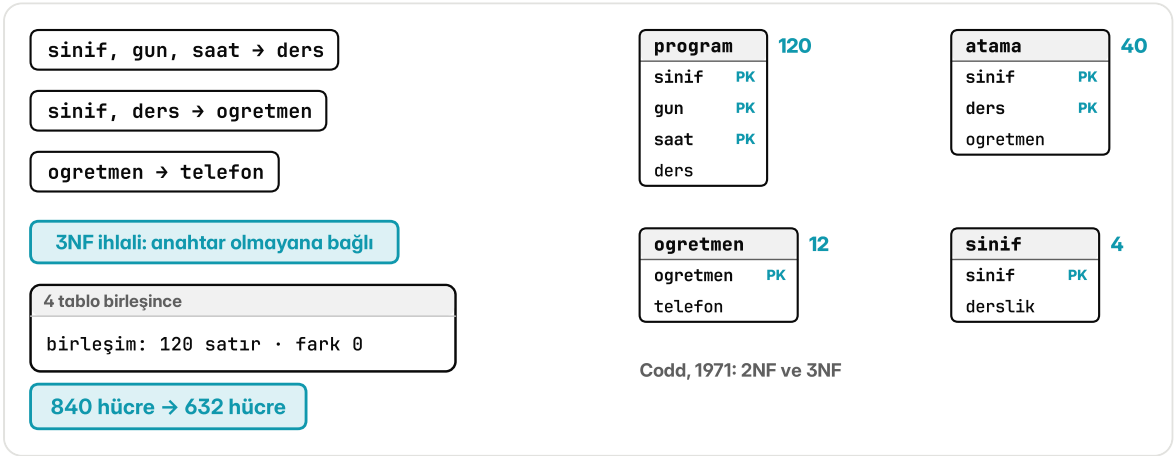
Şekil 8.5 Kapanış adım adım: {sinif, gun, saat} 7 sütunun hepsine ulaşıyor; tek aday anahtar.

İkinci normal form (2NF): anahtar olmayan hiçbir sütun anahtarın yalnız bir parçasına bağlı olmasın. Derslik buna uymuyor, yalnız sınıfa bağlı; bu yüzden her sınıfın dersliği 30 satırda tekrar ediyor. `sinif(sinif, derslik)` tablosu ayrılır: 4 satır.

Üçüncü normal form (3NF): anahtar olmayan bir sütun anahtara başka sütunlar üzerinden bağlanmasın. Telefon anahtara öğretmen üzerinden, öğretmen de ders üzerinden bağlı. İki tablo daha ayrılır:

tablo	sütunlar	satır
program	sinif, gun, saat, ders	120
atama	sinif, ders, ogretmen	40
ogretmen	ogretmen, telefon	12
sinif	sinif, derslik	4

Dört tablo yeniden birleştirilince 120 satırın hepsi aynen geri gelir; büyük tabloda 840 hücre vardı, dört tabloda 632. Kısacası tablo bağımlılıklarının okları boyunca bölünür: her ok, sol yanı anahtar olan kendi tablosuna gider.



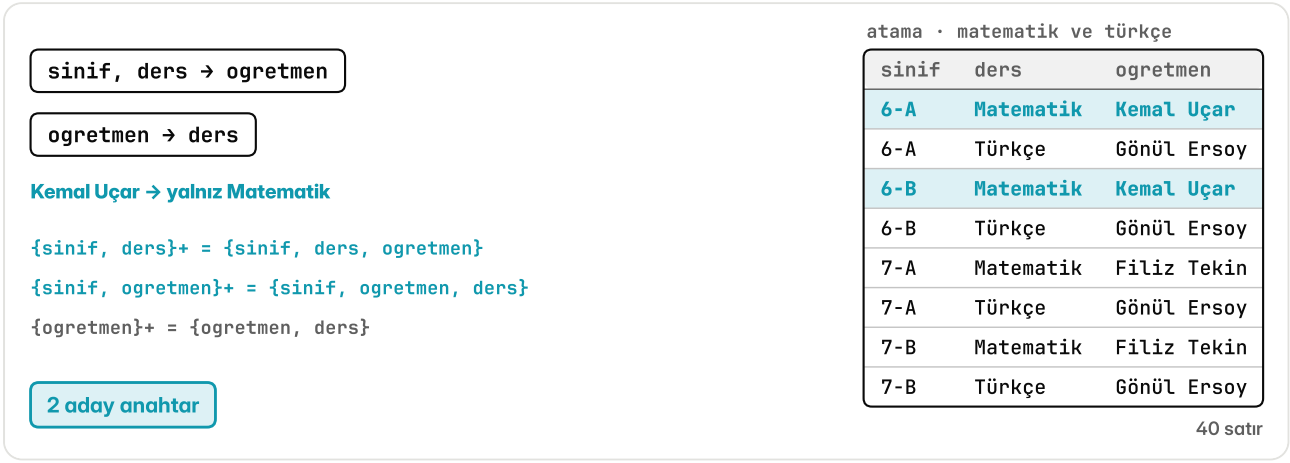
Şekil 8.6 3NF'ye bölme: program, atama, ogretmen ve sinif tabloları; birleşim 120 satır, 840 hücre yerine 632.

İtiraz etmeyen veritabanı

videoda 1:08:32

Okulun atama tablosunda her satır bir sınıf, bir ders ve o derse giren öğretmen: 4 şube, 40 satır. İki kural var: `sinif, ders → ogretmen` ve her öğretmenin tek branşı olduğu için `ogretmen → ders`. İkinci kural yüzünden tablonun iki aday anahtarı olur:

```
{sinif, ders}⁺ = {sinif, ders, ogretmen}
{sinif, ogretmen}⁺ = {sinif, ogretmen, ders}
{ogretmen}⁺ = {ogretmen, ders}
```



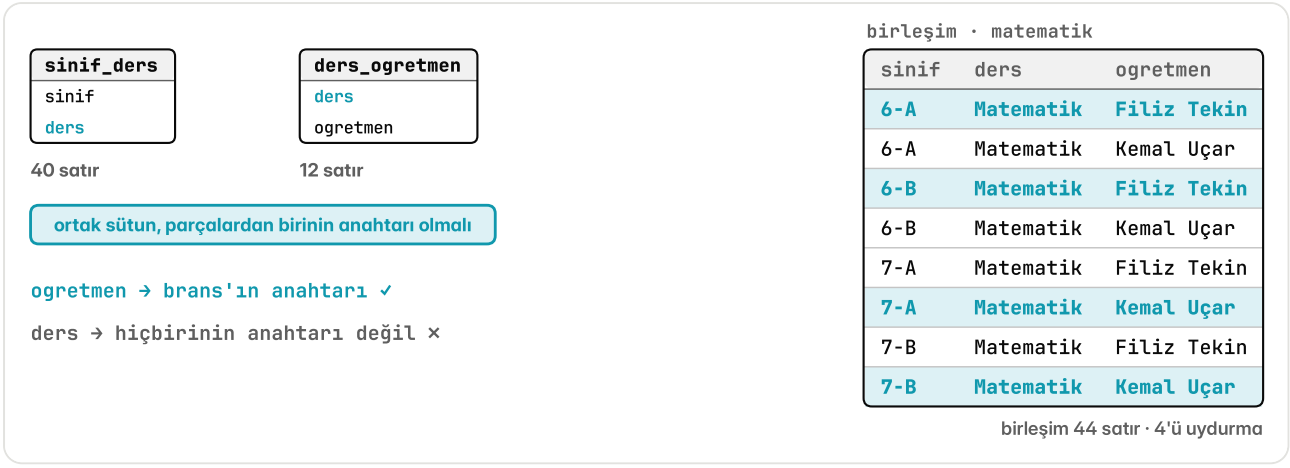
Şekil 8.7 Atama tablosu, iki kural ve kapanışlar: {sinif, ders} ve {sinif, ogretmen} iki aday anahtar.

Tablo üçüncü normal formdadır: ders bir anahtarın parçası olduğu için öğretmene bağlı olması 3NF'ye aykırı değil. Yine de tekrar var; Gönül Ersoy'un Türkçe okuttuğu bilgisi 4 ayrı satırda yazıyor. Bunu yakalayan daha sıkı kural **Boyce-Codd normal formudur** (BCNF; Boyce ve Codd, 1974): her okun sol yanı bir süper anahtar olmalı. `ogretmen → ders` buna uymuyor. Tablo ikiye bölünür: `brans(ogretmen, ders)` 12 satır, `sinif_ogretmen(sinif, ogretmen)` 40 satır.

Bölmenin ilk şartı **kayıpsız birleştirmedir** (lossless join): parçalar birleşince tablo aynen geri gelmeli. Branş ile sınıf-öğretmen tabloları öğretmen sütunundan birleşir ve 40 satırın hepsi geri gelir. Yanlış bir bölmeye bakalım: `(sinif, ders)` ve `(ders, ogretmen)`. Ortak sütun ders; matematiğin 2 öğretmeni olduğu için her şubenin matematiği 2 öğretmenle eşleşir ve birleştirme 40 değil 44 satır verir. Uydurma 4 satır:

sinif	ders	ogretmen
6-A	Matematik	Filiz Tekin
6-B	Matematik	Filiz Tekin
7-A	Matematik	Kemal Uçar
7-B	Matematik	Kemal Uçar

Satır eksilmedi, ama hangi satırın doğru olduğu bilgisi kayboldu. Kural: ortak sütun, parçalardan birinin anahtarı olmalı. Öğretmen branş tablosunun anahtarıdır; ders hiçbir parçanın anahtarı değil.



Şekil 8.8 Yanlış bölmenin birleşimi: matematiğin 8 satırından 4'ü uydurma; ortak sütun kuralı.

İkinci şart **bağımlılığın korunmasıdır** (dependency preservation): her kural tek bir tablonun içinde denetlenebilmeli. `ogretmen → ders` branş tablosunda duruyor, ama `sinif, ders → ogretmen` hiçbir tabloda değil. Bu yüzden şu ekleme kabul edilir (1 satır) ve birleşimde 6-A'nın matematiğinde iki öğretmen görünür:

```
INSERT INTO sinif_ogretmen
VALUES ('6-A', 'Filiz Tekin');
```

Bölünmemiş atama tablosu aynı eklemeyi reddederdi:

```
HATA: IntegrityError: UNIQUE constraint failed: atama.sinif, atama.ders
```

Her tablo iki şartı da bozmadan üçüncü normal forma getirilebilir. BCNF'de kayıpsızlık her zaman sağlanır, ama bir kural böyle kaybolabilir. O zaman ya 3NF'de durulur ya da kaybolan kural ayrıca denetlenir, örneğin bir tetikleyiciyle (trigger). Bölerken iki şey birlikte tartılır: tekrar ne kadar azalıyor ve hangi kurallar tabloların içinde kalıyor.

Tekrar nereden geliyor?

videoda 1:11:22

Bir tabloda hiçbir sütun ötekini belirlemese de aynı bilgi tekrar edebilir. Fen öğretmeni Orhan Yazıcı iki kulüp yürütüyor, bilim ve çevre; salı ve perşembe de nöbetçi. Okul bunları `etkinlik(ogretmen, kulup, nobet_gunu)` tablosunda tutuyor:

ogretmen	kulup	nobet_gunu
Orhan Yazıcı	Bilim	Perşembe
Orhan Yazıcı	Bilim	Salı
Orhan Yazıcı	Çevre	Perşembe
Orhan Yazıcı	Çevre	Salı

Bir kulübün birden çok öğretmeni (tabloda Çevre'nin 2), bir günün birden çok nöbetçisi (Salı'nın 2) olabildiği için hiçbir sütun ötekini belirlemez; anahtar üç sütunun hepsidir. Ama kulüpler ile nöbet günleri birbirinden bağımsız ve tek tabloda her kulüp her günle eşleşir: 2 × 2 = 4 satır. Cuma nöbeti eklenince 2 satır gerekir, her kulüp için bir. Öğretmen bir kulüp kümesini ve bir gün kümesini ayrı ayrı belirliyor; buna **çok değerli bağımlılık** (multivalued dependency) denir ve `ogretmen ↠ kulup` diye yazılır.



Şekil 8.9 Kulüp × nöbet günü ızgarası, Cuma'yla eklenen 2 satır ve ogretmen ↠ kulup, ogretmen ↠ nobet_gunu.

Dördüncü normal form (4NF; Fagin, 1977) bağımsız iki çok değerli bilgiyi iki tabloya ayırır: `kulup(ogretmen, kulup)` tablosunda Orhan Bey'in 2 kulübü, `nobet(ogretmen, nobet_gunu)` tablosunda 2 günü durur; cuma tek satırla eklenir. Her bilgi artık bir kez yazılır: bir kulübü bırakmak kulüp tablosundan tek satır siler, nöbet günlerine dokunmaz. İki tablo öğretmen sütunundan birleştirilince bütün eşleşmeler aynen geri gelir (iki öğretmenli tablonun tamamında 10 satır, fark 0).

Bazen de tekrar bilerek geri getirilir. Aynı okulun programı iki tabloda: `program` ve `atama`. Öğrenciler ve veliler okulun telefon uygulamasında günün programına her gün defalarca bakıyor ve her açılışta iki tablo birleştirilir:

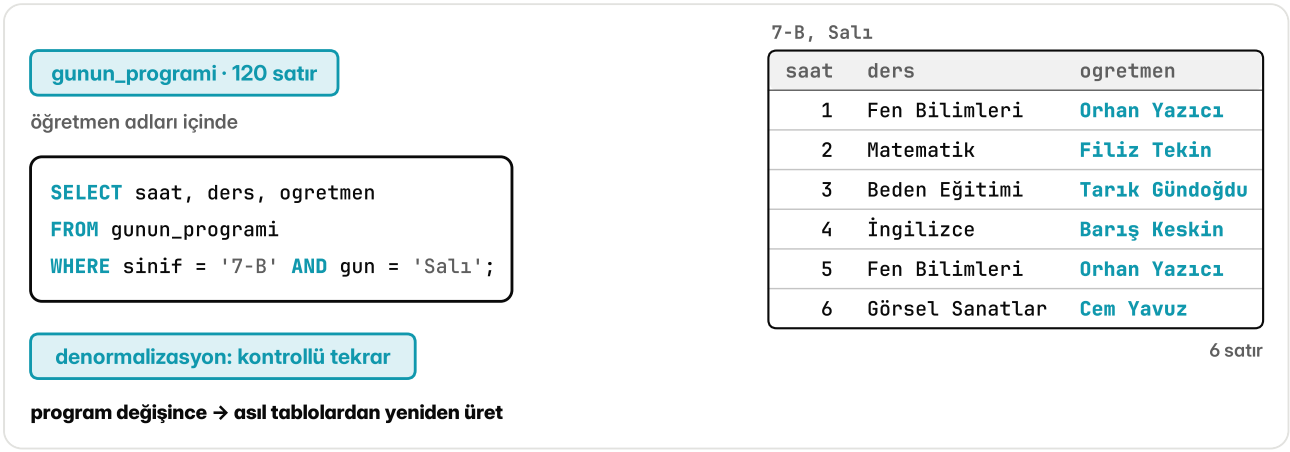
```
SELECT p.saat, p.ders, a.ogretmen
FROM program AS p
    JOIN atama AS a ON a.sinif = p.sinif
        AND a.ders = p.ders
WHERE p.sinif = '7-B' AND p.gun = 'Salı';
```

saat	ders	ogretmen
1	Fen Bilimleri	Orhan Yazıcı
2	Matematik	Filiz Tekin
3	Beden Eğitimi	Tarık Gündoğdu
4	İngilizce	Barış Keskin
5	Fen Bilimleri	Orhan Yazıcı
6	Görsel Sanatlar	Cem Yavuz

Oysa program bir dönemde bir iki kez değişir. Bu yüzden birleştirilmiş hâli, öğretmen adları içinde, hazır bir tabloda tutulabilir (`gunun_programi`, 120 satır); aynı 6 satır tek tablodan okunur:

```
SELECT saat, ders, ogretmen
FROM gunun_programi
WHERE sinif = '7-B' AND gun = 'Salı';
```

Buna **denormalizasyon** denir. Şartı kontrol: hazır tabloya elle dokunulmaz, program her değiştiğinde asıl tablolardan yeniden üretilir. Sorun tekrarın kendisi değil, elle düzeltilen kontrolsüz tekrardır.



Şekil 8.10 Hazır tablo gunun_programi: 7-B'nin salısı birleştirmesiz, tek tablodan; denormalizasyon kontrollü tekrar.

Özet

- Aynı bilgi birçok satırda tekrar edince güncelleme, ekleme ve silme anomalileri çıkar.
- X → Y fonksiyonel bağımlılığında X'i aynı satırların Y'si de aynıdır; veri bir bağımlılığı çürütebilir, kanıtlayamaz.
- Kapanış, bir sütun kümesinin belirlediği her şeyi bulur; bütün satırı belirleyen en küçük küme aday anahtardır.
- 1NF her hücrede tek değer, 2NF anahtarın parçasına bağlılık yok, 3NF anahtar olmayana bağlılık yok, BCNF her okun sol yanı süper anahtar.
- İyi bir bölme kayıpsızdır ve kuralları korur; BCNF'de bir kural kaybolabilir, 3NF'de kaybolmaz.
- 4NF bağımsız çok değerli bilgileri ayırır; denormalizasyon okuma hızı için bilerek ve kontrollü tekrardır.

Sorular

1 **ders_programi tablosunun anahtarı sınıf, gün ve saattir. Derse girmeyen yeni bir rehber öğretmenin telefonu bu tabloya kaydedilemiyor. Bu hangi derttir?**

A Güncelleme anomalisi
B Silme anomalisi
C Ekleme anomalisi
D Kayıplı birleştirme

2 **Okulun ders programında matematiğe iki öğretmen giriyor: 6. sınıflara Kemal Uçar, 7. sınıflara Filiz Tekin. Aşağıdaki bağımlılıklardan hangisi bu veriyle çürütülür?**

A ogretmen → telefon
B ders → ogretmen
C sinif, ders → ogretmen
D sinif → derslik

3 **Kurallar: sinif, gun, saat → ders; sinif, ders → ogretmen; ogretmen → telefon; sinif → derslik. $\{sinif, ders\}^+$ kapanışı kaç sütundan oluşur? Sayıyı yaz.**

Cevap:

4 **R(A, B, C, D, E) tablosunda bağımlılıklar A → B, B → C ve A, D → E. Tablonun aday anahtarını yaz.**

Cevap:

5 **Her hücresinde tek değer olan R(A, B, C, D) tablosunda bağımlılıklar A, B → C ve B → D; tek aday anahtar {A, B}. Tablo en çok hangi normal formdadır?**

A 2NF (3NF değil)
B 3NF (BCNF değil)
C BCNF
D 1NF (2NF değil)

6 **R(A, B, C) tablosunda bağımlılıklar A → B ve B → C; anahtar {A}. Tablo en çok hangi normal formdadır?**

A 2NF (3NF değil)
B 1NF (2NF değil)
C 3NF (BCNF değil)
D BCNF

7 **R(A, B, C) tablosunda bağımlılıklar A, B → C ve C → B. Aday anahtarlar {A, B} ve {A, C}. Tablo en çok hangi normal formdadır?**

A 1NF (2NF değil)
B 2NF (3NF değil)
C 3NF (BCNF değil)
D BCNF

8 **Bir öğretmenin 3 kulübü ve 2 nöbet günü var; ikisi birbirinden bağımsız ve tek tabloda (ogretmen, kulup, nobet_gunu) her kulüp her günle eşleşerek tutuluyor. Öğretmene bir nöbet günü daha eklenince tabloya kaç satır eklenir?**

Cevap:

VERİTABANI SİSTEMLERİ

9 Yarı yapısal veri: XML ve JSON

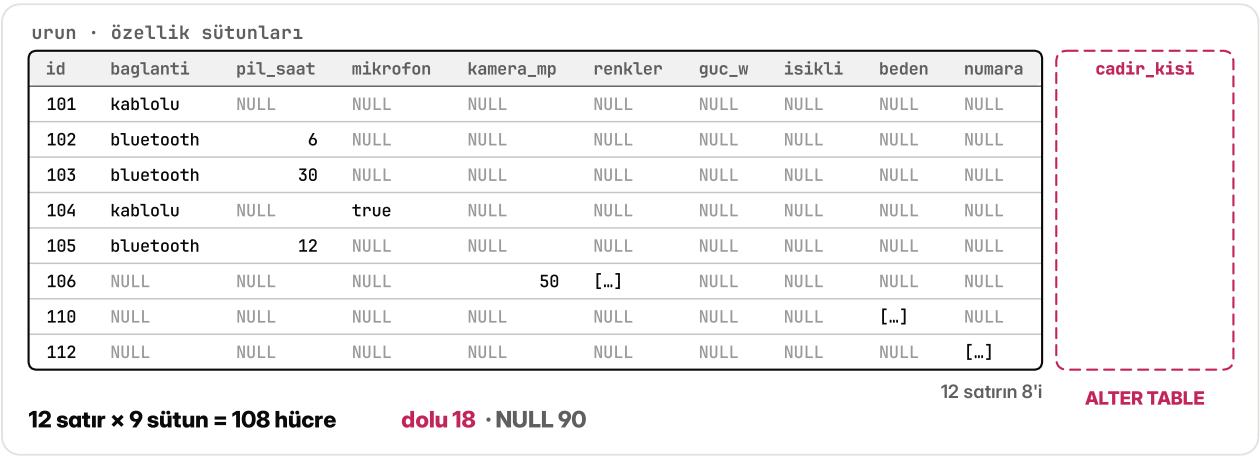
Bu bölümde özellikleri birbirine hiç benzemeyen ürünleri aynı tabloda tutmanın yolunu arıyoruz. Önce veriyi yapısına göre üçe ayırıp kendini tarif eden verinin en yaygın iki yazılışını, XML'i ve JSON'ı tanıyacağız. Sonra mağaza veritabanının ürün tablosuna bir JSON sütunu ekleyip içini önce SQLite'ta, sonra PostgreSQL'de sorgulayacağız.

Kamera ile beden

videoda 1:13:13

Bir çevrimiçi mağazada telefon için kameranın kaç megapiksel olduğu sorulur, gömlek için beden, ayakkabı için numara, kulaklık için de pilin kaç saat gittiği. Mağaza veritabanının ürün tablosunda (`urun`) ise her ürünün aynı 5 sütunu var: `id`, `ad`, `kategori_id`, `fiyat`, `stok`. Özellikleri birbirine benzemeyen ürünler aynı tabloya nasıl sığar?

İlk akla gelen, her özelliğe bir sütun açmaktır. 12 ürünün özelliklerini toplayınca 9 ayrı özellik çıkıyor: bağlantı türü, pil süresi, mikrofon, kamera, renkler, şarj gücü, tuş ışığı, beden ve ayakkabı numarası. 9 sütun açarsak 12 satırda 108 hücre olur; bunların yalnız 18'i dolu, 90'ı NULL. Üstelik gömleğin 4 bedeni (S, M, L, XL) var ve tek bir hücreye 4 değer sığmaz. Mağaza yarın çadır satmaya başlasa, her yeni özellik için tablonun kendisini ALTER TABLE ile değiştirmek gerekir.



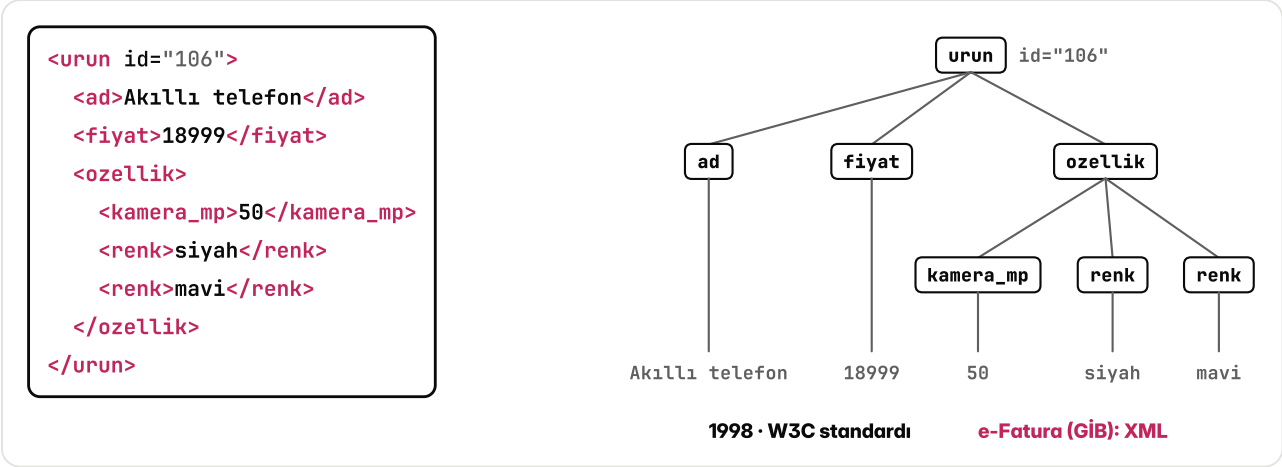
Şekil 9.1 Her özelliğe bir sütun: 12 satırın 8'i görünüyor; 108 hücrenin yalnız 18'i dolu, yeni bir ürün türü yeni bir sütun ister.

Veri bu açıdan üçe ayrılır. **Yapısal veri**de önce kalıp çizilir, her satır aynı sütunlara uyar; ürün tablosu böyledir. **Yapısız veri**de, örneğin bir müşteri yorumunda ya da ürünün fotoğrafında, hangi parçanın ne olduğunu söyleyen bir işaret yoktur. İkisinin arasında **yarı yapısal veri** (semi-structured data) durur: önceden çizilmiş tek bir kalıp yoktur, her kayıt kendi etiketlerini yanında taşır. Değerin yanında ne olduğu da yazar ("kamera, 50"); bu yüzden ona **kendini tarif eden veri** denir.

XML'de her değer bir açılış ve bir kapanış etiketinin arasına yazılır. Telefonu yazalım:

```
<urun id="106">
  <ad>Akıllı telefon</ad>
  <fiyat>18999</fiyat>
  <ozellik>
    <kamera_mp>50</kamera_mp>
    <renk>siyah</renk>
    <renk>mavi</renk>
  </ozellik>
</urun>
```

Ürün etiketinin içinde ad, fiyat ve özellik etiketleri var; özelliğin içinde de kamera ve iki renk. Etiketler iç içe girdikçe kayıt bir ağaca dönüşür: kökte `urun`, altında dalları, uçlarda değerler. Aynı etiket iki kez kullanılabilir; telefonun iki `renk` etiketi var. Ürünün numarası ise etiketin kendisine **nitelik** (attribute) olarak yazıldı: `id="106"`. XML 1998'de W3C standardı oldu; Türkiye'deki e-fatura da bugün XML ile yazılır.



