T.C. BURSA ULUDAĞ ÜNİVERSİTESİ MÜHENDİSLİK FAKÜLTESİ BİLGİSAYAR MÜHENDİSLİĞİBÖLÜMÜ

VERİ TABANI ve YÖNETİM SİSTEMLERİ

DÖNEM ÖDEVİ RAPORU

CHANGE DATA CAPTURE (CDC) SQL'DEN NoSQL'E VERİ AKTARIMI UYGULAMASI

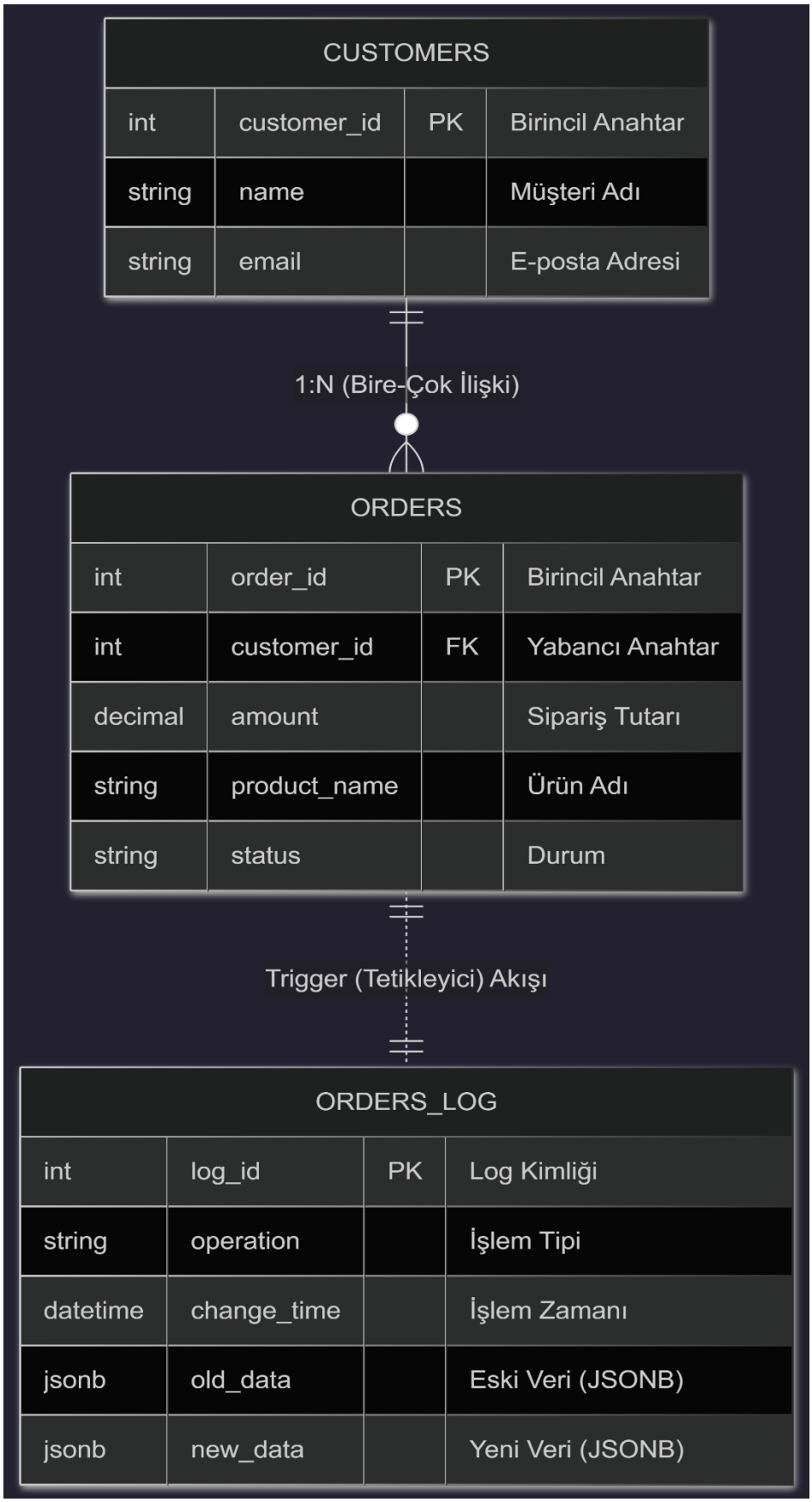
Batouchan RIZA

032190089

Dersi Veren Öğretim Üyesi: Doç. Dr. Metin BİLGİN

BURSA ARALIK 2025

1-ER DİYAGRAMI



1.1 Log Yapısı Tasarımı (Orders_log Tablosu)

Projede CDC mekanizmasını sağlamak için oluşturulan Orders_log tablosu, klasik ilişkisel veri tipleri ile modern JSON veri tiplerini bir arada kullanan hibrit bir yapıda tasarlanmıştır. Bu yapı, SQL ortamındaki katı şema kurallarına takılmadan verinin NoSQL ortamına (MongoDB) esnek bir şekilde aktarılmasını sağlar.

Tablonun sütun yapısı ve kullanım amaçları aşağıdaki gibidir:

Sütun Adı	Veri Tipi	Açıklama ve Kullanım Amacı
log_id	SERIAL (PK)	Her log kaydına otomatik olarak artan benzersiz bir numara verilir. Python scripti verileri okurken bu sırayı takip eder.
order_id	INTEGER	İşlem yapılan siparişin ID bilgisidir. (Not: Sipariş silinse bile logun kalması için Foreign Key kısıtlaması bilerek eklenmemiştir.)
operation_type	VARCHAR(10)	Yapılan işlemin türünü belirtir. Alabileceği değerler: 'INSERT', 'UPDATE', 'DELETE'.
old_data	JSONB	UPDATE ve DELETE işlemlerinde, verinin değişmeden önceki hali burada JSON formatında saklanır.
new_data	JSONB	INSERT ve UPDATE işlemlerinde, verinin yeni hali burada JSON formatında saklanır.
