

Satınalma Verisi için ETL Pipeline: ERP'den Güvenilir Harcama Analitiğine

Enes Utku Karadağ · Veri Mühendisi ve Endüstri Mühendisi · enesutkukaradag.com

Satınalma ekipleri her gün veri üretir: talepler, siparişler, faturalar, teslimatlar. Bu veri çoğu zaman ERP'den alınan Excel dosyalarında, farklı para birimlerinde ve tutarsız tedarikçi adlarıyla durur. Bu yazıda, bu dağınık veriyi her sabah otomatik olarak güvenilir bir harcama analitiği tablosuna dönüştüren bir ETL pipeline'ını adım adım anlatıyorum.

Problem: aynı tedarikçi, beş farklı ad

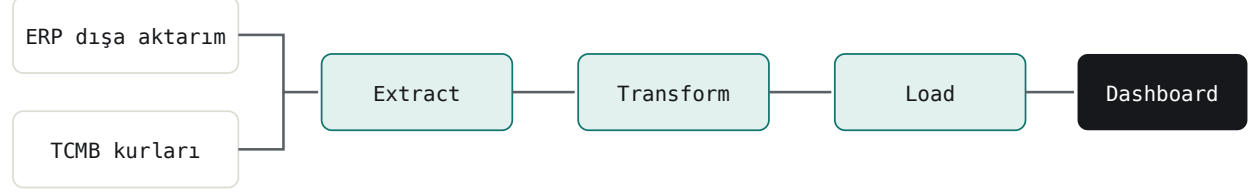
Bir endüstri mühendisi olarak satınalma verisiyle ilk çalıştığımda en basit soruya bile hızlı cevap veremediğimizi gördüm: "Geçen yıl en çok hangi tedarikçiye ne kadar harcadık?" Cevap basit bir toplam olmalıydı. Ama veri şu sorunları taşıyordu:

- **Tutarsız ana veri.** Aynı firma "ABC Elektronik A.Ş.", "ABC ELEKTRONIK AS" ve "Abc Elektronik San. Tic." olarak üç ayrı kayıta duruyordu.
- **Çoklu para birimi.** Siparişler TRY, USD ve EUR ile açılıyordu. Toplamı almak için her satırı sipariş tarihindeki kurla çevirmek gerekiyordu.
- **Elle hazırlanan raporlar.** Her ay biri ERP'den dosya alıyor, formülleri kopyalıyor ve pivot tablo kuruyordu. Bir hücre hatası bütün raporu bozabiliyordu.
- **Eksik kategori.** Kalemlerin bir kısmında malzeme grubu boştu. Kategori bazında harcama analizi bu nedenle eksik kalıyordu.

Çözüm daha büyük bir Excel dosyası değildi. Çözüm, tekrarlanabilir, test edilebilir ve her gün kendiliğinden çalışan bir veri akışıydı.

Mimari

Pipeline klasik üç adımdan oluşur. Her adımın tek bir görevi vardır:



Kaynaklar ham katmana alınır, temizlenir ve analitik şemaya yüklenir. Apache Airflow bütün akışı her sabah 06:00'da çalıştırır.

1. **Extract:** ERP'nin satınalma siparişi dışı aktarımını ve TCMB döviz kurlarını alır. Veriyi hiç değiştirmeden ham katmana yazar.
2. **Transform:** Tedarikçi adlarını eşleştirir, para birimlerini TRY'ye çevirir, kategorileri doldurur ve veri kalitesi kontrollerini çalıştırır.
3. **Load:** Temiz veriyi PostgreSQL'deki yıldız şemasına yükler. Aynı gün iki kez çalışsa bile sonuç değişmez.

Araç seçimi bilinçli olarak sadedir: Python (pandas), PostgreSQL ve Apache Airflow. Bu yığın küçük bir ekipte kolayca bakılabilir ve ileride dbt ya da bir bulut veri ambarına taşınabilir.