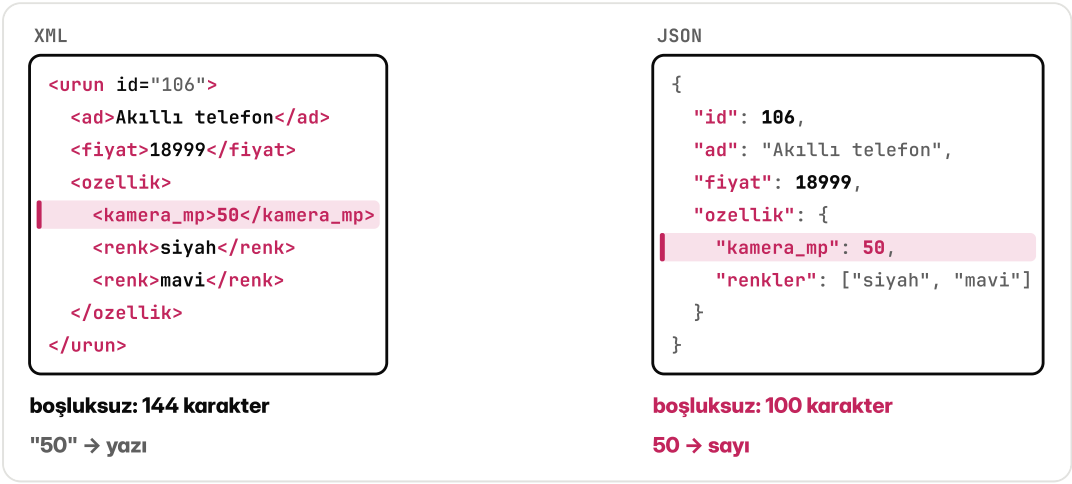
Şekil 9.2 Telefonun XML'i ve ağacı: kökte urun ve id niteliği, dallarda etiketler, uçlarda değerler.

Hangi etiketin nerede bulunabileceği istenirse ayrı bir şema belgesinde yazılır; bilinen iki biçimi belge tipi tanımı (DTD) ve XML şemasıdır (XSD). Şema zorunlu değildir, belge onsuz da okunur. Ağacın içinden değer almak için, bir dosyanın klasör yolu gibi bir yol yazılır (XPath): `/urun/ozellik/kamera_mp` 50'yi getirir. Başa konan çift eğik çizgi "nerede olursa olsun" demektir: `//renk` siyah ile maviyi birlikte getirir.

JSON üç yapı taşı kullanır. Nesne süslü parantezle açılır; içinde ad ve değer çiftleri durur, **anahtar** denen ad ile değerin arasında iki nokta vardır. Dizi köşeli parantezle yazılan sıralı bir listedir. Değer bir metin, bir sayı, `true` ya da `false`, ya da boş anlamına gelen `null` olabilir. Nesnenin içine nesne konabilir; JSON da bir ağaçtır. Aynı telefon:

```
{
  "id": 106,
  "ad": "Akıllı telefon",
  "fiyat": 18999,
  "ozellik": {
    "kamera_mp": 50,
    "renkler": ["siyah", "mavi"]
  }
}
```

Boşluklar atılınca aynı kayıt XML'de 144, JSON'da 100 karakter tutar; JSON'da kapanış etiketleri yoktur. Bir fark daha var: XML'de 50 yalnız bir yazıdır, JSON'da bir sayı.



Şekil 9.3 Aynı kayıt yan yana: boşluksuz XML 144, JSON 100 karakter; XML'de "50" yazı, JSON'da 50 sayı.

Kısa olması ve sayıyı yazıdan ayırması JSON'ı web servislerinin ortak dili yaptı. Telefondaki bir uygulama sunucuya kargonun nerede olduğunu sorduğunda cevap çoğu zaman JSON gelir; programların birbirine açtığı bu kapılara **API** denir. Açılıştaki sorunun cevabı da buradadır: ürünler aynı kalıba sığmak zorunda değildir. Her ürün kendi özelliklerini adlarıyla birlikte yanında taşıyınca telefonun kamerası ile gömleğin bedeni boş sütunlar açmadan yan yana durabilir.

Pil süresi nerede?

videoda 1:16:24

Mağaza, bluetooth'lu ürünlerini pili en uzun gidenden başlayarak listelemek istiyor. Ama ürün tablosunda pil süresi diye bir sütun yok; gömleğin pili, telefonun bedeni olmaz. Tabloya tek bir sütun ekleyelim ve her ürün kendi JSON'ını oraya yazsın:

```
ALTER TABLE urun
    ADD COLUMN ozellik TEXT;
```

id	ad	fiyat	stok	ozellik
102	Kablosuz kulak içi	499	25	{"baglanti": "bluetooth", "pil_saat": 6}
106	Akıllı telefon	18999	4	{"kamera_mp": 50, "renkler": ["siyah", "mavi"]}
110	Pamuklu gömlek	899	20	{"beden": ["S", "M", "L", "XL"]}

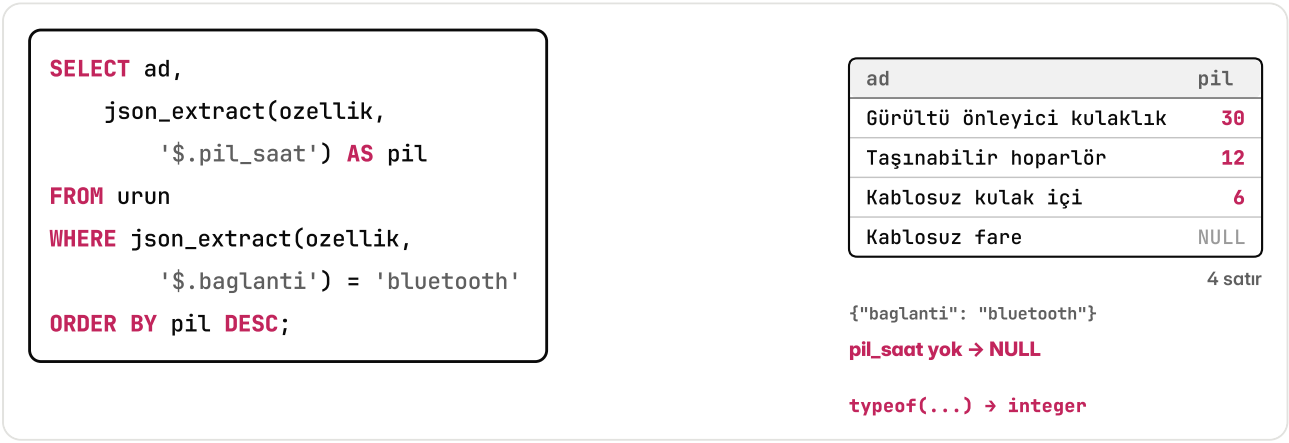
(12 satırın 3'ü.) Fiyat ve stok her üründe olduğu için kendi sütunlarında kalır; ürüne göre değişen ayrıntı JSON'a gider. SQLite'ta bu sütun düz bir metindir.

Metnin içinden değer almak için `json_extract` fonksiyonu var. İki şey ister: sütun ve yol. Yol `$` ile başlar, kaydın kendisi demektir; noktadan sonra anahtarın adı gelir: `'$.pil_saat'`.

```
SELECT ad,
    json_extract(ozellik,
        '$.pil_saat') AS pil
FROM urun
WHERE json_extract(ozellik,
        '$.baglanti') = 'bluetooth'
ORDER BY pil DESC;
```

ad	pil
Gürültü önleyici kulaklık	30
Taşınabilir hoparlör	12
Kablosuz kulak içi	6
Kablosuz fare	NULL

Kablosuz farenin pili NULL, çünkü onun JSON'ında (`{"baglanti": "bluetooth"}`) pil anahtarı hiç yok. Olmayan anahtar hata vermez, NULL döner. Değerin türü de korunur: 30 bir yazı olarak değil, tam sayı olarak gelir (`typeof` integer der); sıralama da sayı sırasıdır.



Şekil 9.4 json_extract ile bluetooth'lu ürünler pil süresine göre: 30, 12, 6 ve pil anahtarı olmayan farede NULL.

Dizinin elemanına sırasıyla ulaşılır ve sayma 0'dan başlar: `'$.renkler[0]'` telefonun ilk rengini, siyahı verir. Dizinin içinde arama yapmak için dizi geçici bir tabloya açılır; `json_each` her elemanı bir satır yapar:

```
SELECT urun.ad
FROM urun,
    json_each(urun.ozellik, '$.beden')
WHERE json_each.value = 'XL';
```

Gömleğin beden dizisi S, M, L, XL diye 4 satıra, montunki M, L diye 2 satıra açılır. XL yalnız gömlekte vardır; sonuç Pamuklu gömlek, 1 satır.

PostgreSQL'in bunun için ayrı bir tipi var: **JSONB**. SQLite'ın metin sütunu, kapanış parantezi unutulmuş `'{baglanti: kablolu'` metnini sessizce kabul etti (1 satır etkilendi); hata ancak içinden okuyan sorguda çıktı:

```
SELECT ad, json_extract(ozellik, '$.baglanti')
FROM urun
WHERE id = 101;
HATA: OperationalError: malformed JSON
```

SQLite'ta sütuna `CHECK (json_valid(ozellik))` kısıtı konursa o da bozuk metni girişte reddeder. JSONB ise bunu kendiliğinden yapar (PostgreSQL 19):

```
UPDATE urun SET ozellik = '{baglanti: kablolu' WHERE id = 101;
ERROR:  invalid input syntax for type json
```

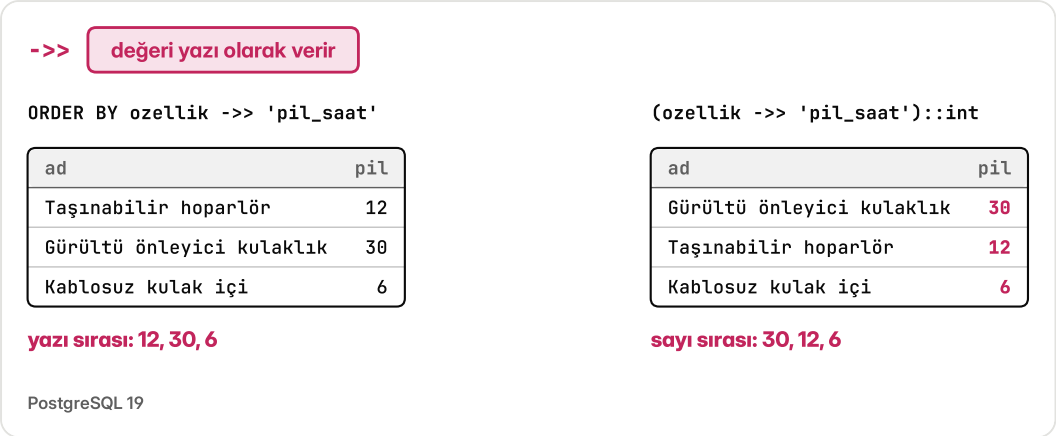
Çünkü JSONB değeri metin olarak değil, çözümlenmiş ikili bir biçimde saklar: boşlukları atar, anahtarları kendi sırasına dizer, aynı anahtar iki kez yazıldıysa sonuncuyu tutar.

```
json:  {"pil_saat": 6,  "baglanti": "bluetooth", "pil_saat": 8}
jsonb: {"baglanti": "bluetooth", "pil_saat": 8}
```

İçinden değer almak için ok işareti `->>` kullanılır; "içeriyor mu" sorusunun da kendi işareti var: `@>`. Beden dizisinde XL olanları bu işaret tek satırla bulur:

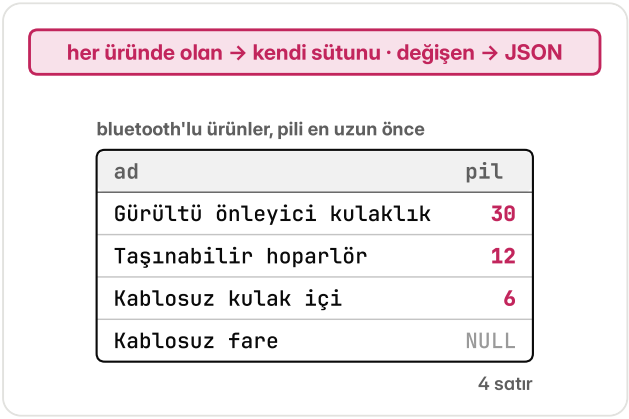
```
SELECT ad
FROM urun
WHERE ozellik @> '{"beden": ["XL"]}';
```

Sonuç yine Pamuklu gömlek. Bu biçimin içine bakan bir indeks de kurulabilir. Yalnız bir tuzak var: `->>` değeri yazı olarak verir. Pil sürelerini `ORDER BY ozellik ->> 'pil_saat'` ile sıralayınca 12, 30, 6 çıkar; yazıda 1, 3'ten ve 6'dan önce gelir. Sayı gibi sıralamak için değer önce tam sayıya çevrilir: `(ozellik ->> 'pil_saat')::int` ile sıra 30, 12, 6 olur.



Şekil 9.5 PostgreSQL'de ->> tuzağı: yazı sırası 12, 30, 6; tam sayıya çevirince 30, 12, 6.

JSON sütunu ne zaman iyi fikirdir? Özellikler üründen ürüne değişiyorsa, seyrek sorgulanıyorsa ya da veri başka bir servisten JSON olarak geliyor ve olduğu gibi saklanacaksa. Ne zaman kötü? Bir bilgi her sorguda süzülüyor, sıralanıyor ya da başka bir tabloyla birleştiriliyorsa, onun yeri kendi sütunudur. Tablonun kuralları JSON'ın içine işlemez: NOT NULL, UNIQUE ve yabancı anahtar (foreign key) gibi kısıtlar oradaki değerleri korumaz; pil süresinin yerine `"çok"` yazısı bile girer. Yazım hatası da uyarı vermez: `'$.pil_saat'` yerine `'$.pil_sat'` yazan sorgu 4 satırın 4'üne de NULL döndürür. Her hücrede tek değer kuralı da esner; gömleğin 4 bedeni tek hücrede durur. Kural basit: her üründe olan bilgi kendi sütununda, üründen ürüne değişen ayrıntı JSON'da. Pil süresi sütunu olmayan tablo böylece pil süresine göre sıralanır.



Şekil 9.6 Ortak bilgi sütunda, değişen ayrıntı JSON'da: bluetooth'lu ürünler pili en uzun giden önce.

Özet

- Yarı yapısal veride kalıp önceden çizilmez; her kayıt değerlerini adlarıyla birlikte taşır, yani kendini tarif eder.
- XML değerleri açılış ve kapanış etiketleri arasına yazar, kayıt bir ağaçtır; içinden değer XPath yoluyla alınır (`//renk`).
- JSON nesne, dizi ve değerle yazar; XML'den kısadır ve sayıyı yazıdan ayırır, web servislerinin ortak dilidir.
- SQLite'ta `json_extract(sütun, '$.anahtar')` içinden okur; olmayan anahtar NULL döner, `json_each` diziyi satırlara açar.
- PostgreSQL'in JSONB tipi bozuk JSON'ı girişte reddeder; `->>` içinden yazı olarak okur, `@>` içeriyor mu diye sorar.
- Her üründe olan bilgi kendi sütununda, üründen ürüne değişen ayrıntı JSON'da durur; tablonun kısıtları JSON'ın içini korumaz.

Sorular

1 **Aşağıdakilerden hangisi yarı yapısal veriye örnektir?**

A Ürün tablosunun bir satırı

B Bir müşterinin yazdığı yorum metni

C {"kamera_mp": 50, "renkler": ["siyah", "mavi"]} kaydı

D Ürünün fotoğrafı

2 **Telefonun XML kaydında özellik etiketinin içinde <kamera_mp>50</kamera_mp>, <renk>siyah</renk> ve <renk>mavi</renk> var. //renk yolu ne getirir?**

A Yalnız siyah

B Siyah ve mavi

C Yalnız mavi

D Hiçbir şey; yol /urun ile başlamalı

3 **Mağaza veritabanında ürün tablosuna ozellik sütunu eklendi ve 12 ürünün JSON'ı yazıldı. SELECT COUNT(*) FROM urun WHERE json_extract(ozellik, '$.baglanti') = 'kablolu'; sorgusu ne döndürür? Yalnız sayıyı yaz.**

Cevap:

4 SELECT ad, json_extract(ozellik, '$.renkler[1]') FROM urun WHERE id IN (106, 110); sorgusunda pamuklu gömleğin (110, {"beden": ["S", "M", "L", "XL"]}) satırında ne döner?

A M

B Hata: renkler anahtarı yok

C Boş metin ''

D NULL

5 SELECT urun.ad FROM urun, json_each(urun.ozellik, '$.beden') WHERE json_each.value = 'L'; sorgusu kaç satır döndürür? Yalnız sayıyı yaz.

Cevap: _____

6 PostgreSQL'de pil süreleri 6, 30 ve 12 olan üç ürün ORDER BY ozellik ->> 'pil_saat' DESC ile sıralanıyor. Sonuçta pil süreleri hangi sırayla gelir?

A 30, 12, 6

B 6, 30, 12

C 12, 30, 6

D 6, 12, 30

7 PostgreSQL'de SELECT '{"beden": ["S"], "beden": ["M", "L"]}'::jsonb; ne döndürür?

A {"beden": ["M", "L"]}

B {"beden": ["S"]}

C {"beden": ["S", "M", "L"]}

D Hata: aynı anahtar iki kez yazılamaz

8 SELECT COUNT(json_extract(ozellik, '$.Pil_saat')) FROM urun; sorgusu ne döndürür? (Pil süresi 3 üründe pil_saat anahtarıyla yazılı.) Yalnız sayıyı yaz.

Cevap: _____

VERİTABANI SİSTEMLERİ

10 Fiziksel tasarım ve indeksler

Bu bölümde verinin diskte nasıl durduğuna iniyoruz. Satırlar sayfalarda saklanır ve bir sorgunun bedeli okuduğu sayfa sayısıyla ölçülür. Tek satırlık bir CREATE INDEX komutunun bir sorguyu nasıl yüzlerce kat hızlandırdığını, indeksin içindeki B+ ağacını, hash indeksini ve tablo büyüdükçe onun nasıl büyüdüğünü, son olarak belleğe sığmayan bir tablonun nasıl sıralandığını görüyoruz. Örnekler SQLite'ta ve PostgreSQL'de çalıştırıldı; süreler bu dersin hazırlandığı bilgisayarda ölçüldü.

Bir milyon sipariş

videoda 1:19:32

Bir çevrimiçi mağazanın sipariş tablosu büyümüş: 1.000.000 satır. Her satır bir sipariş; numarası, müşterinin numarası, tarihi ve tutarı. 50.000 müşterinin her birinin 20 siparişi var. 27318 numaralı müşteri "siparişlerim" sayfasını açınca şu sorgu çalışır:

```
SELECT id, tarih, tutar
FROM siparis
WHERE musteri_id = 27318;
```

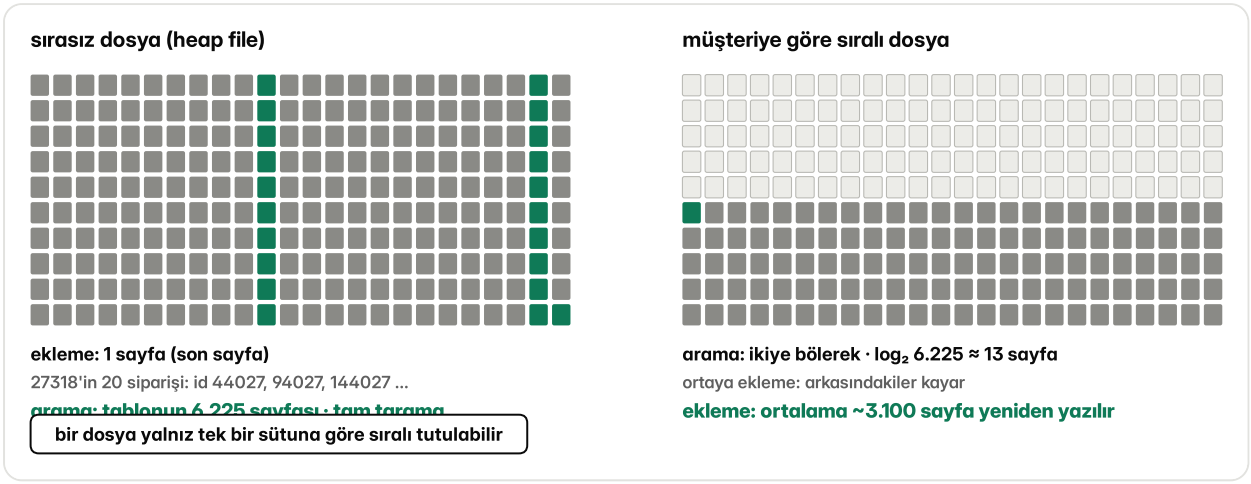
Sonuç 20 satır (20 satırın ilk 8'i):

id	tarih	tutar
44027	2026-01-05	213
94027	2026-07-04	813
144027	2026-12-31	1413
194027	2025-06-29	2013
244027	2025-12-26	2613
294027	2026-06-24	3213
344027	2026-12-21	3813
394027	2025-06-19	4413

Veritabanı satırları diskte tek tek değil, **sayfa** sayfa saklar. SQLite'ta bir sayfa 4 KB'tır (PostgreSQL'de 8 KB) ve bu tablonun bir sayfasına ortalama 161 sipariş sığar. Disk de her okumada bir sayfanın tamamını getirir; bu yüzden bir sorgunun bedeli okuduğu sayfa sayısıyla ölçülür. Veritabanı dosyası 6.226 sayfadır: 6.225'i tablonun kendisi, biri şema sayfası. Tablonun 6.208 sayfası satırları taşıyan yapraklardır.

Siparişler geldikleri sırayla yazılıyor. Böyle **sırasız bir dosyada** (heap file) eklemek ucuzdur: yeni sipariş son sayfaya yazılır. Ama bir müşterinin siparişleri her yere dağılmıştır; 27318'in siparişleri 50.000'er satır arayla durur. Onları bulmak için tablonun 6.225 sayfasının hepsi okunur. Buna **tam**

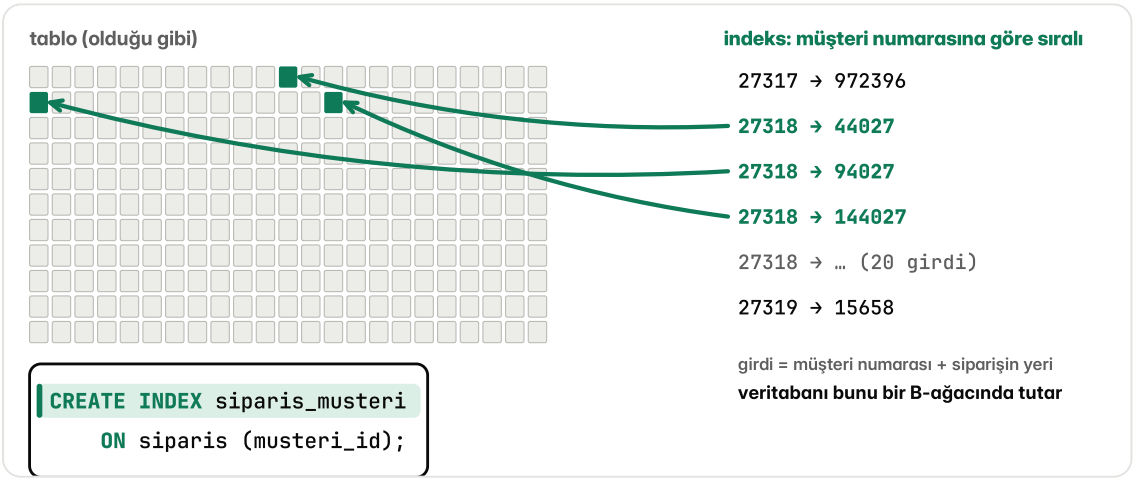
tarama denir. Dosya müşteri numarasına göre sıralı tutulsaydı aranan yer ikiye bölerek bulunurdu: $\log_2 6.225 \approx 12{,}6$, yani 13 sayfa. Ama bu kez ekleme pahalanır: ortaya giren her sipariş için arkasındaki satırlar kayar, ortalama 3.000 sayfa kadar yeniden yazılır. Üstelik bir dosya yalnız tek bir sütuna göre sıralı tutulabilir.



Şekil 10.1 Solda sırasız dosyada 27318'in siparişleri dağınık ve tam tarama bütün sayfaları okur; sağda sıralı dosyada ikiye bölerek 13 sayfa, ama ortaya ekleme pahalı.

Kalın bir kitapta bir terimi ararken sondaki dizine bakarız: terimler alfabe sırasıyla, yanlarında sayfa numaraları; kitabın kendisi değişmez. **İndeks** de budur. Tablo olduğu gibi kalır, yanına müşteri numarasına göre sıralı ayrı bir yapı kurulur; her girdide bir müşteri numarası ve o siparişin tablodaki yeri durur. Artık sıralı tutulan bütün tablo değil bu yapıdır ve veritabanı onu bir B-ağacında tutar. Tek satırlık komut:

```
CREATE INDEX siparis_musteri
    ON siparis (musteri_id);
```



Şekil 10.2 Tablo yerinde kalır; yanındaki indeks müşteri numaralarını sıralı tutar ve 27318'in girdileri tablodaki sayfalarını gösterir.

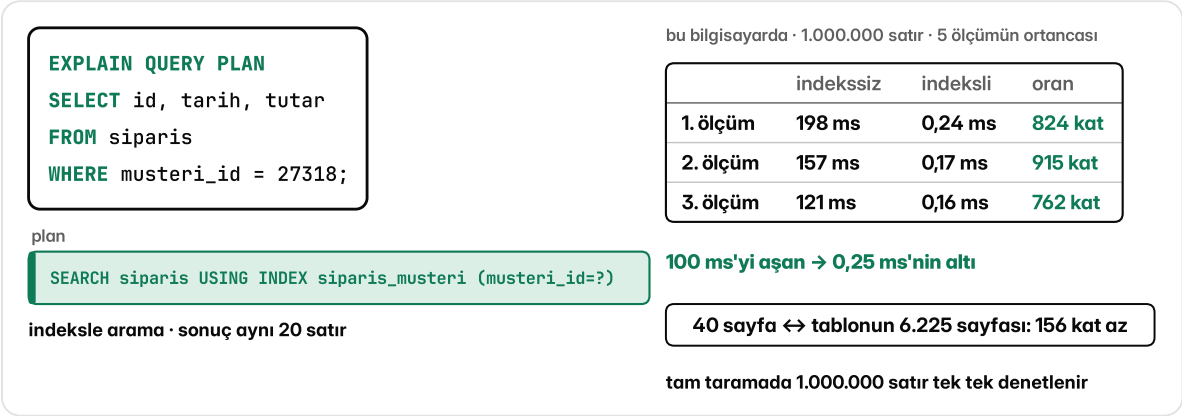
Veritabanının bir sorguyu nasıl çalıştıracağını görmek için sorgunun başına EXPLAIN QUERY PLAN yazılır (SQLite'ta). İndeksten önce plan tek satırdır, `SCAN siparis`: tam tarama. Komuttan sonra plan değişir:

```
SEARCH siparis USING INDEX siparis_musteri (musteri_id=?)
```

Sorgu aynı 20 satırı getirir, ama yolu başkadır. İndeks üç katlı bir ağaçtır (kökte 1, ortada 11, en altta 2.765 sayfa). Kökten yaprağa inilir, yaprakta 20 girdi yan yana bulunur, sonra tabloda o siparişlerin durduğu sayfalara gidilir. Dokunulan sayfa: indeksten 3, tablodan 37, toplam 40; tam taramadaki 6.225 sayfanın 156'da biri. Bu bilgisayarda yapılan üç ölçüm (her biri 5 denemenin ortancası):

ölçüm	indekssiz	indeksli	oran
1.	198 ms	0,24 ms	824 kat
2.	157 ms	0,17 ms	915 kat
3.	121 ms	0,16 ms	762 kat

Süre sayfa sayısından da çok kısaldı, çünkü tam taramada bir milyon satırın her biri tek tek denetlenir.