changed_at	TIMESTAMPTZ	Değişikliğin yapıldığı anın zaman damgasıdır (Default: NOW).
is_processed	BOOLEAN	Verinin MongoDB'ye aktarılıp aktarılmadığını kontrol eden bayraktır (True/False). Varsayılan False olarak gelir.

Neden JSONB Kullanıldı? Log tablosunda old_data ve new_data alanları için PostgreSQL'in JSONB veri tipi tercih edilmiştir. Bunun iki temel nedeni vardır:

1. **Esneklik:** Orders tablosuna ileride yeni bir sütun eklense bile (örneğin "kargo_no"), log tablosunun şemasını değiştirmeye gerek kalmadan tüm satır JSON olarak kaydedilebilir.
2. **NoSQL Uyumu:** Veriler zaten JSON formatında tutulduğu için, MongoDB'ye (Document-Based) aktarım sırasında format dönüşümü maliyeti minimuma indirilmiştir.

2. CDC YAKLAŞIMI VE KULLANILAN YÖNTEM

Bu projede veri değişikliklerini yakalamak ve aktarmak için literatürde "Table-Based / Trigger-Based CDC" (Tablo/Tetikleyici Tabanlı CDC) olarak bilinen yaklaşım benimsenmiştir. Veri aktarım mekanizması olarak ise "Polling" (Düzenli Sorgulama) yöntemi kullanılmıştır.

Sistemin çalışma mimarisi aşağıdaki 5 temel adımdan oluşmaktadır:

1. Algılama (Detection) - Trigger Mekanizması PostgreSQL veritabanındaki kaynak tablo (Orders) üzerinde gerçekleşen her türlü INSERT, UPDATE ve DELETE işlemi, veritabanı seviyesinde tanımlanmış bir Trigger (Tetikleyici) tarafından anında yakalanır.Bu yöntemin seçilme nedeni; uygulama katmanından bağımsız olarak, doğrudan veritabanına yapılan müdahalelerin (örneğin manuel SQL sorgularının) de eksiksiz yakalanabilmesidir.