Şekil 10.3 Planda indeksle arama ve bu bilgisayarda ölçülen süreler: sorgu 762 ile 915 kat arası hızlandı.

İndeksin bir bedeli var. Her yeni siparişte veritabanı indeksin de doğru yerine bir girdi yazar; silmede ve güncellemede indeksi de düzeltir. Müşteri numaraları karışık geldiği için her ekleme indeksin başka bir sayfasına dokunur. Bu bilgisayarda 100.000 yeni sipariş eklemek indekssiz yarım saniyenin altında sürdü (0,40-0,43 sn); tek indeksle birkaç kat uzadı (1,8-4,4 sn), müşteri, tarih ve tutar için üç indeksle 26 ile 32 kat arası (10,4-13,7 sn). İndeks yer de tutar: bu tek indeks dosyayı 25,5 MB'tan 36,9 MB'a, %45 büyüttü. Bu yüzden indeks her sütuna değil, sık aranan sütuna kurulur.

Bir aralık

videoda 1:22:35

Mağazanın ürün tablosunda 12 ürün var, her birinin bir stoğu. Stoğu 10 ile 25 arasında kalan ürünler isteniyor:

```
SELECT ad, stok
FROM urun
WHERE stok BETWEEN 10 AND 25
ORDER BY stok;
```

İndeks tek bir değeri birkaç sayfa okumayla bulur. Bir aralıktaki bütün değerleri nasıl bulur? Veritabanlarının indeksi çoğunlukla bir **B+ ağacıdır** (B+ tree). Stok sütununun ağacını küçük tutalım, her yaprağa 3 anahtar sığsın. 12 stok değeri küçükten büyüğe dört yaprağa dizilir: 4, 6, 7 · 9, 11, 12 · 15, 20, 25 · 30, 40, 60. Kökte yalnız yol gösteren 9, 15 ve 30 durur; her biri sağındaki yaprağın en küçük anahtarı. Ağacın iki kuralı var: bütün anahtarlar yapraklardadır, iç düğümler yalnız yön gösterir; ve her yaprak sağındaki yaprağa bağlıdır.

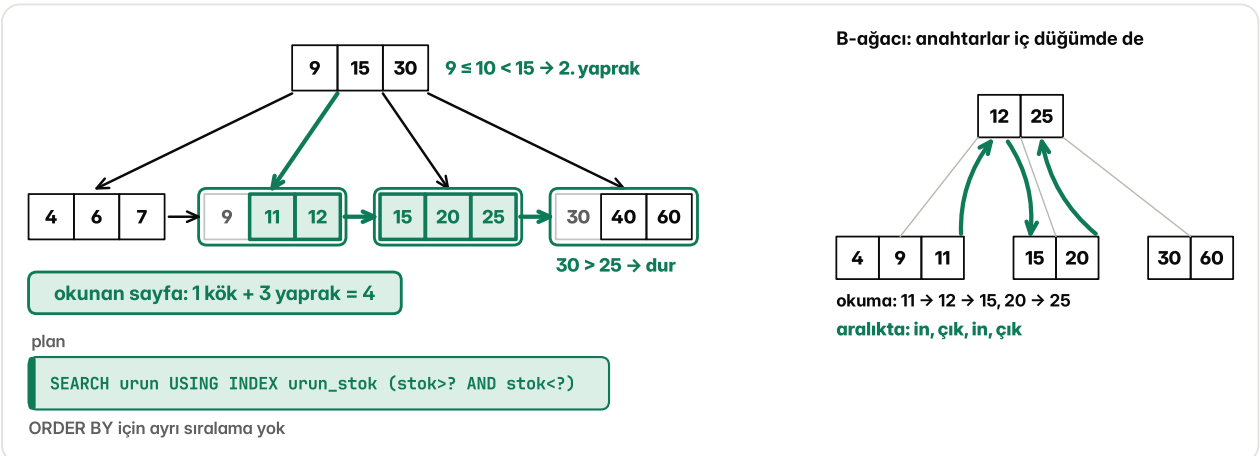
Aralığın başı 10. Kökte 10, 9 ile 15 arasına düşer; ikinci yaprağa inilir. 9 atlanır, 11 ve 12 alınır. Ağaca geri çıkılmaz, yaprağın bağıyla sağdakine geçilir: 15, 20, 25 alınır. Sonraki yaprakta 30, 25'i aşar ve yürüyüş biter. Dört sayfa okundu (kök ve üç yaprak), beş ürün bulundu:

ad	stok
Koşu ayakkabısı	11
Oyuncu kulaklığı	12
Mekanik klavye	15
Pamuklu gömlek	20
Kablosuz kulak içi	25

Sonuç kendiliğinden stoğa göre sıralı gelir; ORDER BY için ayrıca sıralamak gerekmez. Stok sütununa urun_stok indeksi kurulunca SQLite'ın planı:

```
SEARCH urun USING INDEX urun_stok (stok>? AND stok<?)
```

B-ağacında anahtarlar iç düğümlere de dağıldığı için aralıkta ağaçta inip çıkmak gerekir; B+ ağacı bir kez iner, sonra yalnız yürür.

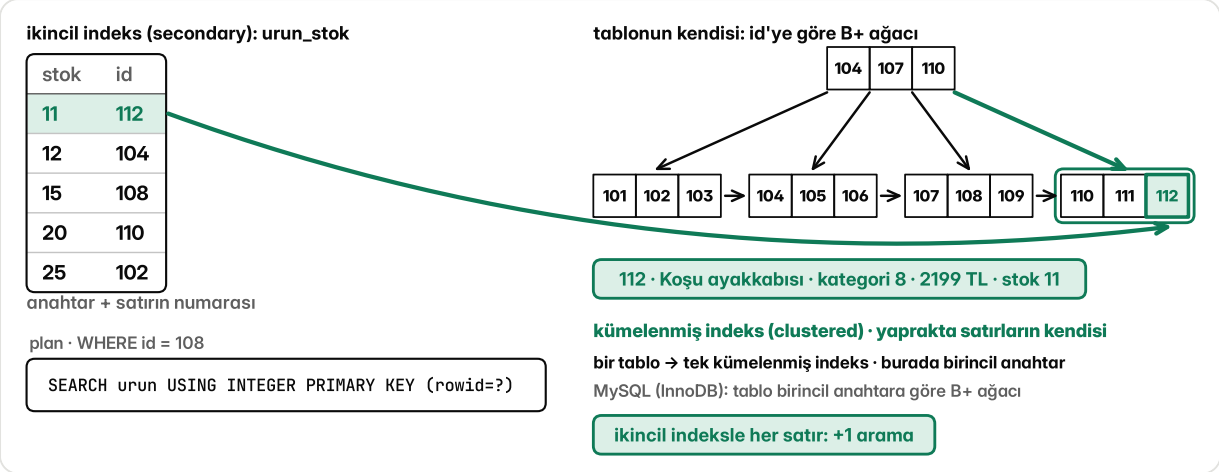


Şekil 10.4 Kökten ikinci yaprağa tek iniş, yaprak zinciriyle üç yapraklık yürüyüş; yanda B-ağacında aynı aralık için in-çık-in yolu.

Yaprakta stoğun yanında yalnız ürünün numarası durur; adı öğrenmek için tabloya gidilir. Tablonun kendisi de numaraya göre sıralı bir B+ ağacıdır ve onun yapraklarında satırların tamamı durur. Satırları kendi sırasıyla tutan bu indekse **kümelenmiş indeks** (clustered index) denir. Satırlar tek bir sıraya dizilebildiği için bir tablonun tek kümelenmiş indeksi olur; bu tabloda o, birincil anahtardır. Numarayla aramada SQLite'ın planı:

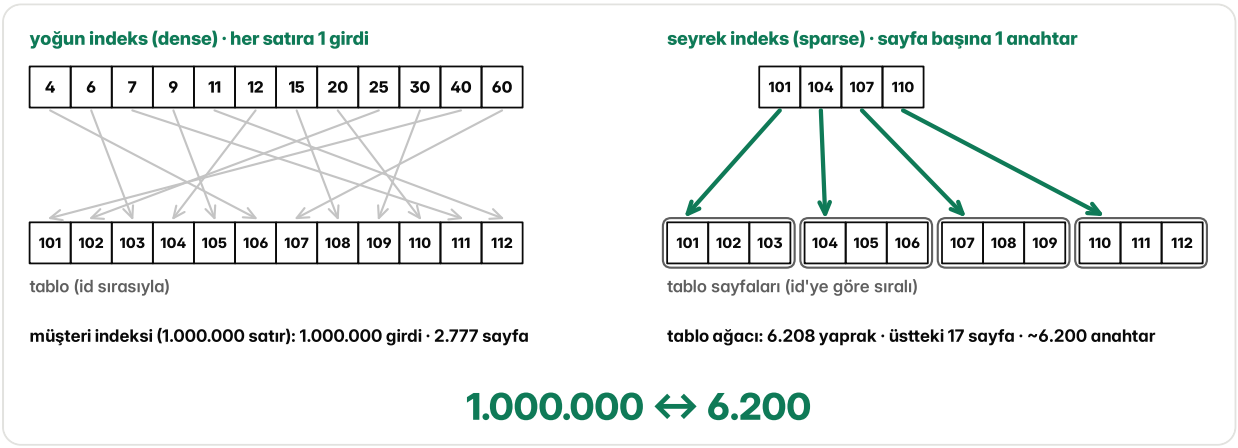
```
WHERE id = 108:  SEARCH urun USING INTEGER PRIMARY KEY (rowid=?)
```

MySQL de (InnoDB) tabloları birincil anahtara göre böyle tutar. Stok indeksi gibi öbürleri **ikincil indekstir** (secondary index); onunla bulunan her satır için kümelenmiş ağaçta bir arama daha yapılır.



Şekil 10.5 İkincil indeksteki 11 → 112 girdisi kümelenmiş ağaçta 112'nin yaprağına gider; satırın tamamı orada.

İkincil indeks her satır için bir girdi tutar, çünkü tablo stoğa göre dizili değildir. Buna **yoğun indeks** (dense index) denir. Kümelenmiş ağaçta ise sayfanın içi zaten sıralı olduğu için her yaprak sayfasına tek anahtar yeter: **seyrek indeks** (sparse index). Bir milyon siparişlik tabloda müşteri numarasının yoğun indeksi 1.000.000 girdi, 2.777 sayfadır; tablonun kendi ağacında 6.208 yaprağı gösteren anahtarlar, üstteki 17 sayfada, 6.200 kadardır.



Şekil 10.6 Solda her satıra bir girdi (yoğun), sağda her sayfaya bir anahtar (seyrek).

Bir indeks birden çok sütunu birlikte de sıralayabilir. Telefon rehberindeki gibi: önce soyada, aynı soyadında ada göre. Sipariş tablosuna müşteri numarası ve tarih için **bileşik bir indeks** kurulsun:

```
CREATE INDEX siparis_musteri_tarih
    ON siparis (musteri_id, tarih);
```

27318 numaralı müşterinin 2026'nın ikinci yarısındaki siparişleri sorulunca plan indeksi iki sütunla birden kullanır:

```
SELECT id, tarih, tutar
FROM siparis
WHERE musteri_id = 27318
    AND tarih >= '2026-07-01';

SEARCH siparis USING INDEX siparis_musteri_tarih (musteri_id=? AND tarih>?)
```

Müşterinin girdileri yan yana ve tarihe göre sıralıdır; 6 sipariş tek parça okunur ve tarih sırasıyla gelir. Ama yalnız tarih sorulursa, örneğin 1 Temmuz 2026'nın 1.370 siparişi (`WHERE tarih = '2026-07-01'`), plan tam taramaya döner: `SCAN siparis`. Rehberde yalnız ada göre aramak gibi, o tarih bütün müşterilerin arasına dağılmıştır. Bileşik indeks ancak ilk sütunu sorguda geçerse işe yarar; sütun sırası bu yüzden önemlidir.

Tek okuma

videoda 1:25:44

B+ ağacı bir müşterinin siparişlerine 3-4 sayfa okumayla iner. Yalnız eşitlik aranıyorsa bu iş tek okumaya inebilir mi; ve tablo büyüdükçe o yapı baştan kurulmadan nasıl büyür?

Hash fonksiyonu anahtarı doğrudan bir göz numarasına çevirir. Diskte her göz bir sayfadır ve adı **kova**dır (bucket). 27318'in kovası hesaplanır ve okunur; ağaçta inmek yok, bir iki okuma yeter. Ama hash sırayı dağıtır: 27318 ile 27319 uzak kovalara düşer, bu yüzden **hash indeksi** aralığa yaramaz. PostgreSQL'de eşitlik sorgusu hash indeksini kullanır, BETWEEN ile sorulan üç müşterilik aralıkta plan tam taramaya döner:

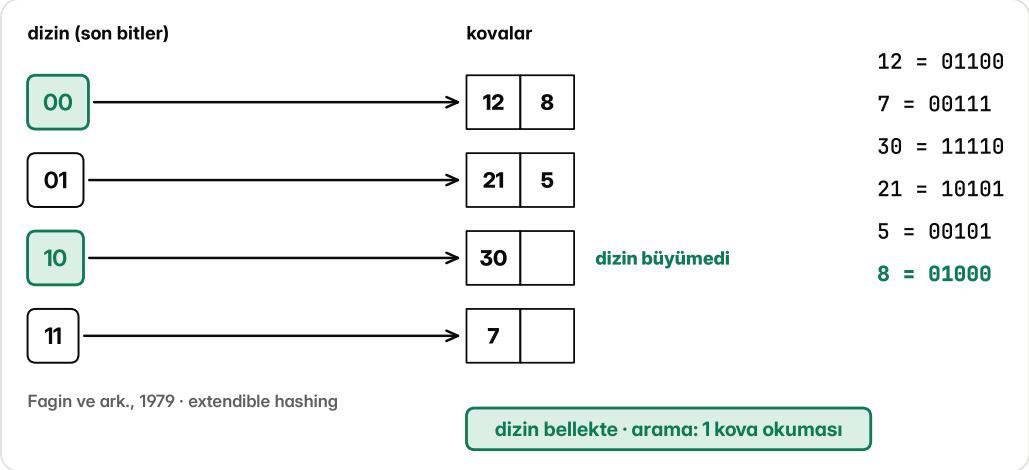
```
CREATE INDEX siparis_musteri_hash ON siparis USING hash (musteri_id);

WHERE musteri_id = 27318
->  Bitmap Index Scan on siparis_musteri_hash

WHERE musteri_id BETWEEN 27318 AND 27320
Seq Scan on siparis
```

Tablo büyüdükçe kovalar dolar, taşan anahtarlar taşma sayfalarına yazılır ve tek okuma ikiye, üçe çıkar. Her şeyi yeniden dağıtmak diskte milyonlarca girdiyi baştan yazmak demektir; **dinamik hash** tabloyu parça parça büyütür. Küçük örnek:

```
her kova en çok 2 anahtar · anahtarlar sırayla 12, 7, 30, 21, 5, 8, 11
ikilik: 12 = 01100, 7 = 00111, 30 = 11110, 21 = 10101, 5 = 00101, 8 = 01000
```

Genişletilebilir hash (extendible hashing; Fagin ve arkadaşları, 1979) kovaların önüne, anahtarın ikilik yazılışının son bitlerine bakan bir dizin koyar. Kova dolunca yalnız o kova bölünür: onu tek göz gösteriyorsa önce dizin ikiye katlanır, iki göz paylaşıyorsa dizin büyümez. Dizin bellekte durur; arama tek kova okumasıdır.

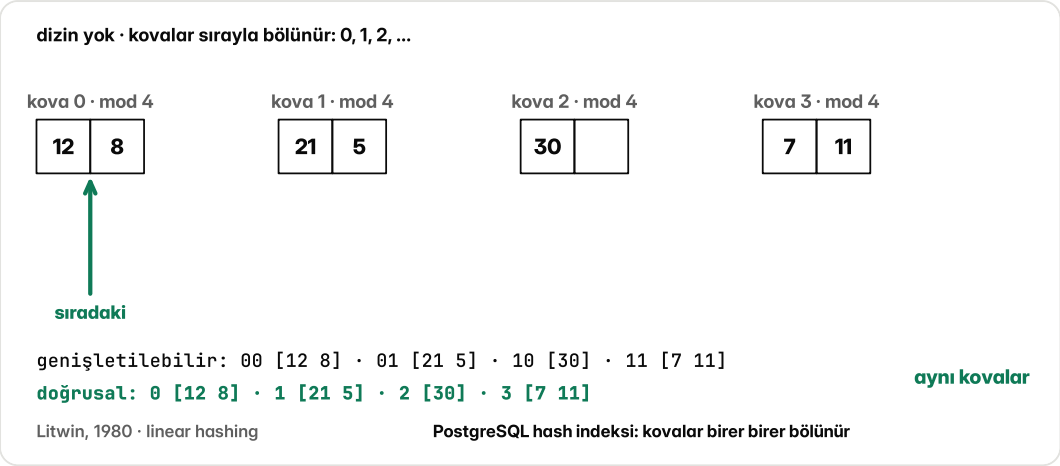
```
1 bit:    0 → [12 30]   1 → [7 21]
5 gelir:  dizin 2 bit; 00, 10 → [12 30] · 01 → [21 5] · 11 → [7]
8 gelir:  dizin büyümez; 00 → [12 8] · 10 → [30]
11 gelir: 11 → [7 11]
```



Şekil 10.7 Genişletilebilir hash: iki bitlik dizin dört kovayı gösterir; 8 geldiğinde dizin büyümeden yalnız kova bölündü.

Doğrusal hash (linear hashing; Litwin, 1980) dizin kullanmaz; kovalar sırayla bölünür: 0, 1, 2 ... Bir kova taşınca taşan anahtar taşma sayfasına yazılır, ama bölünen, sırası gelen kovadır; taşan kova kendi sırası gelince bölünür. Sonuç genişletilebilir hash'teki dört kovanın aynısı, ama dizin yok. PostgreSQL'in hash indeksi de kovalarını böyle birer birer böler.

```
mod 2:    0 → [12 30]   1 → [7 21]
5 gelir:  1 dolu, 5 taşma sayfasına; sırası gelen 0 bölünür (mod 4): 0 → [12] · 2 → [30]
8 gelir:  0 bölünmüş, 8 mod 4 = 0: 0 → [12 8]
11 gelir: 1 yine taşar, sıra onun: 1 → [21 5] · 3 → [7 11], taşma sayfası kalkar
```



Şekil 10.8 Doğrusal hash: kovalar sırayla bölünür, son durum genişletilebilir hash'inkiyle aynı dört kovadır.

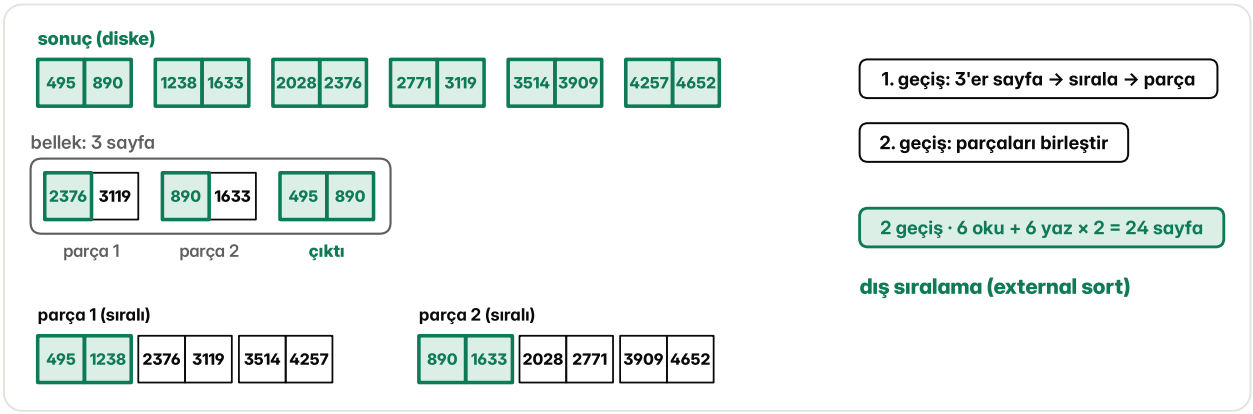
Hash sırayı dağıtır. Oysa ORDER BY ya da yeni kurulan bir B+ ağacı sıralı veri ister. Tablo belleğe sığmıyorsa ne olur? Altı sayfalık tabloda her sayfada iki sipariş tutarı, bellekte üç sayfalık yer olsun:

```
tablo:    [3119 1238] [4257 2376] [495 3514] [1633 4652] [2771 890] [3909 2028]
```

Birinci geçişte üçer sayfa belleğe alınır, sıralanır ve sıralı bir **parça** (run) olarak diske yazılır. İki parça çıkar:

```
parça 1:  [495 1238] [2376 3119] [3514 4257]
parça 2:  [890 1633] [2028 2771] [3909 4652]
```

İkinci geçişte iki parçanın ilk sayfaları belleğe alınır, üçüncü sayfa çıktıya ayrılır; başlardaki tutarlardan küçüğü çıktıya geçer: önce 495, sonra 890. Dolan çıktı sayfası diske yazılır, biten giriş sayfasının yerine parçanın sonraki sayfası gelir. Tablo iki geçişte sıralandı, her geçişte bir kez okunup yazıldı: 24 sayfa okuma ve yazma. Buna **dış sıralama** (external sort) denir.



Şekil 10.9 Dış sıralama: iki sıralı parça, bellekte iki giriş ve bir çıkış sayfasıyla birleşir.

B sayfalık bellek bir kerede B − 1 parçayı birleştirir. 8 KB'lık sayfalarla 100 MB'lık bellek 12.800 sayfadır: 10 GB'lık tablo ilk geçişte 103 parçaya bölünür ve hepsi tek seferde birleşir, yine iki geçiş. Bellek 1 MB'a inerse aynı tablo üç geçiş ister.

PostgreSQL bunu kendiliğinden yapar: bir milyon sipariş tutara göre sıralanırken çalışma belleği (work_mem) 4 MB olunca dış sıralamayı seçti, 256 MB'ta iş bellekte bitti:

```
work_mem = 4MB:    Sort Method: external merge  Disk: 17704kB
work_mem = 256MB:  Sort Method: quicksort  Memory: 48014kB
```

Özet

- Satırlar diskte sayfalarda durur; bir sorgunun bedeli okuduğu sayfa sayısıdır (bir milyon siparişlik tablo 6.225 sayfa).
- Sırasız dosyaya eklemek ucuz, aramak tam tarama; sıralı dosyada arama 13 sayfa, ama ekleme pahalı.
- İndeks, tabloya dokunmadan yanına kurulan sıralı yapıdır: aynı sorgu 6.225 yerine 40 sayfa okudu, bu bilgisayarda 762-915 kat hızlandı; bedeli her yazmada güncellenmesi ve yer tutmasıdır.
- B+ ağacında anahtarlar yalnız yapraklarda, yapraklar birbirine bağlı; aralık sorgusu bir iniş ve bir yürüyüştür. Bileşik indeks ilk sütunu sorguda geçince işe yarar.
- Hash indeksi eşitlikte kovaya doğrudan gider, aralıkta işe yaramaz; genişletilebilir hash dizini ikiye katlar, doğrusal hash kovaları sırayla böler.
- Belleğe sığmayan tablo parça parça sıralanıp birleştirilir; 100 MB bellekle 10 GB'lık tablo iki geçişte sıralanır.

Sorular

1 **Siparişler geldikleri sırayla sırasız bir dosyaya yazılıyor. Aynı tablo müşteri numarasına göre sıralı tutulsaydı ne değişirdi?**

A Hem arama hem ekleme ucuzlardı.

B Bir müşteriyi aramak ucuzlardı, ama ortaya giren her sipariş için arkasındaki satırlar kaydığından ekleme pahalanırdı.

C Ekleme ucuzlardı, arama yine tam tarama olurdu.

D Tablo aynı anda tarihe göre de sıralı olurdu.

2 **Mağaza veritabanında stok sütununa urun_stok indeksi kuruldu. SELECT ad, stok FROM urun WHERE stok BETWEEN 5 AND 12 ORDER BY stok; kaç satır döndürür? (Stoklar: 4, 6, 7, 9, 11, 12, 15, 20, 25, 30, 40, 60.)**

A 4

B 6

C 5

D 3

3 **Stok değerlerinin B+ ağacında yapraklar 4 6 7 · 9 11 12 · 15 20 25 · 30 40 60, kökte 9, 15, 30 var ve yapraklar birbirine bağlı. stok BETWEEN 5 AND 12 aralığı için kökle birlikte kaç sayfa okunur? Sayıyı yaz.**

Cevap:

4 **Bir milyon siparişlik tabloda bileşik indeks var: CREATE INDEX siparis_musteri_tarih ON siparis (musteri_id, tarih); Başka indeks yok. SQLite hangi sorguda bu indeksle arar (SEARCH), hangilerinde tam tarama (SCAN) yapar?**

A WHERE tarih = '2026-07-01' indeksle arar.

B WHERE tarih >= '2026-07-01' AND tutar > 4000 indeksle arar.

C WHERE tutar = 813 indeksle arar.

D WHERE musteri_id = 31415 indeksle arar; ötekilerde tam tarama yapar.

5 **PostgreSQL'de musteri_id sütununa hash indeksi kuruldu (USING hash). Aşağıdaki sorgulardan hangisinde plan bu indeksi kullanır?**

A WHERE musteri_id > 49990

B WHERE musteri_id = 31415

C ORDER BY musteri_id LIMIT 5

D WHERE musteri_id BETWEEN 31415 AND 31417

6 **Genişletilebilir hash'te dizin son 2 bite bakıyor (4 göz). Yalnız tek bir gözün gösterdiği bir kova doluyken bu kovaya yeni bir anahtar geliyor. Dizin kaç gözlü olur? Sayıyı yaz.**

Cevap:

7 **Doğrusal hash'te dört kova var (0-3) ve sıradaki bölünecek kova 0. Bir anahtar 2 numaralı kovayı taşırıyor. Ne olur?**

A 2 numaralı kova hemen bölünür.

B Bütün kovalar ikiye katlanır ve anahtarlar yeniden dağıtılır.

C Taşan anahtar taşma sayfasına yazılır ve sırası gelen 0 numaralı kova bölünür.

D Önce bir dizin kurulur, sonra 2 numaralı kova bölünür.