2. Kayıt (Logging) - Delta Tablosu Trigger tarafından yakalanan değişiklikler, Orders_log adı verilen bir ara tabloya (Delta Table) yazılır. Burada "Write-Ahead Log" (WAL) okuma yöntemi yerine log tablosu yönteminin seçilmesinin sebebi, kurulumun daha basit olması ve değişiklik tarihçesinin (Audit Trail) SQL tarafında da sorgulanabilir olmasını sağlamaktır.

3. Okuma ve Aktarım (Polling & Loading) - Python Uygulaması Geliştirilen Python uygulaması (cdc_app.py), belirli periyotlarla (10 saniye) Orders_log tablosunu sorgular (Polling). Sorgu Mantığı: SELECT * FROM Orders_log WHERE is_processed = FALSE sorgusu ile sadece henüz işlenmemiş kayıtlar çekilir. Çekilen veriler, Python içerisinde NoSQL formatına uygun JSON nesnelerine dönüştürülür ve MongoDB veritabanındaki order_change_logs koleksiyonuna yazılır (insert_one).

4. Onaylama (Acknowledgement) Veri tutarlılığını sağlamak amacıyla "At-least-once delivery" (En az bir kez teslim) prensibi uygulanmıştır. MongoDB'ye yazma işlemi başarılı olduktan sonra, Python scripti SQL veritabanına geri dönerek ilgili log kayıtlarının is_processed alanını TRUE olarak günceller. Bu adım, aynı verinin tekrar tekrar işlenmesini engeller.

Özet Akış Şeması:

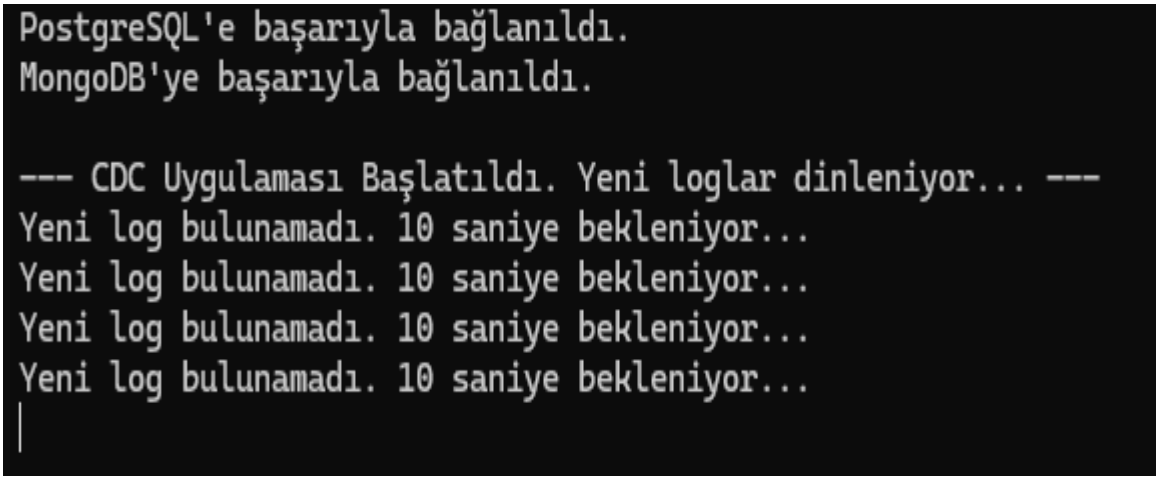
1. Kullanıcı: Orders tablosuna veri girer.
2. SQL Trigger: Veriyi yakalar -> Orders_log tablosuna yazar (is_processed = False).
3. Python Script: Log tablosunu tarar -> Yeni veriyi bulur.
4. MongoDB: Veriyi JSON olarak kaydeder.
5. Python Script: SQL'e döner -> Log kaydını is_processed = True yapar.

3. EKRAN ÇIKTILARI

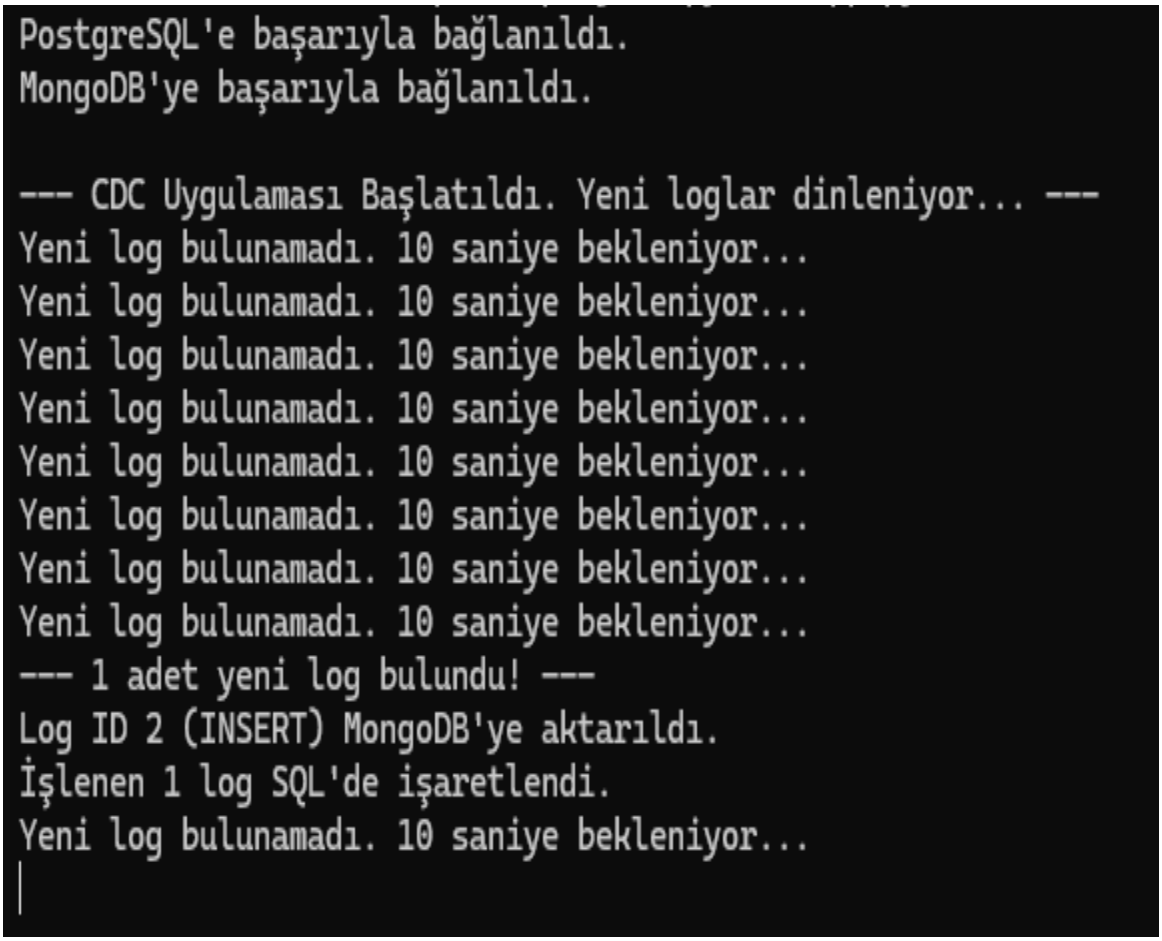
Son 10 değişiklik, En çok işlem gören müşteri ve En çok güncellenen tabloyu çalıştıran Python kodunun ekran çıktısı:

```
================================================
      CDC PROJE ANALİZ RAPORU
================================================

SORGU 1: Son 10 Değişiklik Listeleniyor...
----------------------------------------
1. [2025-12-16 01:41:49] INSERT -> Tablo: Orders
2. [2025-11-19 15:15:40] UPDATE -> Tablo: Orders
3. [2025-11-19 15:15:40] UPDATE -> Tablo: Orders
4. [2025-11-19 15:15:15] INSERT -> Tablo: Orders
5. [2025-11-18 21:15:59] UPDATE -> Tablo: Orders


SORGU 2: En Çok İşlem Gören Müşteri (Customer ID)
----------------------------------------
Müşteri ID   : 1
İşlem Sayısı : 3


SORGU 3: En Çok Güncellenen Tablo
----------------------------------------
Tablo Adı    : Orders
İşlem Sayısı : 5

================================================
```