8 **20 sayfalık bir tablo, 3 sayfalık bellekle dış sıralamayla sıralanıyor. İlk geçişte 3'er sayfalık sıralı parçalar çıkıyor, sonraki her geçişte parçalar ikişer ikişer birleştiriliyor (iki giriş sayfası, bir çıkış sayfası). Toplam kaç geçiş gerekir? Sayıyı yaz.**

Cevap:

VERİTABANI SİSTEMLERİ

11 Sorgu işleme ve iyileştirme

SQL'de ne istediğimizi yazarız, nasıl bulunacağını yazmayız. Bu bölümde veritabanının en hızlı yolu nasıl seçtiğine bakıyoruz. Önce bir sorgunun cevaba giden yolculuğunu, tam tarama ile indeks arasındaki seçimi ve seçiciliği göreceğiz. Sonra iki tabloyu birleştirmenin üç algoritmasını okunan sayfa sayısıyla karşılaştıracağız. En sonda birleştirme sırasını, süzmenin öne alınmasını ve planlayıcının tahminlerini besleyen istatistikleri ele alacağız. Örnekler mağaza veritabanından; PostgreSQL örneklerinde mağaza 40.008 müşteriye ve 400.010 siparişe büyütüldü (ek satırlar yalnız bu bölümün dosyalarında). Planlar SQLite 3.50 ve PostgreSQL 19'da gerçekten çalıştırıldı.

Ne, nasıl değil

videoda 1:29:08

Büyümüş mağazada 1 Kasım 2026'da kaydolan müşterileri isteyelim:

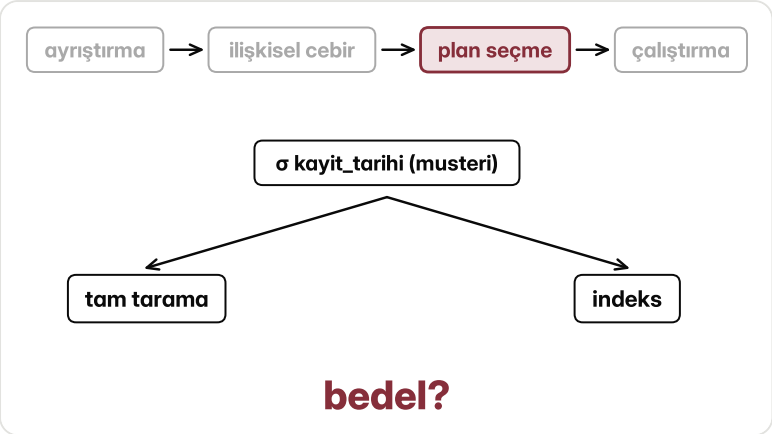
```
SELECT *
FROM musteri
WHERE kayit_tarihi = '2026-11-01';
```

Sorgu yalnız ne istendiğini söyler: müşteri tablosundan, kayıt tarihi 1 Kasım olan satırlar. Hangi sayfalara, hangi sırayla bakılacağı yazmaz. Oysa aynı cevaba birden çok yoldan gidilebilir ve yollar arasındaki fark yüzlerce kat olabilir. Veritabanı en hızlı yolu nereden bilir?

Sorgu cevaba dört durakta ulaşır. İlki **ayrıştırma**: veritabanı metni okur, yazımı ve adları denetler. SELECT yerine SELEC yazılırsa ya da `sehir` sütunu `sehri` diye yazılırsa, tek satır okunmadan hata döner (SQLite):

```
SELEC ad FROM musteri;
HATA: OperationalError: near "SELEC": syntax error

SELECT ad
FROM musteri
WHERE sehri = 'Kars';
HATA: OperationalError: no such column: sehri
```

İkinci durakta sorgu ilişkisel cebire çevrilir: müşteri tablosuna bir seçme işlemi, σ kayit_tarihi = '2026-11-01' (musteri). Cebir ne yapılacağını söyler, nasıl yapılacağını hâlâ söylemez. Üçüncü durak **plan seçmek**: her işlemin birkaç yöntemi vardır; **planlayıcı** (query planner) her birinin bedelini tahmin eder ve en ucuzunu seçer. Dördüncü durakta plan çalışır, satırlar gelir. Bütün hüner üçüncü durakta.



Şekil 11.1 Dört durak: ayrıştırma, ilişkisel cebir, plan seçme ve çalıştırma; plan seçmede tam tarama ile indeks arasında karar verilir.

Bedel okunan sayfa sayısıyla ölçülür. Tablo diskte sayfa sayfa durur; 40.008 müşteri 334 sayfaya sığar. En pahalı iş bir sayfayı diskten okumaktır. Seçmenin iki yolu vardır. **Tam tarama**da 334 sayfanın hepsi okunur, her satır koşulla karşılaştırılır. İndeksle aramada ise kayıt tarihi sütunundaki indeks, aranan tarihin hangi satırlarda olduğunu söyler; önce indeks okunur, sonra yalnız o satırların bulunduğu sayfalara gidilir.

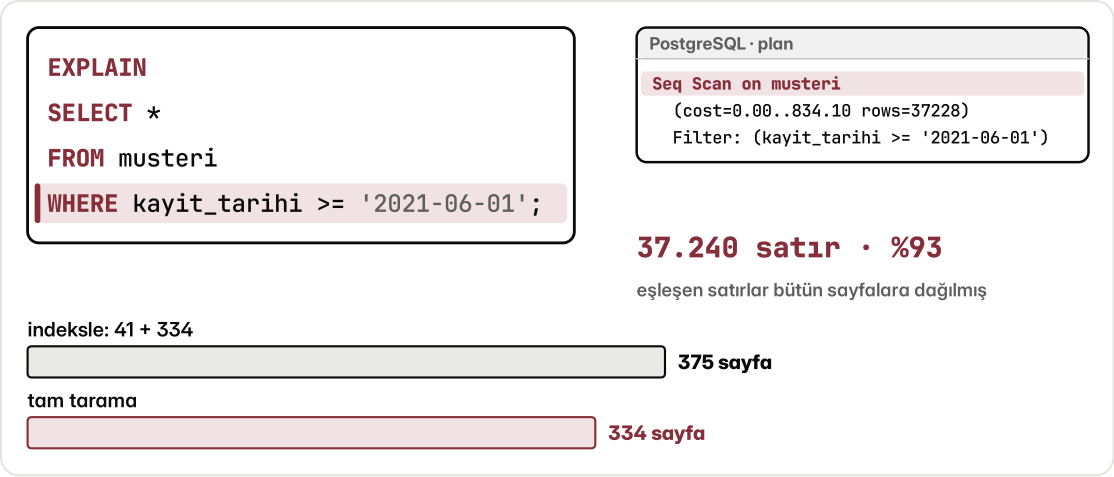
PostgreSQL'de sorgunun başına EXPLAIN yazılınca sorgu çalışmaz, yalnız seçilen plan görünür. 1 Kasım'da kaydolanlar 19 kişidir ve plan indeksi seçer:

```
Bitmap Heap Scan on musteri  (cost=4.44..67.08 rows=19 width=35)
  Recheck Cond: (kayit_tarihi = '2026-11-01'::text)
  ->  Bitmap Index Scan on musteri_kayit  (cost=0.00..4.43 rows=19 width=0)
        Index Cond: (kayit_tarihi = '2026-11-01'::text)
```

Sorgu EXPLAIN (ANALYZE, BUFFERS) ile gerçekten çalıştırılınca okunan sayfa görünür: indeksten 2, tablodan 19, toplam 21 sayfa. Tam tarama 334 sayfa okuyacaktı. Şimdi tarihi değiştirelim: `WHERE kayit_tarihi >= '2021-06-01'`. Bu kez 37.240 kişi gelir, tablonun %93'ü. Aynı sütun, aynı indeks, ama plan tam taramayı seçer:

```
Seq Scan on musteri  (cost=0.00..834.10 rows=37228 width=35)
  Filter: (kayit_tarihi >= '2021-06-01'::text)
```

Çünkü bu kadar satır bütün sayfalara dağılmıştır. Tam tarama kapatılıp indeks yolu zorlanınca ölçülen okuma indeksten 41, tablodan yine 334 sayfa, toplam 375 olur; tam tarama 334'te kalır.



Şekil 11.2 Aynı indeks, iki plan: 37.240 satırlık soruda indeksle 375, tam taramayla 334 sayfa.

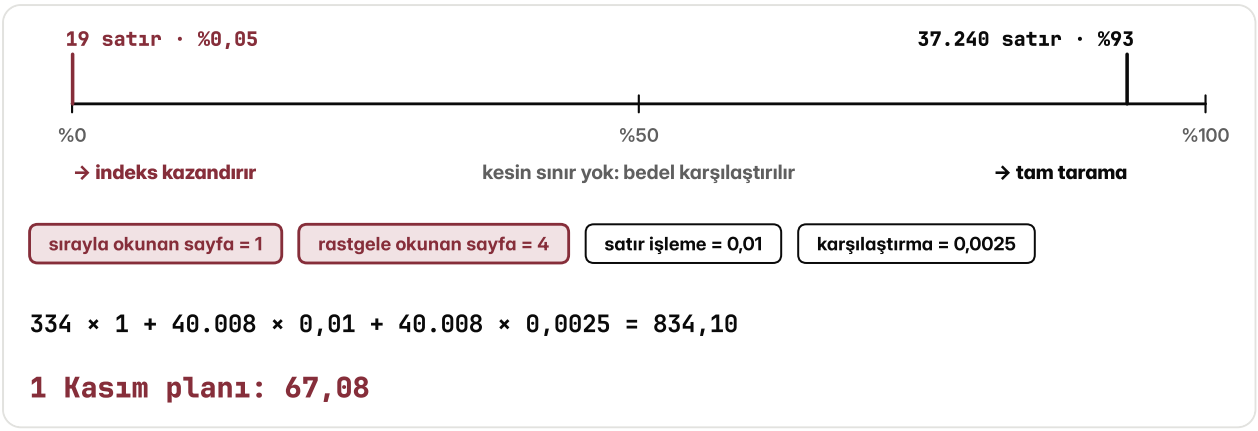
Sorgunun tablonun yüzde kaçını getirdiğine **seçicilik** (selectivity) denir. 19 kişi 40.008'in binde yarımı bile değildir: seçici bir koşul, indeks kazandırır. %93 seçici değildir: indeks yalnızca fazladan iş çıkarır. Arada kesin bir sınır yoktur; planlayıcı iki yolun bedelini hesaplayıp karşılaştırır. PostgreSQL bedeli sayfa biriminde yazar; ayarlarındaki değerler şunlardır:

ayar	değer	anlamı
seq_page_cost	1	sırayla okunan bir sayfa
random_page_cost	4	rastgele okunan bir sayfa
cpu_tuple_cost	0,01	bir satırın işlenmesi
cpu_operator_cost	0,0025	bir koşul karşılaştırması

Tam taramanın bedeli bu birimlerle adım adım hesaplanır:

```
sayfalar:          334 × 1        = 334
satır işleme:   40.008 × 0,01     = 400,08
karşılaştırma:  40.008 × 0,0025   = 100,02
toplam:                             834,10
```

Planın yazdığı 834.10 tam budur. 1 Kasım planının bedeli ise 67,08. Veritabanı en hızlı yolu böyle bilir: her yolun kaç sayfa okuyacağını tahmin eder, en azını seçer. Tahminin anahtarı seçiciliktir; kaç satır geleceği bilinmeden hiçbir yolun bedeli hesaplanamaz.



Şekil 11.3 Seçicilik cetveli: 19 satır (%0,05) indekse, 37.240 satır (%93) tam taramaya gider; altında bedel birimleri ve 834,10'un hesabı.

851 bin mi, 3 bin mi?

videoda 1:31:55

Büyümüş mağazada her siparişin yanına müşterinin adını yazmak istiyoruz; iki tablo müşteri numarasıyla birleşecek:

```
SELECT m.ad, s.tarih
FROM musteri AS m
    JOIN siparis AS s ON s.musteri_id = m.id;
```

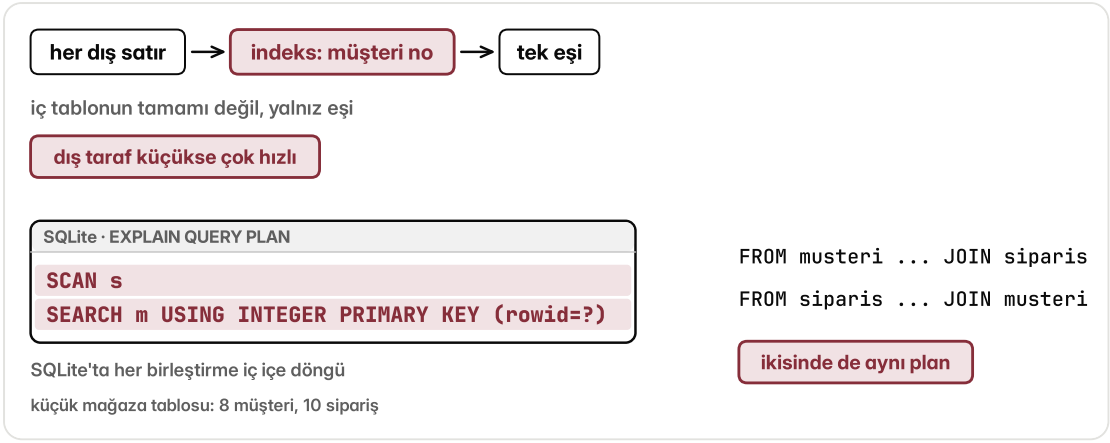
Müşteri tablosu diskte 334 sayfa (40.008 satır), sipariş tablosu 2.548 sayfa (400.010 satır) tutar. Bu birleştirmenin bir yolu 850 binden fazla sayfa okur, bir başkası 3 binden az. Fark birleştirme algoritmasından gelir.

En basit yol **iç içe döngü** (nested loop): dıştaki tablonun her sayfası için içteki tablonun bütün sayfaları okunur, satırlar çift çift karşılaştırılır. Müşteriler dışta olsun:

```
334 + 334 × 2.548 = 851.366 sayfa
40.008 × 400.010 = 16.003.600.080 çift
```

İçteki tabloda birleştirme sütununa indeks varsa döngü değişir: her dış satır için iç tablonun tamamı değil, indeksten yalnız eşi aranır. Dış taraf küçükse bu yol çok hızlıdır. SQLite'ın birleştirmesi hep iç içe döngüdür. Küçük mağaza tablolarında planı, tabloları hangi sırayla yazarsak yazalım aynıdır: sipariş tablosu baştan sona taranır, her siparişin müşterisi birincil anahtardan aranır.

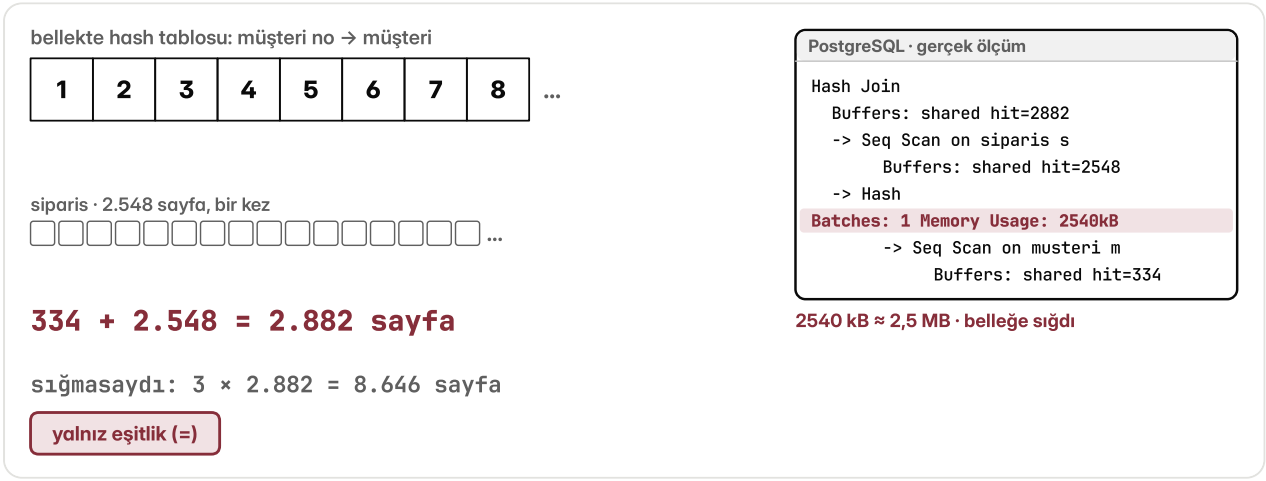
```
SCAN s
SEARCH m USING INTEGER PRIMARY KEY (rowid=?)
```



Şekil 11.4 İç içe döngünün indeksli hâli ve SQLite planı: iki yazılışta da SCAN s, SEARCH m.

İkinci yol **sıralayıp birleştirme** (merge join). İki tablo müşteri numarasına göre sıralanır, sonra iki parmak aynı anda yürür: küçük numarada duran parmak ilerler, numaralar eşitse bir çift çıkar. Tablolar zaten sıralıysa her sayfa bir kez okunur: 334 + 2.548 = 2.882 sayfa. Sıralı değilse önce sıralamanın bedeli ödenir; belleğe sığmayan tablo parça parça sıralanır, bu da tabloyu birkaç kez okuyup yazmak demektir.

Üçüncü yol **hash birleştirme** (hash join). Küçük tablo, yani müşteriler okunur ve bellekte müşteri numarasına göre bir hash tablosu kurulur. Sonra siparişler sayfa sayfa okunur; her siparişin müşterisi hash tablosunda tek adımda bulunur. Her tablo bir kez okunur: yine 2.882 sayfa. PostgreSQL'in ölçümü (EXPLAIN ANALYZE, BUFFERS) bunu doğrular: müşterilerden 334, siparişlerden 2.548 sayfa, toplam `Buffers: shared hit=2882`; hash tablosu `Memory Usage: 2540kB` (yaklaşık 2,5 MB) tuttu ve tek parçada (`Batches: 1`) belleğe sığdı. Sığmasaydı iki tablo önce parçalara bölünür, her parça ayrı birleştirilirdi; okuma yaklaşık 3 katına, 3 × 2.882 = 8.646 sayfaya çıkardı. Hash birleştirmenin bir sınırı vardır: yalnız eşitlikle çalışır.



Şekil 11.5 Hash birleştirme: müşteriler bellekteki hash tablosunda, siparişler bir kez okunuyor; yanda PostgreSQL'in ölçtüğü 334 ve 2.548 sayfa.

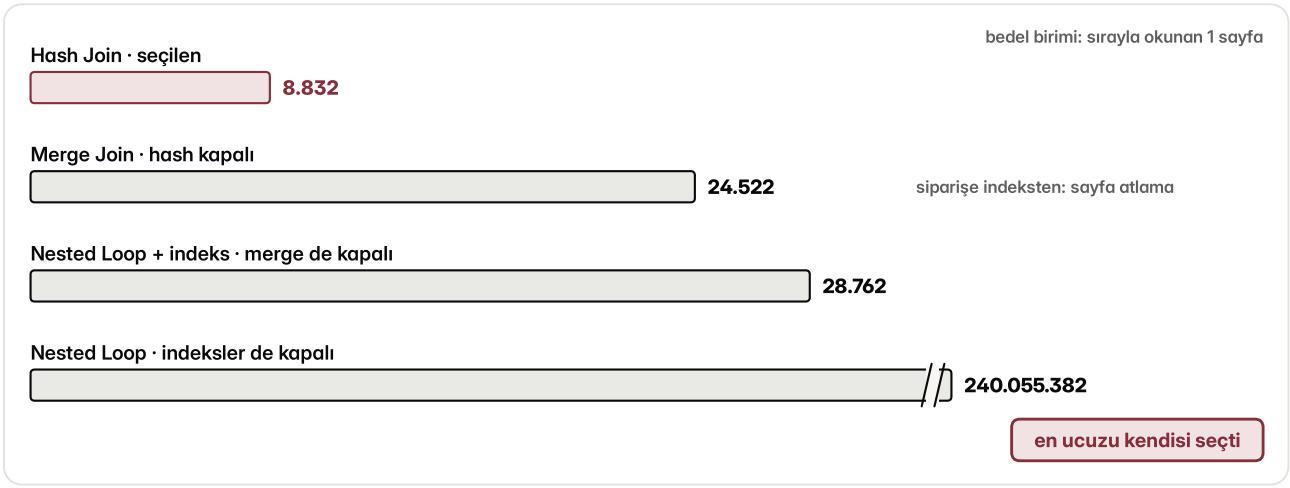
PostgreSQL bu yolların hepsini hesaplar. Seçtiği plan:

```
Hash Join  (cost=1234.18..8832.42 rows=400010 width=24)
  Hash Cond: (s.musteri_id = m.id)
  ->  Seq Scan on siparis s  (cost=0.00..6548.10 rows=400010 width=15)
  ->  Hash  (cost=734.08..734.08 rows=40008 width=17)
        ->  Seq Scan on musteri m  (cost=0.00..734.08 rows=40008 width=17)
```

Yöntemler tek tek kapatılınca (SET enable_hashjoin = off gibi) planlayıcı sıradaki en ucuz yola geçer ve her birinin bedelini gösterir:

plan	kapatılan	bedel
Hash Join	(hiçbiri, seçilen)	8.832
Merge Join	hash	24.522
Nested Loop + indeks	hash, merge	28.762
Nested Loop	hash, merge, indeksler	240.055.383

Sıralayıp birleştirme pahalı çıkar, çünkü sıralı okumak için siparişlere indeksten gider ve sayfalar arasında atlar. Planlayıcı en ucuzunu kendisi seçmişti. Ezber bir kural yoktur, hesap vardır: bir taraf küçük ve öteki tarafta indeks varsa iç içe döngü; iki büyük tablo eşitlikle birleşiyorsa hash; tablolar zaten sıralıysa sıralayıp birleştirme.



Şekil 11.6 Dört planın bedeli: Hash Join 8.832, Merge Join 24.522, indeksli iç içe döngü 28.762, yalın iç içe döngü 240 milyon.

Mekanik klavyeyi kim aldı?

videoda 1:34:44

Mekanik klavyeyi kimin aldığının cevabı dört tablodadır (SQLite, küçük mağaza):

```
SELECT m.ad, s.tarih
FROM musteri AS m
    JOIN siparis AS s ON s.musteri_id = m.id
    JOIN siparis_kalemi AS k ON k.siparis_id = s.id
    JOIN urun AS u ON u.id = k.urun_id
WHERE u.ad = 'Mekanik klavye';
```

ad	tarih
Ece Yıldız	2026-03-27

Sorguda tablolar müşteri, sipariş, sipariş kalemi, ürün sırasıyla yazıldı. Ama dört tablo 4! = 24 farklı sırayla birleştirilebilir ve hepsi aynı cevabı verir. SQLite'ın planı (EXPLAIN QUERY PLAN) başka bir sıra söyler: önce sipariş kalemleri taranır, sonra her kalemin ürünü, siparişi ve müşterisi birincil anahtardan bulunur.

```
SCAN k USING COVERING INDEX sqlite_autoindex_siparis_kalemi_1
SEARCH u USING INTEGER PRIMARY KEY (rowid=?)
SEARCH s USING INTEGER PRIMARY KEY (rowid=?)
SEARCH m USING INTEGER PRIMARY KEY (rowid=?)
```

Yazdığımız sıra yalnız okunuş içindir. Birleştirmenin sonucu sıraya bağlı değildir, bedeli ise sıraya bağlıdır. Sıraların sayısı hızla büyür: n tablo n! sıra verir.

tablo	olası sıra
3	6
5	120
8	40.320
10	3.628.800
12	479.001.600

Planlayıcı hepsini tek tek denemez. Önce ikili birleştirmelerin en ucuzunu bulur, üçlüleri bunların üstüne kurar, böyle büyür. PostgreSQL 12 ve daha çok tabloda (`geqo_threshold = 12`) bunu da bırakır, iyi bir sırayı deneme yanılmayla, genetik bir aramayla arar.

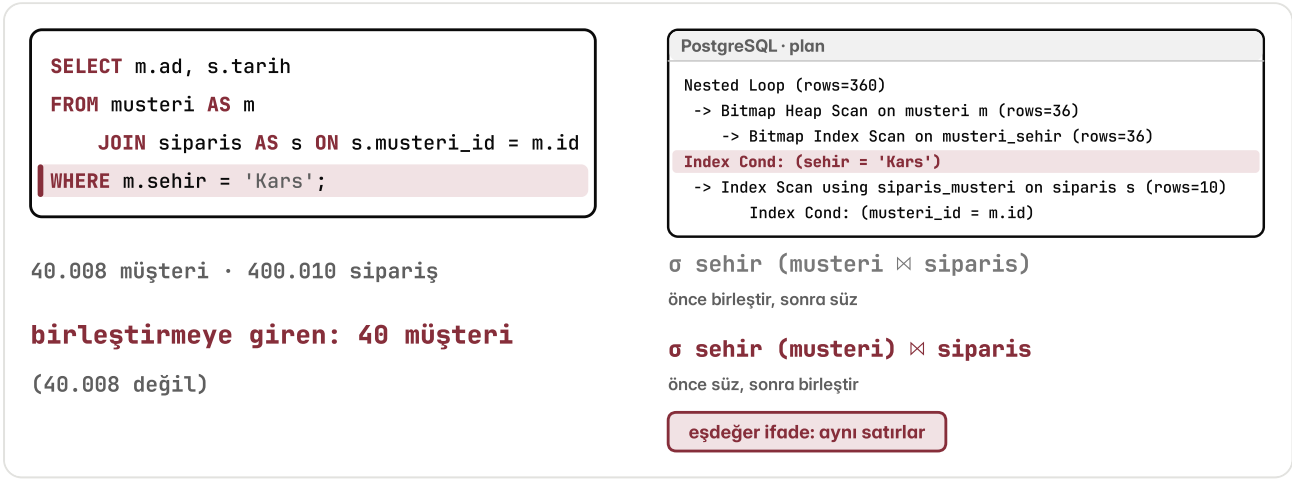


Şekil 11.7 n tablo n! sıra: 10 tabloda 3.628.800; planlayıcı ikililerden yukarı kurar, PostgreSQL 12 tablodan sonra deneme yanılmaya geçer.

Bir kural her zaman kazandırır: süzmeyi birleştirmeden önceye almak. Büyümüş mağazada Kars'taki müşterilerin siparişlerini isteyelim. Sorguda şehir koşulu en sonda, WHERE kısmında yazılıdır; PostgreSQL'in planında ise bu koşul en altta, müşteri tablosu okunurken uygulanır:

```
 Nested Loop  (cost=4.99..1624.13 rows=360 width=24)
   ->  Bitmap Heap Scan on musteri m  (cost=4.57..111.03 rows=36 width=17)
         Recheck Cond: (sehir = 'Kars'::text)
         ->  Bitmap Index Scan on musteri_sehir  (cost=0.00..4.56 rows=36 width=0)
               Index Cond: (sehir = 'Kars'::text)
   ->  Index Scan using siparis_musteri on siparis s  (cost=0.42..41.93 rows=10
width=15)
         Index Cond: (musteri_id = m.id)
```

Birleştirmeye 40.008 değil, 40 müşteri girer. σ sehir (musteri ⋈ siparis) ile σ sehir (musteri) ⋈ siparis aynı satırları verir; ikisi **eşdeğer ifade**dir ve planlayıcı ucuz olanı kendisi yazar.



Şekil 11.8 Önce süz, sonra birleştir: şehir koşulu planda müşteri tablosu okunurken uygulanıyor; birleştirmeye 40 müşteri giriyor.

Peki kaç müşterinin süzmeden geçeceğini nereden bilir? İstatistikten. Tablo yeni yüklenip istatistik toplanmadan plan istenince planlayıcı eşitlik koşulunun satırların binde 5'ini getireceğini varsayar: 40.008 × 0,005 ≈ 200. Kars için de İstanbul için de tahmin 200 müşteridir ve iki şehre aynı plan çıkar:

```
 Hash Join  (cost=313.13..7911.37 rows=2000 width=64)
   Hash Cond: (s.musteri_id = m.id)
   ->  Seq Scan on siparis s  (cost=0.00..6548.10 rows=400010 width=36)
   ->  Hash  (cost=310.63..310.63 rows=200 width=36)
         ->  Bitmap Heap Scan on musteri m  (cost=5.84..310.63 rows=200 width=36)
```

Gerçekte Kars'ta 40, İstanbul'da 15.962 müşteri vardır. İstatistik toplama komutu ANALYZE çalıştırılınca PostgreSQL müşteri tablosunun 334 sayfasını tarayıp 30.000 satırlık bir örneklem aldı (`30000 rows in sample`) ve her sütunun en sık değerlerini oranlarıyla birlikte sakladı: İstanbul 0,4013, Ankara 0,2002, ..., Kars 0,0009. İstanbul yüzde 40, Kars binde 1 bile değil.

Aynı sorgu yeniden istenince tahminler değişir:

şehir	tahmin	gerçek	plan
Kars	36	40	Nested Loop: her müşterinin siparişleri indeksten
İstanbul	16.055	15.962	Hash Join: müşteriler bellekte, siparişler bir kez

Kars planı gerçekten çalıştırılınca 922 + 41 = 963 sayfa okur. Tahmin tutunca iki şehre iki ayrı plan çıkar. Örneklem rastgele olduğu için ANALYZE her çalışmada biraz farklı tahmin verir (Kars için 36 ya da 41 gibi); plan türü değişmez. PostgreSQL istatistiği arka planda kendiliğinden de yeniler; tablo çok değiştiyse eski istatistik yanlış plana yol açar.



Şekil 11.9 Aynı sorgu, iki plan: Kars'a iç içe döngü (tahmin 36, gerçek 40), İstanbul'a hash birleştirme (tahmin 16.055, gerçek 15.962).

Olası planları sayıp her birinin bedelini istatistikten tahmin etmek ve en ucuzunu seçmek, 1979'da IBM'de System R üzerinde çalışan bir araştırma ekibinin (Selinger ve arkadaşları) tanımladığı yaklaşımdır. Bugün PostgreSQL de SQLite da aynı fikirle çalışır.

Özet

- SQL ne istendiğini söyler; sorgu ayrıştırma, cebire çevirme, plan seçme ve çalıştırma duraklarından geçer, nasılını planlayıcı seçer.
- Bedel okunan sayfa sayısıyla ölçülür; PostgreSQL'de sırayla okunan bir sayfa 1, rastgele okunan bir sayfa 4 birimdir.
- Seçicilik, sorgunun tablonun yüzde kaçını getirdiğidir: az satırda indeks, çok satırda tam tarama kazanır.
- Birleştirmenin üç yolu vardır: iç içe döngü (indeksliyse küçük dış tarafta hızlı), sıralayıp birleştirme (tablolar sıralıysa her tablo bir kez okunur) ve hash birleştirme (yalnız eşitlikte; küçük tablo belleğe sığarsa her tablo bir kez okunur).
- Planlayıcı birleştirme sırasını kendisi seçer ve süzmeyi birleştirmeden önceye alır.
- Tahminler ANALYZE'ın topladığı istatistikten gelir; istatistik yoksa ya da eskiyse plan yanlış olabilir.

Sorular

1 **40.008 satırlık müşteri tablosunda kayit_tarihi sütununa indeks var. Bir sorgu bu sütundaki koşulla tablonun %90'ını getiriyor. Planlayıcının seçmesi en olası yol hangisidir?**

A İndeksle arama, çünkü sütunda indeks var

B Tam tarama

C Önce indeks, sonra satırları tek tek rastgele okumak

D Sorgu indeks olmadan çalışamaz

2 **6 tablo kaç farklı sırayla birleştirilebilir? Yalnız sayıyı yaz.**

Cevap:

3 **Hangi birleştirme koşulunda hash birleştirme kullanılamaz?**

A s.musteri_id = m.id

B k.urun_id = u.id

C s.tarih > m.kayit_tarihi

D k.siparis_id = s.id

4 **SQLite'ta SELECT u.ad, k.adet FROM urun AS u JOIN siparis_kalemi AS k ON k.urun_id = u.id WHERE u.ad = 'Kablosuz fare'; sorgusunun planına (EXPLAIN QUERY PLAN) bakılıyor. Planın ilk satırı, yani iç içe döngünün dış tablosu hangisidir?**

A SCAN u (urun)

B SEARCH u USING INTEGER PRIMARY KEY

C SCAN k (siparis_kalemi)

D Plan yazılış sırasını izler, ilk satır her zaman FROM'daki tablodur

5 **Hangi iki ifade her zaman aynı satırları verir (eşdeğerdir)?**

A σ sehir = 'Kars' (musteri ⋈ siparis) ile σ sehir = 'Kars' (musteri) ⋈ siparis

B π ad (musteri ⋈ siparis) ile π ad (musteri)

C musteri ⋈ siparis ile musteri × siparis

D σ sehir = 'Kars' (musteri) ile σ sehir = 'Kars' (siparis)

6 **Dış tablo 120 sayfa, iç tablo 900 sayfa. Sayfa sayfa iç içe döngüyle birleştirme toplam kaç sayfa okur? Yalnız sayıyı yaz.**

Cevap:

7 PostgreSQL'de 40.008 satırlık müşteri tablosu yüklenmiş, sehir sütununa indeks kurulmuş ama istatistik toplanmamış (ANALYZE yok). EXPLAIN SELECT * FROM musteri WHERE sehir = 'Bursa'; için planlayıcı kaç satır tahmin eder?

A 3.201
B 200
C 40.008
D 1

8 Sipariş tablosu 2.548 sayfa ve 400.010 satır. PostgreSQL'de sırayla okunan bir sayfanın bedeli 1, bir satırın işlenmesi 0,01. Koşulsuz SELECT * FROM siparis; için tam taramanın bedeli kaçtır?

Cevap:

VERİTABANI SİSTEMLERİ

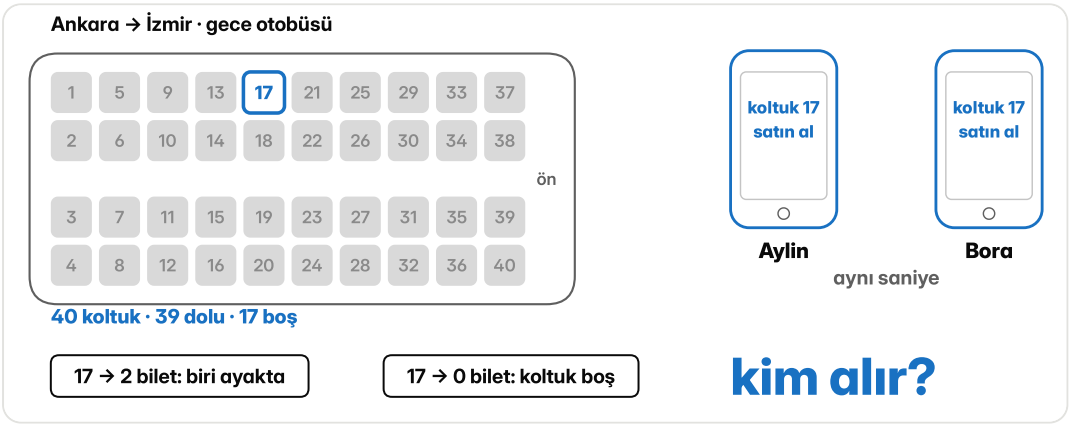
12 İşlemler, eşzamanlılık ve kurtarma

Bir veritabanına aynı anda yüzlerce kişi yazar, elektrik de her an kesilebilir. Bu bölümde veritabanının bu iki tehlikeye karşı verdiği sözleri inceliyoruz. Önce son kalan koltuğu iki kişinin aynı anda almasıyla işlemi, kayıp güncellemeyi ve kilidi göreceğiz. Sonra aynı anda çalışan işlemlerin dört derdini, sıralanabilirliği, kilit türlerini ve kilitlenmeyi ele alacağız. En sonda yalıtım düzeylerine, çok sürümlü eşzamanlılığa ve çökmeden sonra kurtarmaya bakacağız. İki bağlantılı bütün örnekler PostgreSQL 19'da gerçekten çalıştırıldı.

Son koltuk

videoda 1:37:29

Ankara'dan İzmir'e giden gece otobüsünde 40 koltuğun 39'u satılmış, yalnız 17 numara boş. Aylin ile Bora aynı saniyede "satın al" düğmesine basıyor. İkisine de bilet kesilirse biri ayakta kalır, hiçbirine kesilmezse koltuk boş gider. Örnekte iki tablo var: `koltuk(no, yolcu)` 40 satır tutar, boş koltuğun yolcusu NULL'dır; `bilet(koltuk_no, yolcu)` kesilen biletleri tutar.



Şekil 12.1 Otobüsün 40 koltuğu: yalnız 17 numara boş; Aylin ve Bora aynı koltuğu almak istiyor.

Önce tek bir satın almaya bakalım. Bilet almak iki iştir: koltuk tablosunda 17 numaraya yolcunun adını yazmak ve bilet tablosuna yeni bir satır eklemek. Biri yapılıp öteki yapılmazsa veri yalan söyler: ya adı yazılı ama bileti olmayan bir koltuk kalır ya da koltuğu olmayan bir bilet. Bu yüzden iki iş tek bir **işlem** (transaction) içine konur. İşlem BEGIN ile başlar; içindeki değişiklikler COMMIT denene kadar kesinleşmez. Bir şey ters giderse ROLLBACK denir ve işlemin yaptığı her şey geri alınır. Ya hep ya hiç.

```
BEGIN;
UPDATE koltuk SET yolcu = 'Aylin'
    WHERE no = 17;
INSERT INTO bilet
    VALUES (17, 'Aylin');
ROLLBACK;
```

İşlemin içinden bakınca koltukta Aylin görünür. Ödeme reddedilip ROLLBACK denince yeniden bakarız: 17 numaranın yolcusu NULL, bilet tablosunda 0 satır var; yarım iş hiç olmamış gibidir. Ödeme onaylandığında aynı adımlar COMMIT ile biter: koltukta Aylin, bilet tablosunda tek satır.

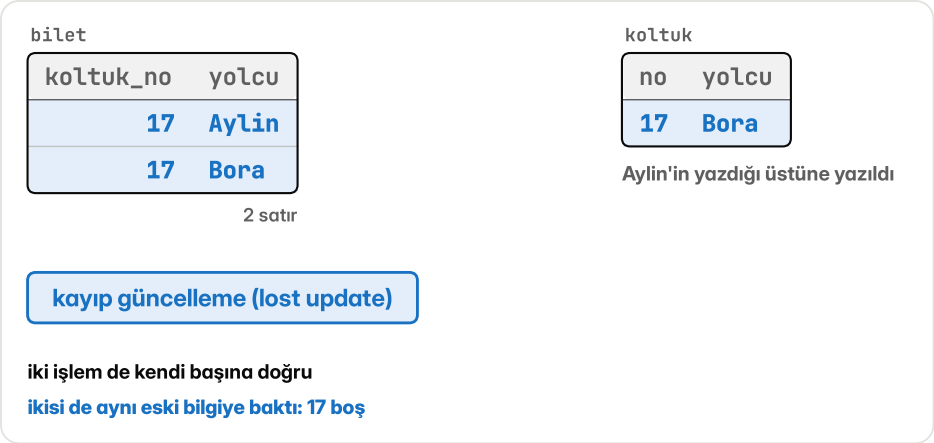
Şimdi yarışa dönelim. Aylin'in ve Bora'nın telefonları veritabanına iki ayrı bağlantıyla bağlanır, ikisi de kendi işlemini başlatır. Adımlar zaman sırasıyla şöyledir:

#	A · Aylin	B · Bora
1	`BEGIN`	`BEGIN`
2	`SELECT yolcu FROM koltuk WHERE no = 17` → NULL	
3		`SELECT yolcu FROM koltuk WHERE no = 17` → NULL
4	`UPDATE koltuk SET yolcu = 'Aylin' WHERE no = 17`	
5	`INSERT INTO bilet VALUES (17, 'Aylin')`	
6	`COMMIT`	
7		`UPDATE koltuk SET yolcu = 'Bora' WHERE no = 17`
8		`INSERT INTO bilet VALUES (17, 'Bora')`
9		`COMMIT`

Hiçbir hata çıkmaz. Ama sonuca bakınca bilet tablosunda 17 numara için iki bilet vardır; koltukta ise yalnız Bora yazar, Aylin'in yazdığı onunkinin altında kaybolmuştur.

koltuk_no	yolcu
17	Aylin
17	Bora

Buna **kayıp güncelleme** (lost update) denir. İki işlem de kendi başına doğruydu; sorun, ikisinin de aynı eski bilgiye, boş görünen koltuğa bakarak karar vermesidir.



Şekil 12.2 Kayıp güncelleme: bilet tablosunda 17 numara için 2 satır, koltukta yalnız Bora.

Çözüm, bakarken koltuğu kilitlemektir. Okuma komutunun sonuna FOR UPDATE eklenir: "bu satırı değiştirmek için okuyorum, işim bitene kadar kimse dokunmasın."

```
SELECT yolcu
FROM koltuk
WHERE no = 17
FOR UPDATE;
```

Aylin bakar, koltuk boştur, satır artık onun kilidindedir. Bora da bakmak ister ama sorgusu cevap vermez, bekler. Aylin adını yazar, biletini ekler, COMMIT eder. Kilit o anda kalkar ve Bora'nın bekleyen sorgusu cevabını alır: koltukta Aylin yazıyor. Bora'nın uygulaması koltuğu dolu görür, işlemini ROLLBACK ile kapatır; bilet tablosunda tek satır kalır.



Şekil 12.3 FOR UPDATE: Bora'nın okuması Aylin COMMIT edene kadar bekler, sonra koltuğu dolu bulur.

Bu örnek bir işlemin verdiği dört sözden ikisini gösterdi. **Bölünmezlik** (atomicity): işlem ya bütünüyle olur ya hiç olmaz; ödeme reddedilince koltuk boş kaldı. **Yalıtım** (isolation): aynı anda çalışan işlemler birbirinin yarım işine bakarak karar vermez; kilit, Bora'yı Aylin'in işi bitene kadar bekletti. Öteki ikisi **tutarlılık** (consistency): işlem veritabanını kurallara uyan bir hâlden yine kurallara uyan bir hâle taşır, bir koltuğa iki bilet kesilmez; ve **kalıcılık** (durability): COMMIT dönmüşse sonuç, elektrik kesilse de kaybolmaz. Baş harfleri **ACID** eder.

Neden sırayla değil

videoda 1:40:30

Bir çevrimiçi mağazada aynı anda yüzlerce müşteri sepet onaylar, depo stok girer. Bütün işlemleri tek sıraya dizsek hiçbir karışıklık olmazdı; ama her müşteri öndekileri bekler, işlemci ve disk çoğu zaman boş durur. Veritabanı işlemleri aynı anda çalıştırır; bunun dertleri şunlardır (örnekler mağaza veritabanından, PostgreSQL'in varsayılan düzeyi READ COMMITTED'da):

- Kayıp güncelleme: kışlık monttan 7 tane var. İki satış aynı anda stoğa bakar, ikisi de 7 görür, ikisi de 6 yazar. İki mont satıldı, stokta 6 kaldı.
- Kirli okuma (dirty read): bir işlem, başkasının henüz onaylamadığı değişikliği okur; sonra o değişiklik geri alınır ve okunan değer hiç var olmamış olur.
- Tekrarlanamayan okuma (non-repeatable read): bir işlem pamuklu gömleğin fiyatına bakar, 899 TL. Arada başka biri fiyatı 999 yapar; aynı işlem yeniden bakınca 999 görür.
- Hayalet satır (phantom): bir işlem bilgisayar aksesuarı kategorisindeki ürünleri sayar, 2. Araya yeni bir ürün eklenir, aynı sayım 3 çıkar.



Şekil 12.4 Dört dert: kayıp güncelleme, kirli okuma, tekrarlanamayan okuma ve hayalet satır, iki işlemin adımlarıyla.

Dördünün ölçüsü aynıdır: birlikte çalışan işlemlerin sonucu, aynı işlemler bir sırayla tek tek çalışsaydı çıkacak sonuçlardan biriyle aynı olmalı. Buna **sıralanabilirlik** (serializability) denir. Mont örneğinde iki satış sırayla çalışsaydı, hangisi önce giderse gitsin stokta 5 kalırdı. 6, hiçbir sıranın vermeyeceği bir sonuçtur; demek ki o birlikte çalışma sıralanabilir değildir.

Veritabanı bunu çoğu zaman kilitlerle sağlar. Okumak için **paylaşımlı kilit** (S) alınır: aynı satırı birçok işlem birlikte okuyabilir. Yazmak için **özel kilit** (X) alınır: satırı yalnız bir işlem tutar, öteki herkes bekler.

tutulan \ istenen	paylaşımlı (S)	özel (X)
paylaşımlı (S)	birlikte	bekle
özel (X)	bekle	bekle

PostgreSQL'de paylaşımlı kilit FOR SHARE ile istenir. Taşınabilir hoparlörün (105) satırında:

#	A	B
1	`BEGIN`	`BEGIN`
2	`SELECT stok FROM urun WHERE id = 105 FOR SHARE` → 9	
3		`SELECT stok FROM urun WHERE id = 105 FOR SHARE` → 9
4		`UPDATE urun SET stok = stok - 1 WHERE id = 105` (bekler)
5	`COMMIT`	bekleyen komut biter
6		`COMMIT`

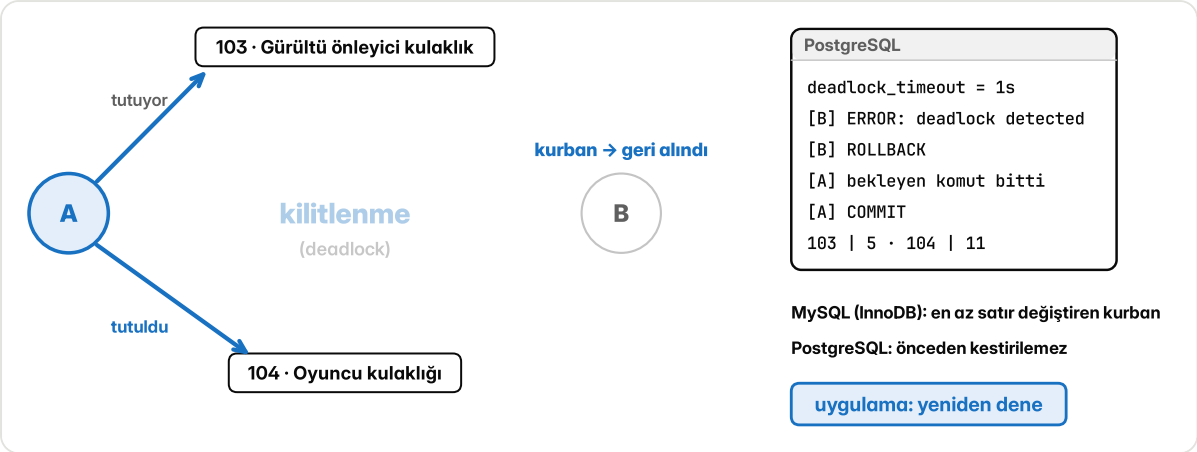
İki okuma birlikte geçer; güncelleme özel kilit ister ve A'nın kilidi sürdükçe bekler. A COMMIT edince güncelleme geçer, stok 8 olur. Bu, programlamadaki thread kilidinin veritabanındaki karşılığıdır; burada kilit bir nesnenin değil, tablodaki bir satırın üstünde durur.

Kilit ne zaman bırakılmalı? Okuyup hemen bırakmak yetmez, mont örneğindeki gibi başkası araya girer. **İki aşamalı kilitleme** (two-phase locking, 2PL) kuralında işlem önce yalnız kilit alır, hiç bırakmaz (büyüme aşaması); ilk kilidi bıraktıktan sonra yeni kilit alamaz (küçülme aşaması). Bu kurala uyan işlemler nasıl iç içe çalışırsa çalışsın sonuç sıralanabilirdir. Pratikte veritabanları kilitleri işlem bitene, yani COMMIT ya da ROLLBACK'e kadar tutar.

Kilidin bir bedeli var. İki sipariş aynı iki ürünün stoğunu ters sırayla düşürsün: A önce gürültü önleyici kulaklığı (103), B oyuncu kulaklığını (104) kilitler; sonra A 104'ü, B 103'ü ister. İkisi de ötekinin kilidini bekler, hiçbiri ilerleyemez. Buna **kilitlenme** (deadlock) denir. Veritabanı kimin kimi beklediğini izler, bir döngü bulunca işlemlerden birini kurban seçip geri alır. PostgreSQL döngüye 1 saniye bekledikten sonra bakar (`deadlock_timeout = 1s`); denemede B'nin komutu şu hatayı aldı:

```
ERROR:  deadlock detected
```

B geri alındı, A'nın bekleyen güncellemesi hemen geçti; sonunda 103'ün stoğu 5, 104'ünkü 11 oldu. Kurban seçme kuralı veritabanına göre değişir: MySQL (InnoDB) en az satır değiştirmiş işlemi seçer, PostgreSQL'in belgesi ise seçimin önceden kestirilemeyeceğini söyler. Bu yüzden uygulama, kurban giden işlemi yeniden dener.



Şekil 12.5 Kilitlenme: A ve B birbirinin tuttuğu ürünü bekler; PostgreSQL B'yi geri alır, A'nın işi geçer.

COMMIT ve elektrik

videoda 1:43:47

Bir sipariş onaylanıyor: uygulama COMMIT gönderiyor, veritabanından "tamam" cevabı geliyor ve tam o anda sunucunun elektriği kesiliyor. Değişiklikler daha bellekteydi, diskteki tabloya yazılmamıştı. Sunucu yeniden açılınca sipariş orada olacak mı? Soru ACID'in kalıcılık sözüyle ilgili; ama önce yalıtıma bakalım, çünkü bu sözün ne kadar tutulacağı ayarlanabilir.

Tam yalıtım pahalıdır: daha çok kilit, daha çok bekleme. Bu yüzden SQL standardı dört **yalıtım düzeyi** (isolation level) tanımlar:

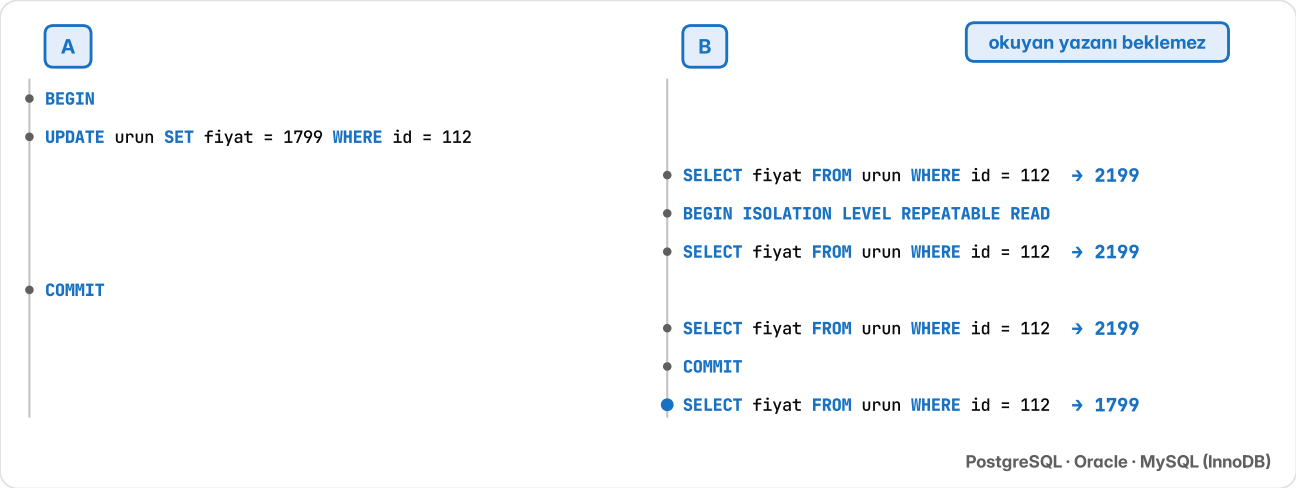
düzey	kirli okuma	tekrarlanamayan	hayalet satır
READ UNCOMMITTED	olabilir	olabilir	olabilir
READ COMMITTED	engellenir	olabilir	olabilir
REPEATABLE READ	engellenir	engellenir	olabilir
SERIALIZABLE	engellenir	engellenir	engellenir

En üstteki SERIALIZABLE, birlikte çalışmanın sonucunu tek tek çalışmışınkine eşitler. Standardın varsayılanı en üst düzeydir; PostgreSQL ise READ COMMITTED'da başlar. PostgreSQL'de en alt düzeyi istesek de kirli okuma olmaz: A mekanik klavyenin fiyatını onaylamadan 649 yaptığında, READ UNCOMMITTED ile okuyan B yine 1150 gördü.

Okumalar için kilitten başka bir yol da var: **çok sürümlü eşzamanlılık** (MVCC). Bir satır güncellenince eskisinin üstüne yazılmaz; yeni bir sürüm eklenir, eskisi bir süre daha kalır ve her işlem kendine uyan sürümü görür. Koşu ayakkabısı (112) 2199 TL:

#	A	B
1	`BEGIN`	
2	`UPDATE urun SET fiyat = 1799 WHERE id = 112`	
3		`SELECT fiyat FROM urun WHERE id = 112` → 2199
4		`BEGIN ISOLATION LEVEL REPEATABLE READ`
5		`SELECT fiyat FROM urun WHERE id = 112` → 2199
6	`COMMIT`	
7		`SELECT fiyat FROM urun WHERE id = 112` → 2199
8		`COMMIT`
9		`SELECT fiyat FROM urun WHERE id = 112` → 1799

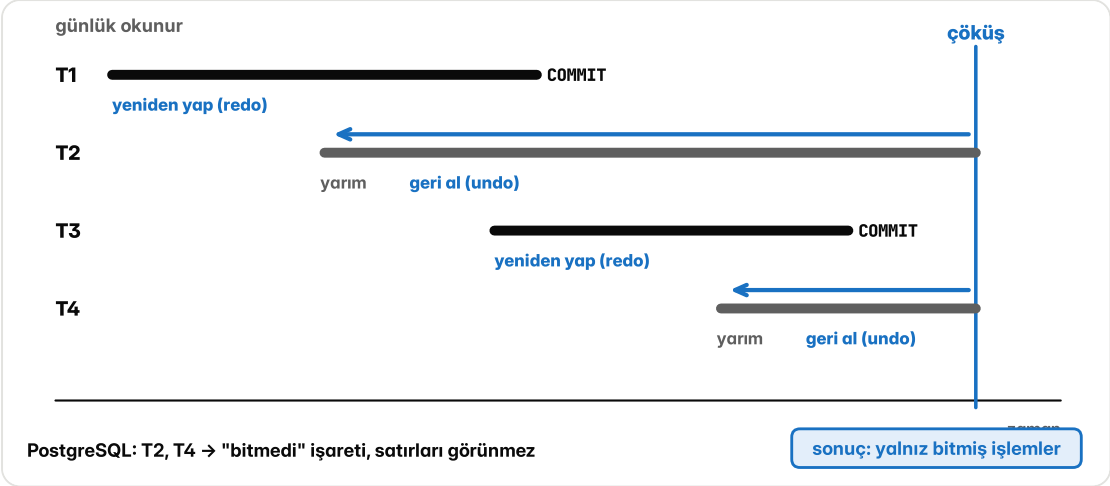
3. adımda B, A'nın onaylanmamış güncellemesini beklemeden eski sürümü okur: okuyan yazanı beklemez, yazan da okuyanı. REPEATABLE READ'deki B, indirim onaylandıktan sonra bile kendi işlemi boyunca 2199 görür; işlemini bitirip yeniden bakınca 1799. PostgreSQL, Oracle ve MySQL okumalarda bu yolu kullanır.