Cdc_app.py dosyasını çalıştırdığımızda herhangi bir değişiklik yapılmamışken karşımıza çıkan cmd çıktısı:



Cdc_app.py dosyası çalışmaya devam ederken database'de bir güncelleme yaptığımızda çıkan ekran görüntüsü:



4. SONUÇ VE DEĞERLENDİRME

Bu dönem ödevi kapsamında geliştirdiğim CDC (Change Data Capture) prototipi sayesinde, veri mühendisliğinde sıkça kullanılan "veri replikasyonu" kavramını uygulamalı olarak deneyimleme fırsatı buldum. Projeyi tamamladığımda sistemin sorunsuz çalıştığını görmek benim için motive ediciydi. Ancak süreç boyunca yöntemin hem çok güçlü yanlarını hem de dikkat edilmesi gereken zayıf noktalarını bizzat gözlemledim.

Projede Gördüğüm Avantajlar:

- **Veri Kaçağı Sıfıra İniyor:** En çok hoşuma giden özellik, veri yakalama işini Python tarafında değil de doğrudan veritabanının kalbinde (Trigger ile) yapıyor olmamızdı. Uygulamada bir hata olsa bile veya manuel bir SQL sorgusuyla veri silsem bile, trigger bunu affetmiyor ve hemen logluyor. Bu sayede hiçbir değişiklik gözden kaçmıyor.
- **Sistemi Yormadan Raporlama:** Normalde rapor çekmek için ana veritabanını (PostgreSQL) yorardık. Ama bu yapıda verileri MongoDB'ye taşıdığım için, ağır analiz sorgularını (örneğin "en çok sipariş veren müşteri" gibi) MongoDB üzerinde çalıştırdım. Bu mantık, gerçek hayatta ana sistemin performansını korumak için harika bir yöntem.
- **Geçmişe Dönük İz Sürme:** Log tablosunda hem eski veriyi (old_data) hem yeni veriyi (new_data) tuttuğum için, bir hata olduğunda "Bu veri eskiden neymiş?" diye bakabiliyorum. Bu da projeye bir "Audit Log" (Denetim İzi) özelliği kazandırmış oldu.

Karşılaştığım Zorluklar ve Kısıtlar:

- **Anlık Değil, Gecikmeli Aktarım:** Projemde "Polling" (Sorgulama) yöntemini kullandığım için veriler MongoDB'ye anında değil, benim belirlediğim süre (örneğin 10 saniye) sonunda düşüyor. Gerçek zamanlı borsa verisi gibi işler için bu süreyi çok daha kısaltmak gerekebilir, bu da sistemi yorabilir.
- **Trigger Maliyeti:** Her sipariş girişinde arkada sessizce bir trigger çalışıyor. Şu an az veriyle sorun yok ama milyonlarca sipariş alan bir sistemde, her işlemde veritabanına ekstra bir yazma yükü bindirmek performansı düşürebilir.
- **Şema Değişikliği Derdi:** PostgreSQL tarafında tablo yapısını değiştirirsem (örneğin yeni bir sütun eklersem), trigger fonksiyonunu da güncellemem gerekebiliyor. Bu da bakım maliyetini biraz artırıyor.

Özetle; Bu proje bana SQL ve NoSQL dünyalarının birbirine rakip değil, tamamlayıcı olduğunu öğretti. İlişkisel veritabanının tutarlılığı ile NoSQL'in esnekliğini birleştirerek modern bir veri mimarisi kurmuş oldum.