Şekil 12.6 Çok sürümlü eşzamanlılık: B beklemeden eski fiyatı okur; REPEATABLE READ işlemi boyunca 2199 görür.

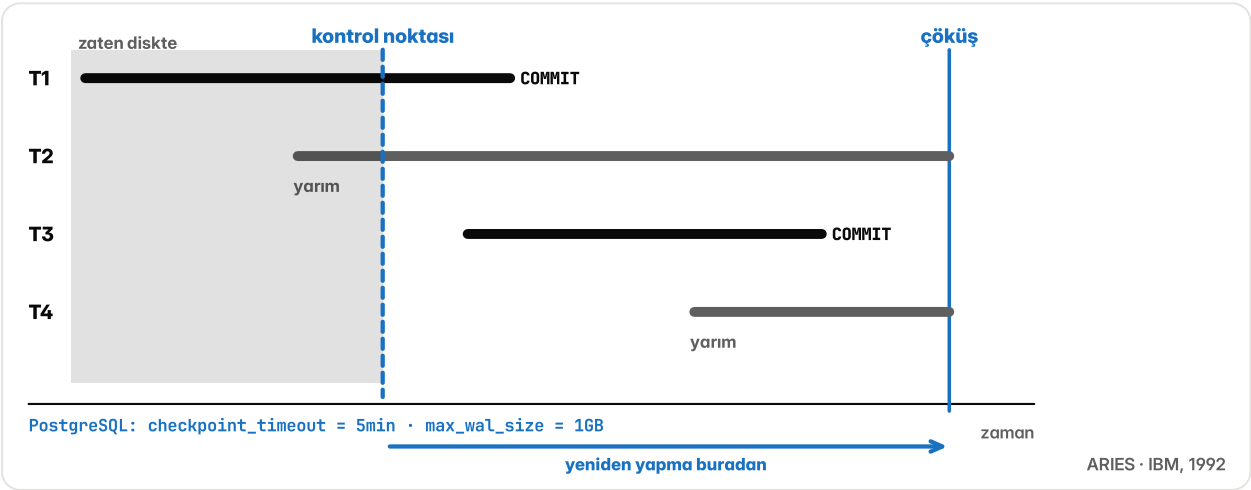
Şimdi kalıcılık. Tablolar diskte sayfa sayfa durur, değişiklik önce bellekteki kopyaya yapılır. Her COMMIT'te değişen sayfaları diske yazmak yavaş olurdu, çünkü sayfalar diskin farklı yerlerine dağılmıştır. Bunun yerine veritabanı her değişikliği önce bir **günlüğe** yazar: hangi işlem, hangi satırı, neyken neye çevirdi. Günlük, dosyanın sonuna art arda eklenen kısa kayıtlardır; diske yazması hızlıdır. Kural şudur: günlük kaydı diske ulaşmadan değişiklik diskteki tabloya yazılmaz, COMMIT kaydı diske ulaşmadan da kullanıcıya "tamam" denmez. Adı da buradan gelir: **önce günlüğe yaz** (write-ahead logging, WAL). PostgreSQL'de ilgili ayarlar varsayılan olarak açıktır (`synchronous_commit = on`, `fsync = on`).

Elektrik kesildikten sonra sunucu açılınca veritabanı günlüğü okur. Çöküşten önce COMMIT edilmiş işlemlerin (T1, T3) değişiklikleri diskteki tabloya ulaşmamış olabilir; günlükte yazdığı için **yeniden yapılır** (redo). COMMIT kaydı olmayan işlemler (T2, T4) yarım kalmıştır; tabloya ulaşmış izleri günlükteki eski değerlerle **geri alınır** (undo). PostgreSQL ise yarım işlemi geri almak yerine onu "bitmedi" diye işaretler, yazdığı satırlar kimseye görünmez kalır. Sonunda veritabanında yalnız bitmiş işlemler kalır.



Şekil 12.7 Çöküşten sonra: COMMIT edilmiş T1 ve T3 yeniden yapılır, yarım kalan T2 ve T4 geri alınır.

Günlük büyür; aylardır çalışan bir sunucuda baştan okumak saatler sürerdi. Bu yüzden veritabanı ara ara **kontrol noktası** (checkpoint) koyar: o ana kadar bellekte değişmiş bütün sayfaları diske yazar ve günlüğe bir işaret düşer. Çöküşten sonra yeniden yapma bu işaretten başlar, ondan öncesi zaten diskte. PostgreSQL varsayılan olarak 5 dakikada bir kontrol noktası koyar (`checkpoint_timeout = 5min`), günlük çok büyürse (`max_wal_size = 1GB`) daha erken.



Şekil 12.8 Kontrol noktası: solundaki her şey diskte; yeniden yapma bu işaretten çöküşe kadar.

Sorunun cevabı: "tamam" geldiyse sipariş oradadır. O cevap ancak COMMIT kaydı diskteki günlüğe ulaştıktan sonra verilir, sunucu açılınca sipariş günlükten yeniden yapılır. Cevap gelmeden kesildiyse sipariş yarım sayılır ve geri alınır; iki durumda da yarım bir sipariş kalmaz.

Özet

- İşlem (transaction) birden çok değişikliği ya hep ya hiç yapar: BEGIN ile başlar, COMMIT ile kesinleşir, ROLLBACK ile geri alınır.
- İki işlem aynı eski değere bakıp yazarsa biri ötekinin yazdığını siler (kayıp güncelleme); SELECT ... FOR UPDATE satırı okurken kilitleyerek bunu önler.
- ACID: bölünmezlik, tutarlılık, yalıtım, kalıcılık. Aynı anda çalışmanın dertleri kayıp güncelleme, kirli okuma, tekrarlanamayan okuma ve hayalet satırdır; ölçü sıralanabilirliktir.
- Paylaşımlı kilitler birlikte tutulur, özel kilit tek başına; iki aşamalı kilitleme sıralanabilirliği sağlar, bedeli kilitlenmedir ve veritabanı bir işlemi kurban seçip geri alır.
- Yalıtım düzeyleri READ UNCOMMITTED'dan SERIALIZABLE'a uzanır; PostgreSQL READ COMMITTED'da başlar, çok sürümlü eşzamanlılıkla okuyan yazanı beklemez.
- Önce günlüğe yaz kuralı kalıcılığı sağlar: çöküşten sonra bitmiş işlemler yeniden yapılır, yarımlar geri alınır; kontrol noktası kurtarmanın başlangıcını öne çeker.

Sorular

1 **Mekanik klavyenin (108) stoğu 15. İki satış işlemi aynı anda başlar; ikisi de stoğa bakar ve 15 görür. Önce A, sonra B stoğu 14 yapıp COMMIT eder. Hangi sorun oluşmuştur?**

A Kirli okuma
B Hayalet satır
C Kayıp güncelleme
D Kilitlenme

2 **Aynı çizelge bittikten sonra mekanik klavyenin stoğu kaçtır? Yalnız sayıyı yaz.**

Cevap:

3 **B işlemi fiyatı 500 TL'nin altındaki ürünleri sayar: 4. Bu sırada A işlemi 199 TL'lik yeni bir telefon kılıfı ekler ve onaylar. B aynı işlemin içinde aynı sayımı yeniden yapınca 5 bulur (READ COMMITTED). Bu hangi sorundur?**

A Hayalet satır
B Tekrarlanamayan okuma
C Kirli okuma
D Kayıp güncelleme

4 **Kablosuz farenin (109) stoğu 30. A: BEGIN; SELECT stok FROM urun WHERE id = 109 FOR UPDATE; sorgusu 30 döndürür. Ardından B aynı satırı FOR UPDATE ile okumak ister. Sonra A stoğu 29 yapıp COMMIT eder. B'nin okuması ne olur?**

- **A** Hemen 30 döner
- **B** Hata verir ve B geri alınır
- **C** Hemen 29 döner
- **D** A COMMIT edene kadar bekler, sonra 29 döner

5 **A önce pamuklu gömleğin (110), sonra kışlık montun (111) stoğunu düşürür; B önce montun, sonra gömleğin. İkisinin ilk güncellemesi geçer, ikinci güncellemeleri birbirinin kilidini bekler. PostgreSQL ne yapar?**

- **A** İkisi de sonsuza dek bekler
- **B** Kısa bir beklemeden sonra birini "deadlock detected" hatasıyla geri alır, öteki sürer
- **C** İkisini de geri alır
- **D** B'nin güncellemesi A'nınkinin üstüne yazılır

6 **B, BEGIN ISOLATION LEVEL REPEATABLE READ; ile işlem başlatır ve kablolu kulaklığın (101) fiyatını okur: 349. Bu sırada A fiyatı 399 yapıp onaylar. B aynı işlemin içinde fiyatı yeniden okursa kaç görür? Yalnız sayıyı yaz.**

Cevap: _____

7 **Saat 10.00'da kontrol noktası kondu. T1 09.58'de başladı, 09.59'da COMMIT edildi. T2 10.01'de başladı, 10.03'te COMMIT edildi. T3 10.02'de başladı ve 10.04'teki çöküşe kadar COMMIT edilmedi. Sunucu açılınca kurtarmada ne olur?**

- **A** T1 ve T2 yeniden yapılır, T3 geri alınır
- **B** T2 ve T3 yeniden yapılır, T1'e dokunulmaz
- **C** T2 yeniden yapılır, T3 geri alınır, T1'e dokunulmaz
- **D** Üçü de geri alınır, çünkü çöküş hepsini yarıda bıraktı

8 **COMMIT cevabı geldikten sonra elektrik kesilse de işlemin sonucunun kaybolmaması ACID'in hangi sözüdür? Türkçe adını yaz.**

Cevap: _____

VERİTABANI SİSTEMLERİ

13 NoSQL, dağıtık ve vektör veritabanları

Tek bir sunucuya sığmayan veri ve trafik, veritabanlarını başka biçimlere zorladı. Bu bölümde önce bir uygulama büyüdüğünde neden daha çok makineye geçildiğini ve NoSQL veritabanlarının dört büyük ailesini göreceğiz. Sonra veriyi makinelere bölmeyi, kopyalamayı ve ağ koptuğunda verilmesi gereken kararı, yani CAP teoremini ele alacağız. En sonda kelime geçmeden anlamca benzer belgeyi bulan vektör veritabanlarına ve hangi veritabanının ne zaman seçildiğine bakacağız.

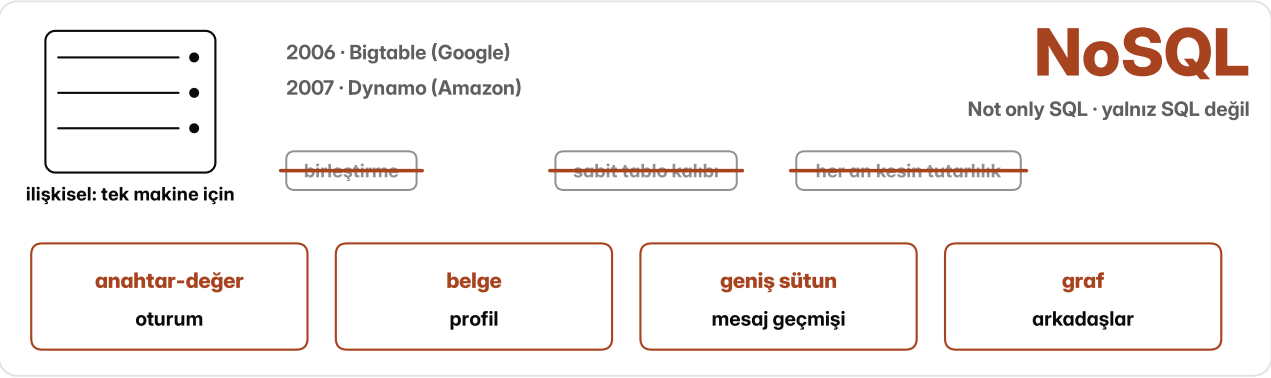
Yüz kat büyüyen uygulama

videoda 1:47:17

Bir mesajlaşma uygulamasının kullanıcı sayısı bir yılda 100 katına çıktı ve tek veritabanı sunucusu yetişmiyor. İki yol var: daha güçlü bir makine ya da daha çok makine.

Daha güçlü makineye **dikey büyüme** (scale up) denir: aynı sunucuya daha hızlı işlemci, daha çok bellek, daha büyük disk takılır. Kolaydır; ama en büyük sunucunun da bir sonu vardır, fiyat güçten hızlı artar ve tek makine bozulursa bütün uygulama durur.

Daha çok makineye **yatay büyüme** (scale out) denir. Yük artınca bir makine daha eklenir, biri bozulursa ötekiler sürer. Bedeli, verinin bölünmesidir: birleştirme satırları ağ üzerinden toplar, iki makineye dokunan bir işlemin ya hep ya hiç çalışması için makineler anlaşmalıdır. İlişkisel veritabanları tek makine düşünülerek büyüdü. 2000'li yıllarda büyük internet şirketleri başka bir yol denedi: birleştirmeden, sabit tablo kalıbından, bazen de her an kesin tutarlılıktan vazgeçen ama makinelere kolayca yayılan veritabanları. Bunlara **NoSQL** denir; ad bugün çoğu zaman "yalnız SQL değil" (Not only SQL) diye açılır.



Şekil 13.1 Yatay büyüme ve NoSQL: birleştirmeden, sabit kalıptan ve kesin tutarlılıktan vazgeçen dört aile.

NoSQL'in dört büyük ailesi var; dördünü de aynı uygulamanın dört verisinde görelim.

- **Anahtar-değer** (Redis): oturum. Kullanıcının anahtarı (`oturum:7Kq2` → `kullanıcı 40213`) her istekte kim olduğunu bulur; içi dev bir hash tablosu gibidir. Veri bellekte durduğu için cevap

milisaniyenin altında gelir, anahtara "30 dakika sonra silin" de denebilir. Bedeli: yalnız anahtarla sorulur; "Ankara'dan giriş yapan herkes kim?" sorulamaz.

- **Belge** (MongoDB): profil. İlişkisel tasarımda üç tabloya dağılan profil, burada JSON'a benzer iç içe tek bir belgedir ve tek okumada gelir. Aynı koleksiyondaki iki belgenin alanları aynı olmak zorunda değildir. Bedeli: belgeler arasında birleştirme zayıftır, aynı bilgi birkaç belgede tekrar edebilir.
- **Geniş sütun** (Cassandra): mesaj geçmişi. Satırın anahtarı sohbetin numarasıdır; her yeni mesaj o satıra zamanıyla yeni bir sütun olarak eklenir ve sıralı durur (bir satırda 3, ötekinde 3 milyon sütun olabilir). "Bu sohbetin son 50 mesajı" bir arada durduğu için tek seferde okunur. Bedeli: soru satırın anahtarıyla sorulmalıdır.
- **Graf** (Neo4j): arkadaşlıklar.

Videodaki profil belgesi (numaralar kısaltılmış):

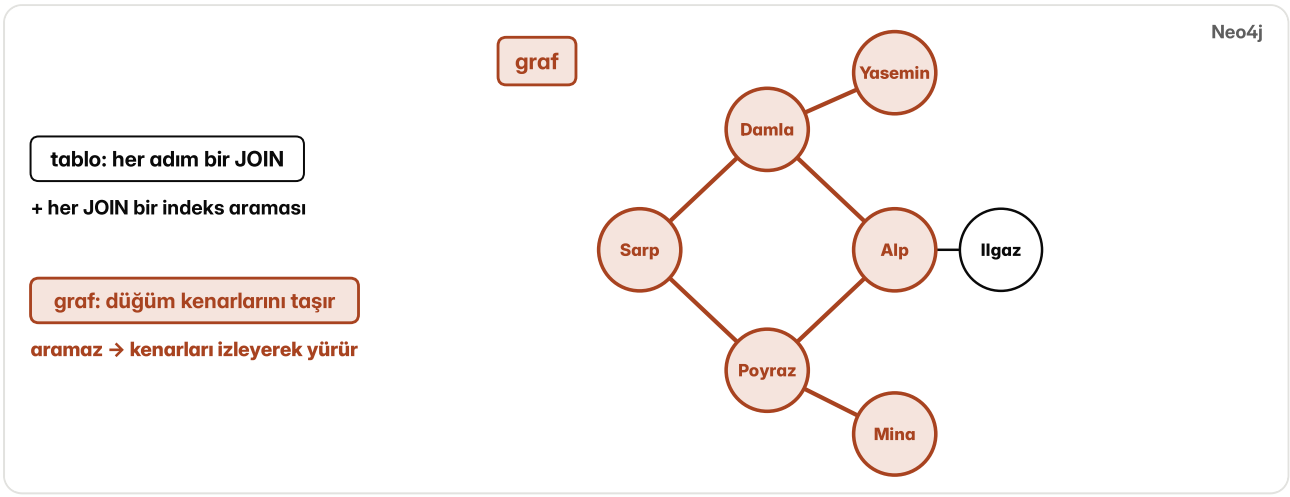
```
{
  "_id": 40213,
  "ad": "Sarp",
  "sehir": "Ankara",
  "telefonlar": ["0532 ...", "0312 ..."],
  "ayarlar": {
    "bildirim": true,
    "sessiz_saat": "23:00"
  }
}
```

Uygulama Sarp'a arkadaşlarının arkadaşlarını önermek istiyor. İlişkisel tabloda bu, `arkadaslik(kisi, arkadas)` tablosunun (14 satır; her arkadaşlık iki yönlü) kendisiyle birleşmesidir. Sorgu tabloyu iki kez kullanır: `a` Sarp'ın arkadaşlarını bulur, `b` onların arkadaşlarına geçer; WHERE Sarp'ın kendisini çıkarır, GROUP BY her öneriyi ortak arkadaş sayısıyla toplar (SQLite'ta çalıştırıldı).

```
SELECT b.arkadas AS oneri,
    COUNT(*) AS ortak
FROM arkadaslik a
    JOIN arkadaslik b
        ON b.kisi = a.arkadas
WHERE a.kisi = 'Sarp'
    AND b.arkadas <> 'Sarp'
GROUP BY b.arkadas
ORDER BY ortak DESC, oneri;
```

oneri	ortak
Alp	2
Mina	1
Yasemin	1

Alp, Sarp'ın iki arkadaşının (Damla ve Poyraz) ortak arkadaşıdır. Bir adım daha uzak için tablo bir kez daha birleşir (`JOIN arkadaslik c ON c.kisi = b.arkadas`) ve üç adımda Ilgaz çıkar. Her adım yeni bir birleştirme, her birleştirme yeni bir indeks araması demektir. Graf veritabanında kişiler düğüm, arkadaşlıklar kenardır ve her düğüm kendi kenarlarını yanında taşır; sorgu aramaz, kenarları izleyerek yürür.



Şekil 13.2 Arkadaşın arkadaşı: tabloda her adım bir JOIN; grafta düğümler kenarları izleyerek gezilir.

Cevap: daha çok makine. Ama her aile bir yetenekten vazgeçerek yayılır: anahtar-değer sorgudan, belge birleştirmeden, geniş sütun esnek sorgudan; graf ise makinelere en zor bölünendir. Büyüyen uygulama her verisine uyan aileyi seçer.



Şekil 13.3 Dört aile ve bedelleri: her biri bir yetenekten vazgeçer, her veri kendine uyan aileye gider.

İki ekran, iki sayı

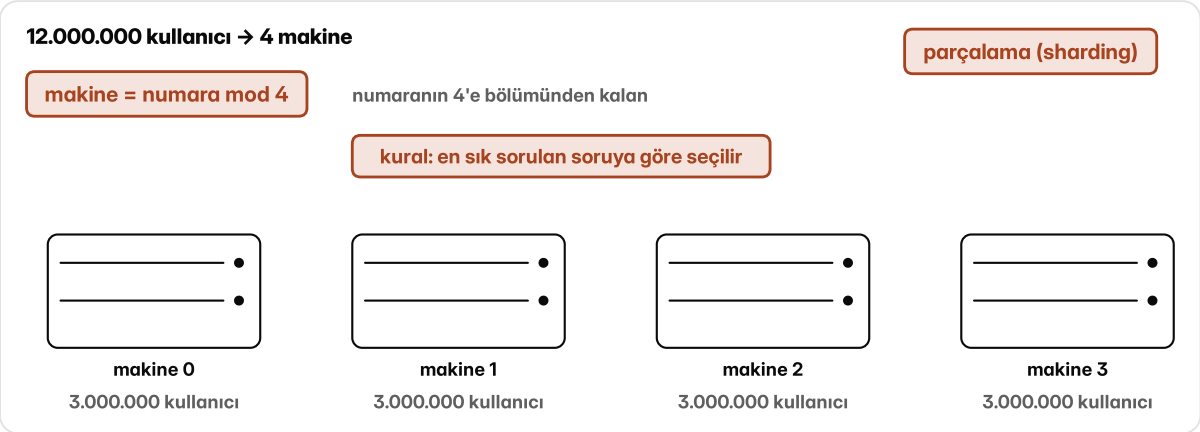
videoda 1:50:59

Bir gönderinin beğeni sayısı telefonda 1.204, aynı anda bilgisayarda 1.198 görünüyor. Hangisi doğru, veritabanı buna neden izin veriyor? Cevap için verinin makinelere nasıl dağıldığına bakmak gerekir.

12 milyon kullanıcı tek makineye sığmıyor ve 4 makineye bölünecek. Kural basit: kullanıcı numarasının 4'e bölümünden kalan (mod 4), makinenin numarasıdır.

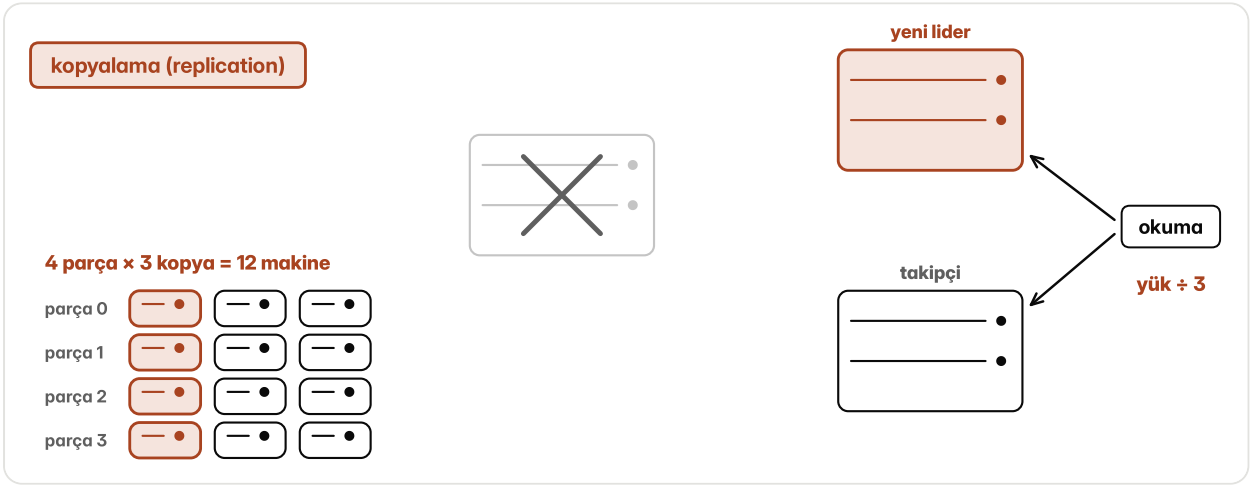
```
40213 = 10053 × 4 + 1  →  makine 1
40214 = 10053 × 4 + 2  →  makine 2
```

1'den 12.000.000'a kadar numaralar her makineye tam 3.000.000 kullanıcı düşürür. Buna **parçalama** (sharding) denir. Bir kullanıcının profili sorulunca istek doğrudan tek makineye gider; ama "Trabzon'daki bütün kullanıcılar" sorulunca dört makineye birden sorulur ve cevaplar birleştirilir. Bu yüzden bölme kuralı en sık sorulan soruya göre seçilir.



Şekil 13.4 Parçalama: numaranın 4'e bölümünden kalan makineyi seçer; her makinede 3.000.000 kullanıcı.

Parçalama tek başına yetmez: 1 numaralı makine bozulursa 3 milyon kullanıcının verisi gider. Bu yüzden her parça birkaç makinede kopya olarak tutulur; buna **kopyalama** (replication, replikasyon) denir. Üç kopya olsun. Biri **lider**dir: yazmaları yalnız o kabul eder ve her değişikliği sırayla öteki ikisine gönderir. Bu ikisine **takipçi** denir. Okuma isteklerini takipçiler de cevaplar, böylece okuma yükü üçe bölünür. Lider bozulursa takipçilerden biri yeni lider seçilir. 4 parça, 3'er kopya: 12 makine.

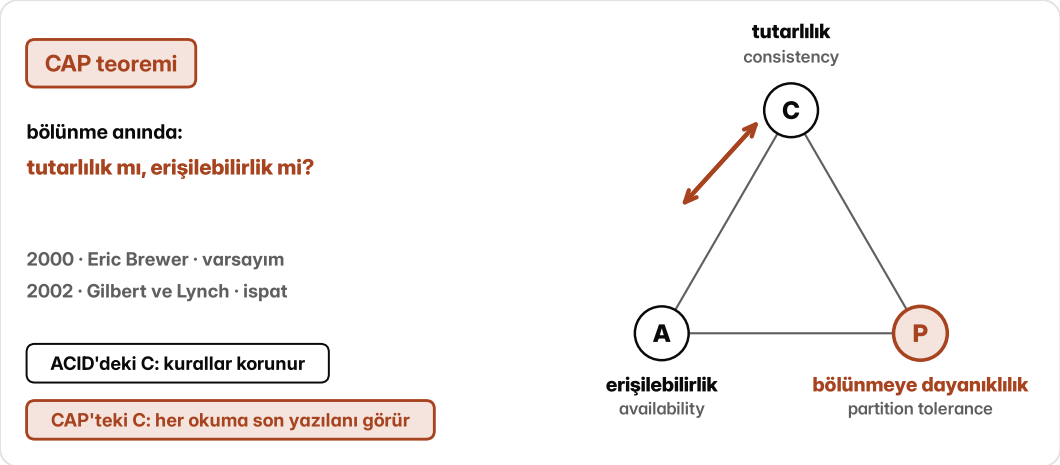


Şekil 13.5 Kopyalama: yazma lidere gider, okuma üç kopyaya dağılır; lider bozulunca takipçi yeni lider olur; 4 parça × 3 kopya = 12 makine.

Beğeni sayısının sırrı buradadır. Lider yeni beğeniyi yazar yazmaz kullanıcıya "tamam" der ve takipçilere sonra gönderir. Bu gönderim birkaç milisaniye, yoğun anlarda birkaç saniye sürebilir. Telefon lideri, bilgisayar ise henüz güncellenmemiş bir takipçiyi okuduysa iki ekran iki farklı sayı

gösterir. Lider her yazmada takipçileri bekleseydi iki sayı hep aynı olurdu, ama her beğeni daha yavaş kaydedilirdi.

Asıl zor an, makineler arasındaki bağlantı koptuğunda gelir. İki veri merkezi var ve aradaki hat kesildi; iki taraf da çalışıyor ama birbirini duymuyor. Bir taraf yeni bir yazma aldı. Veritabanının iki seçeneği var. Ya bağlantı gelene kadar bekler ya da yazmayı reddeder: her okuma en son yazılanı görür, ama sistem o sürede cevap veremez. Ya da iki taraf da cevap vermeyi sürdürür: sistem erişilebilir kalır, ama öteki taraf eski değeri okur. Bağlantı kopukken ikisi birden olmaz. Buna **CAP teoremi** denir; harfler **tutarlılık** (consistency), **erişilebilirlik** (availability) ve **bölünmeye dayanıklılık** (partition tolerance) içindir. Ağda bölünme er geç yaşanacağı için asıl seçim, bölünme anında tutarlılık ile erişilebilirlik arasındadır. Eric Brewer bunu 2000 yılında bir varsayım olarak ortaya attı; 2002'de Seth Gilbert ile Nancy Lynch biçimsel olarak ispatladı. Buradaki tutarlılık ACID'deki tutarlılıkla adaştır ama aynı şey değildir: ACID'de kuralların korunması, CAP'te her okumanın en son yazılanı görmesi demektir.



Şekil 13.6 CAP teoremi: bölünmeye dayanıklılık sabit; bölünme anında tutarlılık ile erişilebilirlik arasında seçim yapılır.

Hangisi seçilir? Verinin ne olduğuna bakılır. Biri şifresini değiştirdiyse, eski şifrenin bir kopyada hâlâ geçmesindense girişin birkaç saniye beklemesi iyidir: tutarlılık. Beğeni sayısındaki birkaç saniyelik fark ise kimseye zarar vermez: erişilebilirlik. Erişilebilirliği seçen sistemler bir söz verir: yeni yazma durursa bütün kopyalar bir süre sonra aynı değere ulaşır. Buna **nihai tutarlılık** (eventual consistency) denir. Hat düzelince iki taraf kaçırdıklarını birbirine gönderir. Telefondaki 1.204 doğruydu; bilgisayar da birazdan aynı sayıyı gösterecek. (Beğeni sayıları örnek için seçilmiştir.)

Ortak kelime yok

videoda 1:53:54

Bir yardım sayfasının arama kutusuna "telefonum akşama kadar dayanmıyor" yazılıyor; ilk sırada, ortak tek kelimesi olmayan "Pil ömrünü uzatmanın yolları" çıkıyor. Nasıl?

Kelimeyle arama burada eli boş döner. Beş yazı `yardim(id, baslik)` tablosunda durur; sorudaki dört kelimeyi başlıklarda arayan sorgu her kelime için bir LIKE satırı yazar (SQLite):

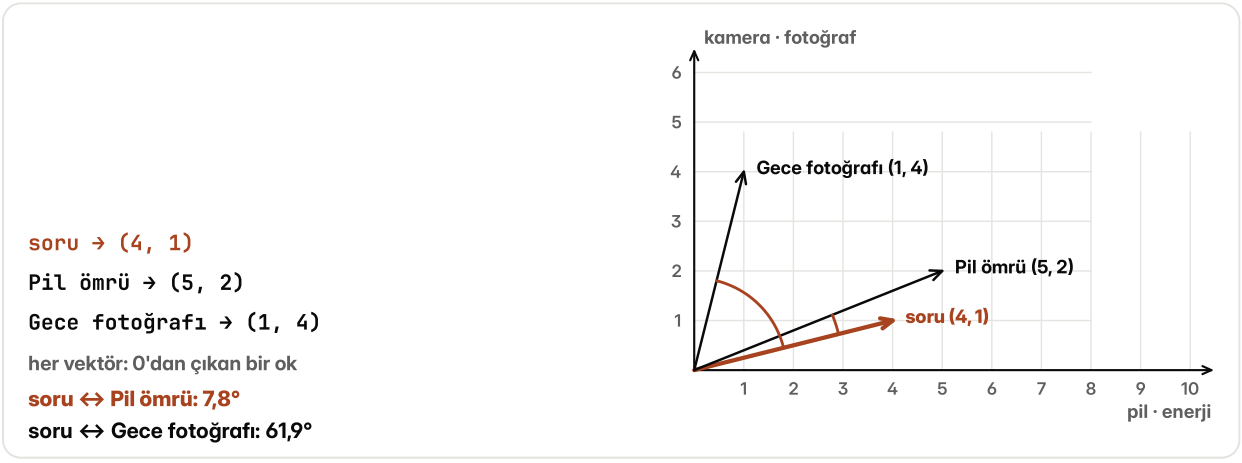
```
SELECT baslik
FROM yardim
WHERE baslik LIKE '%telefonum%'
    OR baslik LIKE '%akşama%'
    OR baslik LIKE '%kadar%'
    OR baslik LIKE '%dayanmıyor%';
```

Sonuç 0 satır. `LIKE '%pil%'` pil ömrü yazısını bulurdu (1 satır), ama soran kişi "pil" demedi.

Anlamı karşılaştırmak için bir model her metne bir sayı dizisi, yani **vektör** verir; anlamca yakın metinler yakın vektörler alır. Gerçek vektörlerde yüzlerce sayı vardır (ör. 768) ve bu sayıların adları yoktur. Aşağıdaki iki boyutlu vektörler yalnız çizebilmek için seçilmiş temsilî sayılardır: birinci sayı metnin pil ve enerjiyle, ikincisi kamera ve fotoğrafla ne kadar ilgili olduğunu gösterir.

metin	vektör
soru: telefonum akşama kadar dayanmıyor	(4, 1)
Pil ömrünü uzatmanın yolları	(5, 2)
Enerji tasarrufu modu	(4, 2)
Batarya bakımı: kapsamlı rehber	(9, 4)
Gece fotoğrafı çekmek	(1, 4)
Mercek temizliği	(0, 5)

Her vektör başlangıç noktasından çıkan bir oktur. Soru ile pil yazısı arasında 7,8 derece, gece fotoğrafıyla 61,9 derece vardır.



Şekil 13.7 İki boyutlu temsilî vektörler: soru (4, 1) ile pil ömrü (5, 2) arasında 7,8°, gece fotoğrafı (1, 4) ile 61,9°.

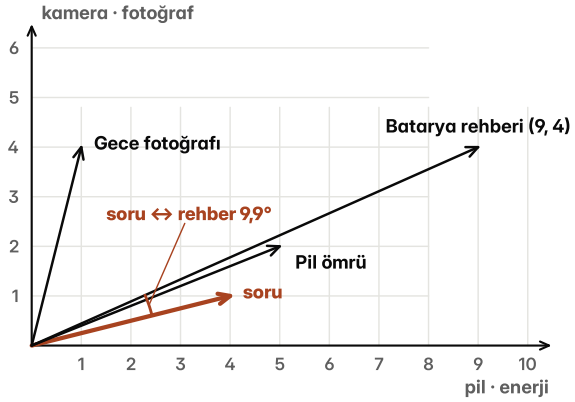
Benzerliğin ölçüsü bu açıdır, okların uzunluğu değil: neyden bahsedildiğini yön taşır, uzunluk başka şeylerle değişebilir. Batarya rehberi (9, 4) uzun bir oktur ama yönü soruyla neredeyse aynıdır (9,9°). Okların uçları arasındaki uzaklığa bakılsaydı en uzak yazı o olurdu: soruya uzaklığı 5,83, gece fotoğrafınınki 4,24. Bu yüzden açının kosinüsü kullanılır: aynı yön 1, dik 0, zıt −1. **Kosinüs benzerliği** için karşılıklı sayılar çarpılıp toplanır, sonuç iki okun uzunluklarının çarpımına bölünür.

```
cos = (a · b) / (|a| × |b|)

soru (4, 1) ile pil ömrü (5, 2):
a · b = 4×5 + 1×2 = 22
|a| = √17 = 4,12      |b| = √29 = 5,39
cos = 22 / (4,12 × 5,39) = 22 / 22,2 = 0,99

soru (4, 1) ile gece fotoğrafı (1, 4):
a · b = 4×1 + 1×4 = 8
cos = 8 / (√17 × √17) = 8 / 17 = 0,47
```

yazı	kosinüs	açı	uzaklık
Pil ömrünü uzatmanın yolları	0,991	7,8°	1,41
Batarya bakımı: kapsamlı rehber	0,985	9,9°	5,83
Enerji tasarrufu modu	0,976	12,5°	1,00
Gece fotoğrafı çekmek	0,471	61,9°	4,24
Mercek temizliği	0,243	76,0°	5,66

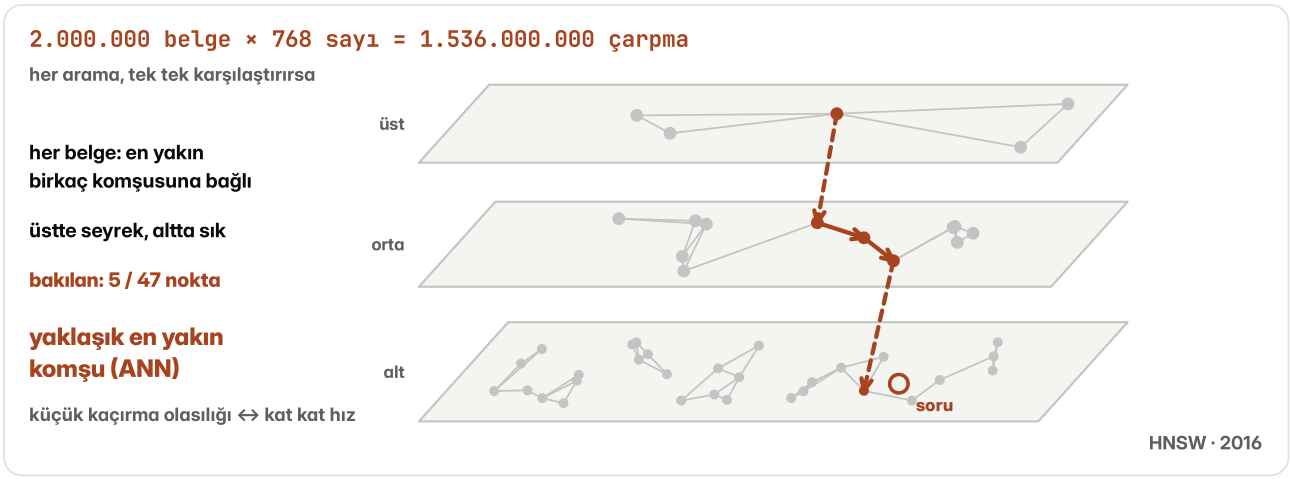
benzerlik sırası

yazı	kosinüs
Pil ömrünü uzatmanın yolları	0,991
Batarya bakımı: kapsamlı rehber	0,985
Enerji tasarrufu modu	0,976
Gece fotoğrafı çekmek	0,471
Mercek temizliği	0,243



Şekil 13.8 Kosinüs benzerliği: beş yazının soruya benzerlik sırası; en üstte pil ömrü (0,991).

2 milyon belge varsa ve her vektör 768 sayıysa, tek bir arama 2.000.000 × 768 = 1.536.000.000 çarpma ister. **Vektör veritabanı** bunun için özel bir indeks kurar. En yaygınında (HNSW, 2016) her belge kendisine en yakın birkaç belgeyle bağlanır. Bu bağlantılar katmanlar hâlinde durur: üstte az sayıda, birbirinden uzak belge, aşağı indikçe daha sık. Arama üstte başlar, her adımda soruya daha yakın komşuya geçer, daha yakını kalmayınca bir alt katmana iner; belgelerin küçük bir kısmına bakar (videodaki örnekte 47 noktanın 5'ine). Buna **yaklaşık en yakın komşu** araması (approximate nearest neighbor) denir. Yaklaşıktır, çünkü en yakını kaçırma olasılığı küçüktür ama sıfır değildir; karşılığında arama kat kat hızlanır.



Şekil 13.9 Yaklaşık en yakın komşu: arama seyrek üst katmandan başlar, soruya yaklaşarak alt katmanlara iner.

Akış böyledir: yazılar önceden vektöre çevrilip saklanır, soru da aynı modelle çevrilir, veritabanı en yakın vektörleri benzerlik sırasıyla getirir. Soru, "pil" geçmeden pil ömrü yazısını bu yüzden bulur: kelimeler değil, yönler karşılaştırıldı.

Peki hangisi seçilir? Varsayılan cevap yine ilişkisel veritabanıdır. Tablolar ve anahtarlar, tekrarı ayıklanmış bir tasarım, SQL ile sorulan her soru, kısıtların koruduğu doğruluk, indekslerin getirdiği hız ve ya hep ya hiç çalışan işlemler, verinin büyük çoğunluğu için yıllardır sınanmış bir bütündür. Ötekiler belirli bir ihtiyaç doğunca gelir: anahtarla milisaniyenin altında okumak için anahtar-değer, şekli kayıttan kayda değişen veri için belge, durmadan yazılan ve zamana göre sıralı veri için geniş sütun, ilişkilerin üstünde yürümek için graf, anlamca aramak için vektör. Büyük uygulamaların çoğu bunlardan birkaçını yan yana kullanır. Hangisi seçilirse seçilsin amaç aynıdır: veriyi, sorulduğunda doğru cevap hızla ve güvenle geri gelecek biçimde saklamak.

Özet

- Dikey büyüme tek makineyi güçlendirir ama tavanı, fiyatı ve tek arıza noktası vardır; yatay büyüme makine ekler, bedeli veriyi bölmektir.
- NoSQL'in dört ailesi (anahtar-değer, belge, geniş sütun, graf) birer yetenekten vazgeçerek makinelere kolay yayılır.
- Parçalama veriyi makinelere böler (ör. numara mod 4); kopyalama her parçayı lider ve takipçilerde tutar, takipçiler geride kalabilir.
- CAP teoremi: ağ bölündüğünde tutarlılık ile erişilebilirlikten yalnız biri korunur; erişilebilirliği seçen sistemler nihai tutarlılık sözü verir.
- Vektör veritabanı metinleri vektöre çevirir ve kosinüs benzerliğiyle ($\cos = a \cdot b / (|a| \times |b|)$) yön karşılaştırır; yaklaşık en yakın komşu araması bunu hızlandırır.
- Varsayılan seçim ilişkisel veritabanıdır; öteki aileler belirli bir ihtiyaç için gelir.

Sorular

1 **Bir uygulama oturumları "oturum:7Kq2" gibi bir anahtarla saklıyor, cevabın milisaniyenin altında gelmesini ve oturumun 30 dakika sonra kendiliğinden silinmesini istiyor. Hangi NoSQL ailesi buna en uygundur?**

A Graf
B Geniş sütun
C Anahtar-değer
D Belge

2 **Kullanıcılar dört makineye "makine = numara mod 4" kuralıyla bölünüyor. 77303 numaralı kullanıcının verisi hangi makinededir?**

A Makine 3
B Makine 0
C Makine 1
D Makine 2

3 **Veri 6 parçaya bölünüyor ve her parça 3 kopyayla (1 lider, 2 takipçi) tutuluyor. Toplam kaç makine gerekir? Yalnız sayıyı yaz.**

Cevap:

4 **İki veri merkezi arasındaki hat koptu. Sistem, bağlantı gelene kadar bir taraftaki yazmayı bekletiyor ve o sürede cevap vermiyor. CAP teoremine göre hangisini seçmiştir?**

A Erişilebilirlik
B Tutarlılık
C Nihai tutarlılık
D Bölünmeye dayanıklılık

5 **Videodaki temsilî vektörlerle soru (4, 1) ile "Enerji tasarrufu modu" (4, 2) arasındaki kosinüs benzerliğini iki ondalık basamakla yaz.**

Cevap:

6 **Soru (4, 1) iken "Batarya bakımı: kapsamlı rehber" (9, 4) kosinüse göre ikinci sırada (0,985), ama okların uçları arasındaki uzaklığa göre soruya en uzak yazıdır (5,83). Benzerlikte neden açı kullanılır?**

A Uzaklığı hesaplamak karekök istediği için
B Kosinüs her zaman uzaklıkla aynı sırayı verdiği için
C Uzun okların anlamı daha az önemli olduğu için
D Neyden bahsedildiğini yön taşır; okun uzunluğu başka şeylerle değişebilir

7 **Videodaki arkadaslik tablosunda Mina'nın tek arkadaşı Poyraz'dır. Sarp için yazılan arkadaşın arkadaşı sorgusu Mina için çalıştırılırsa (WHERE a.kisi = 'Mina' AND b.arkadas <> 'Mina') kaç satır döner? Yalnız sayıyı yaz.**

Cevap: _____

8 **Vektör veritabanının katmanlı indeksle yaptığı aramaya neden "yaklaşık" en yakın komşu denir?**

A Kosinüs yerine uzaklık kullandığı için

B Belgelerin küçük bir kısmına baktığı için en yakını kaçırma olasılığı küçük ama sıfır olmadığı için

C Vektörleri yuvarlayarak sakladığı için

D Yalnız üst katmandaki belgeleri döndürdüğü için

Cevaplar ve çözümler

0 Veritabanı nedir: dosyadan veritabanına

1. **B** · Aynı bilginin tekrarından doğan tutarsızlık
   Adres iki dosyada ayrı ayrı tutuluyor; aile taşınınca yalnız biri düzeltilmiş. Aynı bilgi birkaç yerde tutulunca er geç birbirini tutmaz: tekrar, tutarsızlığı doğurur.

2. **D** · Kurtarma
   Kurtarma, elektrik kesilirse yarım kalan işi geri alır. Dosyalarda ise kayıt iki yerde var, birinde yok ve yarım kaldığını gösteren bir işaret de yok.

3. **2**
   3. hatta Irmak, Kerem ve Arda var. Irmak ödemiş; Kerem ve Arda ödememiş. Ad düzelince Kerem de eşleşir ve cevap 2 öğrenci olur: Kerem Polat ve Arda Yurt.

4. **A** · Örnek değişti, şema aynı kaldı
   Şema tablonun kalıbıdır (no, ad, adres, hat); yeni satır kalıbı değiştirmez. Örnek o anki satırlardır ve bir satır eklenince değişir.

5. **C** · CREATE TABLE servis (no INTEGER PRIMARY KEY, ad TEXT NOT NULL, adres TEXT, hat INTEGER);
   Veri tanımlama dili tablonun kalıbını kurar ve değiştirir; CREATE TABLE bunu yapar. SELECT, INSERT ve UPDATE satırlarla çalışır, yani veri işleme diline (DML) aittir.

6. **B** · Fiziksel veri bağımsızlığı
   İndeks iç düzeyde, yani diskteki düzende bir değişikliktir. İç düzey değişse de sorgu ve sonucu aynı kalıyorsa buna fiziksel veri bağımsızlığı denir. Mantıksal veri bağımsızlığı ise kavramsal şemaya yeni bir sütun eklenince görünümlerin değişmemesidir.

7. **1**
   3. hatta Irmak, Kerem ve Arda var. Bunlardan yalnız Irmak'ın yemek ücreti ödenmiş (evet); Kerem ve Arda'nınki hayır. Sorgu 1 döndürür.

8. **C** · Veriye giden yol (okulu bul, sınıfa in, öğrenciye geç) programın içine yazılı olduğu için
   Bu sistemlerde program veriye kayıtların arasındaki yolu adım adım yürüyerek ulaşırdı ve yol programın içine yazılıydı. Düzen değişince yol da değişir, programlar yeniden yazılırdı. Üç düzeyli mimari bu bağı koparır: program yalnız neyi istediğini söyler.

1 İlişkisel model ve ilişkisel cebir

1. **C** · Not henüz verilmedi, değer bilinmiyor
   NULL bilinmeyen ya da olmayan bir değerdir; burada ders sürüyor ve not henüz verilmedi. NULL sıfır değildir (0 bir nottur), boş bir yazı da değildir.

2. **B** · {ad, bolum}
   Süper anahtar iki satırda hiç aynı olmayan sütun grubudur. 2101 ve 2103'ün ikisi de Ahmet Yalçın ve Bilgisayar; {ad, bolum} bu iki satırda aynı. {no}, {eposta} ve {no, ad} her satırda farklıdır.

3. **D** · Ad çıkarılınca kalan {no} da satırları ayırdığı için

   Aday anahtar, hiçbir sütunu çıkarılamayan süper anahtardır. {no, ad}'dan ad çıkarılınca kalan {no} yine ayırır; demek ki ad fazladır. Aday anahtar birden çok sütunlu da olabilir: kayıt tablosundaki (no, kod) gibi.

4. **3**

   Bölüm sütununda 5 değer var: Bilgisayar, Elektrik, Bilgisayar, Bilgisayar, Makine. İzdüşüm tekrar eden satırları siler; geriye Bilgisayar, Elektrik ve Makine kalır: 3 satır.

5. **C** · 20

   Çarpımda birinci tablonun her satırı ikincinin her satırıyla yan yana konur: 5 × 4 = 20 satır. 9 toplamdır; 55 ise Ogrenci × Kayit'ın (5 × 11) satır sayısıdır.

6. **D** · (2106, 'ALG', NULL)

   Öğrenci tablosunda 2106 numaralı satır yok; yabancı anahtar var olmayan bir satırı gösteremez (referans bütünlüğü). SQLite "FOREIGN KEY constraint failed" der. Ötekilerin numaraları var ve bu öğrenciler o derse daha önce yazılmamış.

7. **2101, 2104**

   AYR'yi alanlar 2101 ve 2104; ALG'yi alanlar 2101, 2102, 2104 ve 2105. Kesişim ikisinde de olanları verir: 2101 ve 2104.

8. **3**

   Bölme, bölen tablonun her satırıyla, yani hem VT hem ALG ile eşleşen numaraları verir. 2101, 2102 ve 2104 ikisini de almış; 2103 yalnız VT'yi, 2105 yalnız ALG'yi almış. Sonuç 3 öğrenci.

2 SQL temel: tablo kurmak ve sorgulamak

1. **C** · FROM, WHERE, SELECT, ORDER BY

   Önce FROM tabloyu getirir, sonra WHERE satırları süzer; SELECT ancak kalan satırlardan sütunları alır, en son ORDER BY sıralar. Yazılış sırası SELECT ile başlasa da çalışma sırası FROM ile başlar.

2. **B** · 4

   BETWEEN aralığın iki ucunu da içine alır: 349, 389, 449 ve 499'un dördü de gelir. Uçları dışarıda sanan 389 ile 449'u, yani 2 satırı bulur.

3. **A** · Bilinmiyor (NULL)

   NULL değer değil, bilinmeyenin işaretidir; NULL'la her karşılaştırma bilinmiyor çıkar, NULL = NULL bile. WHERE yalnız doğruyu bıraktığı için = NULL hiçbir satır getirmez; doğrusu IS NULL'dur.

4. **D** · 5

   İzmirli 2 kişi düşer. Deniz'in şehri bilinmediği için İzmir'e eşit olmayıp olmadığı da bilinmez; o satır da düşer. 8 − 2 − 1 = 5 satır kalır. Deniz'i de istemek için OR sehir IS NULL eklenir.

5. **2**

   WHERE yalnız yüzdesi 15'ten küçük olanları seçer: YAZ10 (10) ve YENI5 (5). OKUL15'in yüzdesi 15'e eşit olduğu için seçilmez; veritabanı 2 satırın güncellendiğini söyler.

6. **5**

   'K%' K ile başlayan, ardından herhangi sayıda karakter gelen adları seçer: Kablolu kulaklık, Kablosuz kulak içi, Kablosuz fare, Kışlık mont ve Koşu ayakkabısı. Adın içinde k geçmesi yetmez; desen K ile başlamayı ister.

7. **B** · 0 satır; sütun başlıkları yerinde durur

   WHERE'siz DELETE bütün satırları siler ama tablo sütunlarıyla yerinde kalır; sorgu 0 satır döndürür. Tabloyu kendisiyle birlikte kaldıran DROP TABLE'dır; ondan sonra no such table hatası gelir.

8. **4**

   WHERE 5 ürünü bırakır, kategorileri 3, 3, 4, 5 ve 7'dir. DISTINCT tekrar eden 3'ü bir kez bırakır: 3, 4, 5, 7, yani 4 satır.

3 SQL orta: birleştirme ve gruplama

1. **A** · 10 ve 9

   COUNT(*) bütün satırları sayar: 10. COUNT(musteri_id) yalnız o sütunu dolu olanları sayar; NULL'lu misafir siparişi atlanır: 9.

2. **2**

   İç birleştirmede yalnız eşi olan satırlar kalır: Zeynep'in 2 siparişi müşterisiyle eşleşir, Ozan'ın eşleşecek siparişi yoktur. WHERE İzmirli olanları bırakır; sonuç 2.

3. **D** · 11

   Ozan önceden NULL'lu tek satırla geliyordu; şimdi o satırın yerine siparişiyle eşleşen tek satır gelir. Toplam değişmez: 11. (Doğrulamada 1011 numaralı sipariş eklenince sonuç yine 11.)

4. **B** · SELECT urun_id, COUNT(*) FROM siparis_kalemi GROUP BY urun_id HAVING COUNT(*) >= 2;

   Grupları süzen HAVING'dir ve GROUP BY'dan sonra çalışır; sorgu 5 ürün (101, 102, 107, 109, 110) döndürür. WHERE'de COUNT yazmak "misuse of aggregate: COUNT()" hatası verir, çünkü WHERE çalışırken henüz grup yoktur.

5. **3**

   GROUP BY her siparişin kalemlerini toplar, HAVING 3000'i geçen grupları bırakır: 1002 (21498), 1004 (3997) ve 1007 (4149). Sonuç 3 satır.

6. **C** · 3

   UNION aynı değeri bir kez bırakır: 102, 104, 107, yani 3 satır. UNION ALL tekrarları da tutardı ve 4 satır verirdi.

7. **D** · 107

   EXCEPT ilk listede olup ikincide olmayanları verir: 102 iki listede de var, 107 yalnız ilkinde. Sonuç 107, hızlı şarj aleti. Yanlar yer değiştirseydi 104 çıkardı; EXCEPT'te sıra önemlidir.

8. **6**

   Tek siparişli müşterinin siparişi ancak kendisiyle eşleşebilir, onu da <> eler. İki siparişli her müşterinin çifti bir kez bu sırayla, bir kez ters sırayla gelir: 3 × 2 = 6. Her çifti bir kez bırakmak için b.tarih > a.tarih yazılır; o zaman 3 satır.

4 SQL ileri: alt sorgu ve pencere fonksiyonu

1. **B** · 0

   Alt sorgunun listesi 6, 2, 7, 3 ve NULL'dur. Her müşteri NULL'la da karşılaştırılır; sonuç bilinmiyor olduğu için NOT IN hiçbir satırı doğru çıkaramaz. Sorgu 0 satır döndürür; 4, NULL olmasaydı gelecek cevaptır.

2. 4

Haziran'dan beri sipariş veren müşteriler 2, 3, 6 ve 7 numaralılardır. Misafirin NULL numarası hiçbir müşterininkine eşit olmadığı için soruyu bozmaz. Geriye Ece Yıldız, Can Demir, Selin Öztürk ve Ozan Koç kalır: 4 satır.

3. C · 6

İçteki sorgu tek bir değer döndürür: kulaklıkların ortalaması (349 + 499 + 1299 + 2499) / 4 = 1161,5. Dıştaki sorgu bütün ürünlerde bu değerden pahalı olanları sayar: 2499, 1299, 1750, 18999, 3250 ve 2199; sonuç 6. Yalnız kulaklıklara bakan biri 2 der.

4. D · Mert Aydın

Sorgu "almadığı tek bir üst giyim ürünü bile olmayan" müşteriyi arar. Zeynep'in almadığı mont var, elenir. Mert'in almadığı ürün yok; sonuç yalnız Mert Aydın.

5. 3

Başlangıç parçası Giyim'i getirir; 1. turda çocukları Üst giyim ve Ayakkabı gelir; 2. turda yeni satır çıkmaz. Bu üç kategorideki ürünler pamuklu gömlek, kışlık mont ve koşu ayakkabısıdır: 3.

6. A · az kaldı

CASE koşulları yukarıdan aşağı dener ve ilk tutan kazanır. 4 < 10 doğru olduğu için 'az kaldı' yazılır; 4 < 5 koşuluna hiç sıra gelmez. Bu yüzden dar koşul önce yazılmalıdır.

7. D · 3

Dört kulaklık aynı penceredir ve stoğa göre çoktan aza dizilir: 40, 25, 12, 6. Oyuncu kulaklığı (12) üçüncü sıradadır. Az stoktan başlayan biri 2 der; DESC çoktan aza dizer.

8. 3793

Yürüyen toplam her satıra o satıra kadarki toplamı yazar: 1047, 1496, 2494, 3793. 1006'nın son kalemi dördüncü satırdır; kasa 3793 lira. Sonraki iki kalemle 7942'ye çıkar.

5 Veritabanı programlama ve saklı yordamlar

1. C · Hiçbir satır değişmemiş görülür

rowcount yalnız o bağlantının içinde kaç satırın değiştiğini söyler. commit çağrılmadan bağlantı kapanınca değişiklik onaylanmamış sayılır ve kaybolur; stoklar eski değerleriyle görünür.

2. 6

Stoğu 15'ten az olan ürünler gürültü önleyici kulaklık (6), oyuncu kulaklığı (12), taşınabilir hoparlör (9), akıllı telefon (4), kışlık mont (7) ve koşu ayakkabısıdır (11): 6 satır. Mekanik klavyenin stoğu tam 15'tir, < koşulunu sağlamaz.

3. B · Değer komutun parçası olamaz ve aynı sorgu metni yeniden kullanılır

Ayrı gönderilen değer ne olursa olsun yalnız değer kalır, komutu değiştiremez. Sorgu metni değişmediği için veritabanı hazırladığını yeniden kullanabilir. Sonucun satır sayısı ve commit gereği yazımdan etkilenmez.

4. D · 10

1 okuma (SELECT) ve 9 güncelleme (UPDATE): 10 komut. Aynı işi WHERE stok < 30 koşullu tek bir UPDATE 1 komutla yapar.

5. 897

798, 1500'ün altında olduğu için IF koşulu doğru çıkar ve 99 eklenir: 798 + 99 = 897.

6. **498,66**

   1496 / 3 = 498,666... kuruşa yuvarlanınca 498,67. İlk iki taksit 498,67; son ayda koşul çalışır: 1496 − 2 × 498,67 = 498,66. Toplam 1496,00 tutar.

7. **A** · Fonksiyon değer döndürür ve SELECT içinde kullanılır; yordam CALL ile çağrılır ve çoğu zaman veriyi değiştirir

   siparis_tutari bir değer döndürür ve SELECT id, siparis_tutari(id) FROM siparis gibi bir sorgunun içinde çağrılır. taksitlendir kendi başına CALL ile çağrılır ve taksit tablosuna satır yazar. PostgreSQL'de yordamı SELECT içinde çağırmak hata verir.

8. **B** · SQLite programın içinde çalışan bir kütüphanedir; program ile veritabanı arasında kazanılacak ağ trafiği yoktur, kural programın kendi dilinde yazılır

   SQLite ayrı bir sunucu değil, gömülü bir kütüphanedir. Saklı yordamın ağ gidiş dönüşünü azaltma kazancı burada yoktur; gereken fonksiyon programın dilinde yazılıp create_function ile bağlantıya tanıtılır.

6 Veri bütünlüğü ve güvenlik

1. **C** · UPDATE urun SET fiyat = 0 WHERE id = 103;

   CHECK (fiyat > 0) sıfırı kabul etmez: fiyat = 0 komutu "CHECK constraint failed: fiyat > 0" hatası alır. 0,5 sıfırdan büyüktür; 40 - 40 = 0 ve 0 da stok >= 0 kuralını tutar, bu yüzden öteki üçü yazılır.

2. **2**

   Denetim kapalı olduğu için silme hatasız geçer. Zeynep'in 1004 ve 1009 numaralı siparişleri hâlâ 3 numaralı müşteriyi gösterir, ama böyle bir müşteri artık yoktur: 2 yetim sipariş.

3. **A** · Selin silinir; sipariş tablosu değişmez.

   RESTRICT yalnız silinecek satırı gösteren bir sipariş varsa silmeyi reddeder. Selin'i gösteren sipariş yoktur; satır silinir, müşteri sayısı 8'den 7'ye iner, 10 sipariş ve 20 kalem olduğu gibi kalır.

4. **4**

   Silme zincirleme ilerler: 1 müşteri, onun 1 siparişi ve o siparişin 2 kalemi; 1 + 1 + 2 = 4 satır. Sayılar 8, 10, 20'den 7, 9, 18'e iner. Komut yine de "1 satır etkilendi" der, çünkü yalnız müşteri satırını sayar.

5. **D** · 2

   SET NULL Emre'nin 1006 numaralı siparişini silmez, müşteri numarasını NULL yapar. Önceden NULL olan 1010 ile birlikte 2 sipariş olur: 1006 ve 1010.

6. **12**

   Kalem eklenir eklenmez tetikleyici kendiliğinden çalışır ve stoktan NEW.adet kadar düşer: 15 - 3 = 12. Stok için ayrı bir komut yazılmaz.

7. **B** · UPDATE urun SET stok = 0 WHERE id = 101;

   Depo rolünün güncelleme yetkisi yalnız stok sütunu içindir: stok komutu UPDATE 1 döner. Fiyatı değiştirmek ve ürün silmek "permission denied for table urun", müşteri tablosunu okumak "permission denied for table musteri" hatası alır.

8. **B** · 0

   AND, OR'dan önce bağlanır: koşul eposta = 'x' OR ('1'='1' AND sifre_ozeti = '...') diye okunur. 'x' adresli üye yok ve özet hiçbir üyeninkine uymuyor; sorgu 0 satır döner. Videodaki saldırının sonundaki iki tire şifre denetimini açıklamaya çevirdiği için 4 satır getiriyordu; tırnaklar eşleştiği için sözdizimi hatası da çıkmaz.

7 ER ve EER ile kavramsal tasarım

1. **B** · Çift elips

   Birden çok değer alabilen nitelik çok değerli niteliktir ve çift elipsle çizilir (telefon gibi). Kesik elips türetilmiş niteliği, altı çizili ad anahtarı, çift dikdörtgen zayıf varlığı gösterir.

2. **41**

   2027 - 1985 = 42, ama 9 Mayıs'ta doğum günü henüz gelmemiştir: hasta 41 yaşındadır, ertesi gün 42 olur. Aynı hesabı yapan sorgu 9 Mayıs için 41, 10 Mayıs için 42 döndürür; tabloya yazılmış bir yaş tam bu yüzden eskir.

3. **2**

   (205, 1) ve (312, 1) zaten var, anahtar tekrar edemez: "UNIQUE constraint failed: yatak.oda_no, yatak.yatak_no". 999 numaralı oda yok: "FOREIGN KEY constraint failed". (205, 3) ve (312, 2) kabul edilir; tabloda 7 yatak olur.

4. **C** · Hasta tablosuna aile hekiminin numarası yabancı anahtar olarak eklenir.

   Bire çokta yabancı anahtar çok olan tarafa konur: her hasta satırına tek bir hekim numarası yazılır. Hekim tarafına konsaydı bir hücreye birden çok hasta numarası gerekirdi; ara tablo yalnız çoka çok ilişkide gerekir.

5. **D** · 3 satır, 2 doktor

   50121 üç satırda geçer: 7001 ile 2 Kasım ve 16 Kasım, 7003 ile 9 Kasım. Doktorlar 7001 ve 7003, yani 2 farklı doktor. Aynı hasta aynı doktora başka bir gün yine geldiği için tarih de anahtarın parçasıdır.

6. **A** · Her doktorun tam bir bölümü var; bir bölümde sıfır ya da çok doktor olabilir.

   Bir ucun işaretleri, karşıdaki bir örneğin o taraftan kaç örnekle bağlandığını söyler. Bölüm ucundaki iki dik çizgi "en az bir, en çok bir": her doktorun tam bir bölümü var. Doktor ucundaki daire ve üç çatal "sıfır olabilir, çok olabilir": yeni açılan bir bölümde henüz doktor olmayabilir.

7. **C** · Özelleştirme hem tam hem ayrıksa

   Kısmi olursa hiçbir alt sınıfa girmeyen personelin (radyoloji teknikeri gibi) yeri kalmaz. Örtüşen olursa aynı kişi iki tabloda tekrar eder ve bütün personel listesinde iki kez görünür. Ancak herkes bir alt sınıfa giriyor ve kimse iki alt sınıfta değilse bu yol sorunsuzdur.

8. **7**

   2 doktorun vardiyası (2), 3 hemşirenin uzmanlığı (3) ve teknikerin iki hücresi (2): 2 + 3 + 2 = 7. Bu iki sütunda 6 × 2 = 12 hücre var; 7'si NULL. Tek tablo yolunun bedeli bu boş hücrelerdir.

8 Normalizasyon

1. **C** · Ekleme anomalisi

   Satır eklemek için anahtarın parçaları (sınıf, gün, saat) gerekiyor; öğretmenin dersi olmadığı için bunlar yok. Bir bilgiyi kaydetmek için ilgisiz bir bilgi uydurmak gerekiyorsa buna ekleme anomalisi denir.

2. **B** · ders → ogretmen

   Ders sütunu aynı (Matematik) olan satırlarda öğretmen sütunu farklı (Kemal Uçar, Filiz Tekin); bu yüzden ders → ogretmen tutmaz. Sınıf da verilince öğretmen tektir, sinif, ders → ogretmen tutar.

3. **5**

   {sinif, ders} ile başla. sinif, ders → ogretmen ile ogretmen, ogretmen → telefon ile telefon, sinif → derslik ile derslik eklenir. Gün ve saat getirilemez: {sinif, ders, ogretmen, telefon, derslik}, 5 sütun.

4. **{A, D}**

   A ve D hiçbir okun sağında yok; her anahtarda bulunmak zorundalar. $\{A, D\}^+$: A → B ile B, B → C ile C, A, D → E ile E eklenir; 5 sütunun hepsi. $\{A\}^+ = \{A, B, C\}$ ve $\{D\}^+ = \{D\}$ yetmez; tek aday anahtar {A, D}.

5. **D** · 1NF (2NF değil)

   D anahtar olmayan bir sütun ve anahtarın yalnız B parçasına bağlı (B → D). Bu 2NF'ye aykırıdır; tablo yalnız 1NF'dedir. Düzeltmek için (B, D) ayrı tabloya alınır.

6. **A** · 2NF (3NF değil)

   Anahtar tek sütun olduğu için parçasına bağlılık olamaz, tablo 2NF'dedir. Ama C anahtara B üzerinden bağlı (A → B → C); B süper anahtar değil, C de anahtarın parçası değil. Bu 3NF'ye aykırıdır.

7. **C** · 3NF (BCNF değil)

   C → B okunun sol yanı C süper anahtar değil, bu yüzden tablo BCNF'de değil. Ama B bir aday anahtarın parçası; 3NF buna izin verir. Okuldaki atama tablosunun (ogretmen → ders) durumu da budur.

8. **3**

   Tabloda 3 × 2 = 6 satır var. Yeni gün her kulüple eşleşmeli: 3 satır eklenir, tablo 9 satır olur. 4NF'de kulüpler ve nöbet günleri ayrı tablolarda tutulsaydı yeni gün tek satırla eklenirdi.

9 Yarı yapısal veri: XML ve JSON

1. **C** · {"kamera_mp": 50, "renkler": ["siyah", "mavi"]} kaydı

   JSON kaydı her değeri adıyla birlikte taşır; önceden çizilmiş bir kalıp yoktur, kayıt kendini tarif eder. Tablo satırı yapısaldır (kalıp önce çizilir), yorum metni ve fotoğraf yapısızdır (hangi parçanın ne olduğunu söyleyen işaret yoktur).

2. **B** · Siyah ve mavi

   Başa konan çift eğik çizgi "nerede olursa olsun" demektir; ağaçtaki bütün renk etiketleri bulunur. Aynı etiket iki kez kullanıldığı için sonuç iki değerdir: siyah ve mavi.

3. **3**

   Bağlantısı kablolu olan ürünler kablolu kulaklık, oyuncu kulaklığı ve mekanik klavyedir: 3. Bağlantı anahtarı olmayan ürünlerde json_extract NULL döner ve koşul doğru çıkmaz.

4. **D** · NULL

   Gömleğin JSON'ında renkler anahtarı yoktur; olmayan anahtar hata vermez, NULL döner. Telefonun satırında ise sayma 0'dan başladığı için [1] ikinci rengi, maviyi verir. M, beden dizisinin [1]'idir; yol renkler'i sorduğu için gelmez.

5. **2**

   json_each beden dizisini satırlara açar: gömlek S, M, L, XL diye 4, mont M, L diye 2 satır verir. L ikisinde de vardır; sonuç pamuklu gömlek ve kışlık mont, 2 satır.

6. **B** · 6, 30, 12

   ->> değeri yazı olarak verir; yazılar sayı gibi değil harf harf karşılaştırılır. Büyükten küçüğe ilk karakterler 6, 3, 1 sırasıyla dizilir: 6, 30, 12. Sayı sırası (30, 12, 6) için değer önce (ozellik ->> 'pil_saat')::int ile tam sayıya çevrilmelidir.

7. **A** · {"beden": ["M", "L"]}

   JSONB metni çözümleyip saklar; aynı anahtar iki kez yazıldıysa sonuncuyu tutar. Diziler birleştirilmez, hata da verilmez. Sonuç {"beden": ["M", "L"]}.

8. **0**

   Anahtar adları büyük-küçük harfe duyarlıdır: Pil_saat diye bir anahtar hiçbir üründe yoktur, 12 satırın hepsinde NULL döner. COUNT NULL'ları saymaz; sonuç 0. Hata ya da uyarı çıkmaz; doğru yazımla ('$.pil_saat') sonuç 3 olur.

10 Fiziksel tasarım ve indeksler

1. **B** · Bir müşteriyi aramak ucuzlardı, ama ortaya giren her sipariş için arkasındaki satırlar kaydığından ekleme pahalanırdı.

   Sıralı dosyada aranan yer ikiye bölerek bulunur (6.225 sayfada 13 sayfa). Ama yeni satır sırasındaki yere girer ve arkasındaki satırlar kayar; sırasız dosyada ise yalnız son sayfaya yazılır. Bir dosya da yalnız tek bir sütuna göre sıralı tutulabilir.

2. **C** · 5

   BETWEEN iki ucu da sayar: 6, 7, 9, 11 ve 12 aralıkta, 4 ve 15 dışarıda. Sonuç 5 satır (Gürültü önleyici kulaklık, Kışlık mont, Taşınabilir hoparlör, Koşu ayakkabısı, Oyuncu kulaklığı); planda SEARCH urun USING INDEX urun_stok (stok>? AND stok<?) yazar.

3. **4**

   Kökte 5, 9'dan küçük: birinci yaprağa inilir (6, 7 alınır). Zincirle ikinci yaprağa geçilir (9, 11, 12 alınır). Üçüncü yaprakta 15, 12'yi aştığı için yürüyüş biter. Kök ve üç yaprak: 4 sayfa.

4. **D** · WHERE musteri_id = 31415 indeksle arar; ötekilerde tam tarama yapar.

   İndeks önce müşteri numarasına, aynı müşteride tarihe göre sıralı (telefon rehberinde önce soyad, sonra ad gibi). İlk sütun sorguda geçince arama yapılır: SEARCH siparis USING INDEX siparis_musteri_tarih (musteri_id=?). Yalnız tarih ya da tutar sorulunca o değerler bütün müşterilerin arasına dağılmıştır; plan SCAN siparis olur.

5. **B** · WHERE musteri_id = 31415

   Hash anahtarı doğrudan bir kova numarasına çevirir; komşu numaralar uzak kovalara düşer, sıra kaybolur. Bu yüzden yalnız eşitlikte kullanılır: Bitmap Index Scan on siparis_musteri_hash. Büyüktür, BETWEEN ve sıralamada plan Seq Scan on siparis olur.

6. **8**

   Kovayı tek göz gösteriyorsa onu bölmek için ayırt edici bir bit daha gerekir: dizin ikiye katlanır, 4 gözden 8 göze çıkar ve son 3 bite bakar. Yine de yalnız dolu kova bölünür. Kovayı iki göz paylaşsaydı dizin büyümeden yalnız kova bölünürdü.

7. **C** · Taşan anahtar taşma sayfasına yazılır ve sırası gelen 0 numaralı kova bölünür.

   Doğrusal hash dizin kullanmaz, kovaları önceden belli sırayla birer birer böler. Taşan anahtar geçici olarak taşma sayfasına yazılır; bölünen, sırası gelen kovadır. 2 numaralı kova kendi sırası gelince bölünür ve taşma sayfası kalkar.

8. 4

İlk geçiş 20 sayfadan 7 sıralı parça çıkarır (6 parça 3 sayfa, 1 parça 2 sayfa). İkişer birleştirme: 7 → 4 → 2 → 1, yani 3 geçiş daha. Toplam 1 + 3 = 4 geçiş; her geçişte tablo bir kez okunur ve bir kez yazılır.

11 Sorgu işleme ve iyileştirme

1. B · Tam tarama

Koşul seçici değil: satırların %90'ı bütün sayfalara dağılmıştır. İndeksle gidilse indeksin sayfaları üstüne tablonun hemen bütün sayfaları yine okunur. Bölümdeki örnekte %93'lük sorguda tam tarama 334, indeks yolu 375 sayfa okudu; indeksin varlığı tek başına onu kullanmayı gerektirmez.

2. 720

n tablonun sırası n! kadardır: 6! = 6 × 5 × 4 × 3 × 2 × 1 = 720. Sayı hızla büyüdüğü için planlayıcı hepsini tek tek denemez; ikili birleştirmelerin en ucuzundan yukarı kurar.

3. C · s.tarih > m.kayit_tarihi

Hash birleştirme, bir satırın eşini hash tablosunda tek adımda bulur; bu yalnız eşitlikte işe yarar. "Büyüktür" koşulunda aranan tek bir değer yoktur, bir aralık vardır. Öteki üç koşul eşitliktir.

4. C · SCAN k (siparis_kalemi)

SQLite'ın planı iki satırdır: SCAN k, ardından SEARCH u USING INTEGER PRIMARY KEY (rowid=?). Sipariş kalemleri taranır, her kalemin ürünü birincil anahtardan bulunur; ürün tablosu FROM'da önce yazılsa da iç döngüdedir. Birleştirme sırasını yazılış değil planlayıcı belirler. Sorgu 2 satır döndürür (adet 1 ve 2).

5. A · σ sehir = 'Kars' (musteri ⋈ siparis) ile σ sehir = 'Kars' (musteri) ⋈ siparis

Şehir yalnız müşteri tablosunun sütunu olduğu için süzme birleştirmeden önceye alınabilir; sonuç değişmez, bedel düşer (bölümdeki örnekte birleştirmeye 40.008 yerine 40 müşteri girdi). İkincide hiç siparişi olmayan müşteriler farklılaşır, üçüncüde çarpım eşleşmeyen çiftleri de tutar, dördüncüde sipariş tablosunda sehir sütunu yoktur.

6. 108120

Dış tablo bir kez okunur (120), dıştaki her sayfa için iç tablonun tamamı okunur (120 × 900 = 108.000): 120 + 108.000 = 108.120 sayfa. Hash birleştirme, küçük tablo belleğe sığarsa yalnız 120 + 900 = 1.020 sayfa okurdu.

7. B · 200

İstatistik yokken planlayıcı eşitlik koşulunun satırların binde 5'ini getireceğini varsayar: 40.008 × 0,005 ≈ 200; plan da rows=200 yazar. Gerçekte Bursa'da 3.201 müşteri vardır; bu farkı ancak ANALYZE'ın topladığı en sık değerler kapatır.

8. 6548,10

Sayfalar: 2.548 × 1 = 2.548. Satırlar: 400.010 × 0,01 = 4.000,10. Koşul olmadığı için karşılaştırma payı yok. Toplam 6.548,10; PostgreSQL'in planı da Seq Scan on siparis (cost=0.00..6548.10 rows=400010) yazar.

12 İşlemler, eşzamanlılık ve kurtarma

1. **C** · Kayıp güncelleme
   İki işlem aynı eski değere (15) bakıp yazdı; B'nin yazdığı 14, A'nın yazdığının üstüne geldi ve A'nın satışı kayboldu. Bu kayıp güncellemedir. Onaylanmamış bir değer okunmadığı için kirli okuma değildir; kimse kimseyi beklemediği için kilitlenme de yoktur.

2. **14**
   İki işlem de 14 yazdı; PostgreSQL'de çalıştırınca son stok 14 çıkar. İki satış sırayla çalışsaydı stok 13 olurdu; 14, hiçbir sıranın vermeyeceği bir sonuçtur, yani bu çizelge sıralanabilir değildir.

3. **A** · Hayalet satır
   Var olan hiçbir satır değişmedi; koşula uyan yeni bir satır belirdi ve sayım 4'ten 5'e çıktı. Buna hayalet satır denir. Tekrarlanamayan okuma, aynı satırın değerinin iki okuma arasında değişmesidir.

4. **D** · A COMMIT edene kadar bekler, sonra 29 döner
   FOR UPDATE satıra kilit koyar; A'nın kilidi sürdükçe B'nin FOR UPDATE okuması bekler. A COMMIT edince kilit kalkar ve B güncel değeri, 29'u okur. Bu yüzden B eski 30'a bakarak karar veremez; kayıp güncelleme önlenir.

5. **B** · Kısa bir beklemeden sonra birini "deadlock detected" hatasıyla geri alır, öteki sürer
   Bu bir kilitlenmedir: A, B'nin tuttuğu montu, B de A'nın tuttuğu gömleği bekler. PostgreSQL 1 saniye (deadlock_timeout) bekledikten sonra döngüyü bulur ve bir işlemi kurban seçer; denemede B "deadlock detected" hatası aldı ve geri alındı, A'nın bekleyen güncellemesi geçti.

6. **349**
   REPEATABLE READ'de işlem kendi anlık görüntüsünü korur; A'nın onayladığı yeni sürüm bu işlemin içinden görünmez. B yine 349 görür; işlemini bitirip yeniden bakınca 399.

7. **C** · T2 yeniden yapılır, T3 geri alınır, T1'e dokunulmaz
   Kontrol noktasında o ana kadar değişen bütün sayfalar diske yazıldı; T1'in işi bu yüzden zaten diskte, yeniden yapma kontrol noktasından başlar. T2 kontrol noktasından sonra COMMIT edildi, günlükten yeniden yapılır. T3'ün COMMIT kaydı yok, yarım kalmıştır ve geri alınır.

8. **kalıcılık**
   Kalıcılık (durability): COMMIT kaydı diskteki günlüğe ulaşmadan "tamam" denmez; sunucu açılınca işlem günlükten yeniden yapılır.

13 NoSQL, dağıtık ve vektör veritabanları

1. **C** · Anahtar-değer
   Yalnız anahtarla okunan, bellekte duran ve süresi dolunca silinebilen veri anahtar-değer veritabanının (Redis gibi) işidir. Bedeli, değere göre soru sorulamamasıdır: "Ankara'dan giriş yapan herkes" sorulamaz.

2. **A** · Makine 3
   77303 = 19325 × 4 + 3; 4'e bölümünden kalan 3'tür, veri 3 numaralı makinededir. Makine numaraları 0'dan başlar; kalan 0, 1, 2 ya da 3 olabilir.

3. **18**
   Her parça 3 makinede durur: 6 × 3 = 18. Videodaki örnekte 4 parça × 3 kopya = 12 makineydi.

4. **B** · Tutarlılık

   Beklemek ya da reddetmek, her okumanın en son yazılanı görmesini korur ama sistem o sürede cevap veremez: tutarlılık seçilmiştir. Erişilebilirliği seçen sistem iki tarafta da cevap vermeyi sürdürür ve öteki taraf eski değeri okur; nihai tutarlılık o sistemlerin verdiği sözdür.

5. **0,98**

   Önce çarpıp topla: 4×4 + 1×2 = 18. Uzunluklar √17 = 4,12 ve √20 = 4,47; çarpımları 18,44. 18 / 18,44 = 0,976, iki basamakla 0,98.

6. **D** · Neyden bahsedildiğini yön taşır; okun uzunluğu başka şeylerle değişebilir

   Rehberin oku uzundur ama soruyla neredeyse aynı yöne bakar (9,9°); konusu aynıdır. Uzaklık bu uzunluk farkını ceza sayar ve rehberi en sona atar. Kosinüs yalnız yöne bakar: aynı yön 1, dik 0, zıt −1.

7. **2**

   Poyraz'ın arkadaşları Sarp, Alp ve Mina'dır; Mina çıkarılınca Alp ve Sarp kalır, ikisinin de ortak arkadaş sayısı 1'dir. SQLite'ta çalıştırınca 2 satır döner.

8. **B** · Belgelerin küçük bir kısmına baktığı için en yakını kaçırma olasılığı küçük ama sıfır olmadığı için

   Arama üst katmandan başlar, her adımda soruya daha yakın komşuya geçip alt katmanlara iner; bütün belgelere bakmaz. Bu yüzden çok hızlıdır, ama en yakın belgeyi kaçırma olasılığı küçük de olsa vardır.

Öğrendiğini pekiştir

Bu notlardaki bütün soruları ve daha fazlasını sitede, her cevabın hemen ardından çözümüyle birlikte çözebilirsin.

hekzon.com/dersler

HEKZON