1. Extract: veriyi değiştirmeden al

İlk kural: ham veriyi asla yerinde düzeltme. Ham dosyayı tarih damgasıyla saklarsanız, bir hata çıktığında pipeline'ı geçmiş bir gün için yeniden çalıştırabilirsiniz.

```
from datetime import date
from pathlib import Path
import pandas as pd

RAW = Path("data/raw")

def extract_purchase_orders(export_file: Path, run_date: date) -> Path:
    df = pd.read_excel(export_file, dtype=str) # her şeyi metin olarak oku, tipleri sonra belirle
    df["_loaded_at"] = pd.Timestamp.now(tz="Europe/Istanbul")
    out = RAW / f"purchase_orders/{run_date:%Y-%m-%d}.parquet"
    out.parent.mkdir(parents=True, exist_ok=True)
    df.to_parquet(out, index=False)
    return out
```

Döviz kurları için TCMB'nin günlük XML dosyasını kullanıyorum. Hafta sonu ve resmi tatillerde dosya yayınlanmaz. Bu nedenle fonksiyon geriye doğru ilk iş gününü arar:

```

import xml.etree.ElementTree as ET
from datetime import timedelta
import requests

def tcmb_rates(day: date, max_back: int = 7) -> dict[str, float]:
    for i in range(max_back):
        d = day - timedelta(days=i)
        url = f"https://www.tcmb.gov.tr/kurlar/{d:%Y%m}/{d:%d%m%Y}.xml"
        r = requests.get(url, timeout=10)
        if r.status_code == 200:
            root = ET.fromstring(r.content)
            return {
                c.get("CurrencyCode"): float(c.findtext("ForexSelling"))
                for c in root.findall("Currency")
                if c.findtext("ForexSelling")
            } | {"TRY": 1.0}
    raise RuntimeError(f"{day} için {max_back} gün içinde kur bulunamadı")

```

2. Transform: işin asıl yeri

Tedarikçi adlarını eşleştirme

Harcama analizinin doğruluğu, tedarikçi eşleştirmesinin doğruluğuna bağlıdır. Önce adları kurala göre normalize ediyorum. Sonra kalan farkları bulanık eşleştirme (fuzzy matching) ile bir ana tedarikçi listesine bağlıyorum.

```

import re
from rapidfuzz import process, fuzz

TR = str.maketrans("çğİöşüÇĞİÖŞÜ", "cgiosuCGIOSU")
SUFFIXES = r"\b(a\.?s|ltd|sti|san|tic|ve|limited|sirketi)\b\."

def normalize(name: str) -> str:
    n = name.translate(TR).lower()
    n = re.sub(SUFFIXES, " ", n)
    n = re.sub(r"^[a-z0-9 ]", " ", n)
    return " ".join(n.split())

def match_supplier(name: str, master: dict[str, str], threshold: int = 90) -> str | None:
    """master: normalize edilmiş ad -> tedarikçi kimliği"""
    hit = process.extractOne(normalize(name), master.keys(), scorer=fuzz.token_sort_ratio)
    if hit and hit[1] >= threshold:
        return master[hit[0]]
    return None # eşleşmeyen kayıt sessizce kaybolmaz, inceleme listesine gider

```

Eşik değeri önemli bir karardır. Eşik çok düşük olursa iki farklı firma birleşir. Çok yüksek olursa aynı firma ikiye bölünür. Ben eşleşmeyen kayıtları otomatik birleştirmek yerine haftalık bir inceleme

listesine yazıyorum. Satınalma ekibi bu listeyi onaylar ve ana listeye ekler. Böylece sistem her hafta biraz daha akıllı hale gelir.

Para birimini çevirme

Her sipariş satırı, *sipariş tarihindeki* kurla TRY'ye çevrilir. Bugünün kuruyla çevirmek, geçmiş harcamayı olduğundan farklı gösterir.

```
def to_try(lines: pd.DataFrame) -> pd.DataFrame:
    rates = {d: tcmb_rates(d) for d in lines["order_date"].dt.date.unique()}
    lines["fx_rate"] = [rates[d][c] for d, c in zip(lines["order_date"].dt.date, lines["currency"])]
    lines["amount_try"] = (lines["quantity"] * lines["unit_price"] * lines["fx_rate"]).round(2)
    return lines
```

Veri kalitesi kontrolleri

Pipeline hatalı veriyi yükleyeceğine durmalıdır. Kontroller açık ve okunabilir olmalıdır:

```
CHECKS = {
    "sipariş numarası boş olamaz": lambda df: df["po_number"].notna().all(),
    "satır tekrarı olamaz": lambda df: not df.duplicated(["po_number", "po_line"]).any(),
    "miktar pozitif olmalı": lambda df: (df["quantity"] > 0).all(),
    "tutar TRY'ye çevrilmiş olmalı": lambda df: df["amount_try"].notna().all(),
    "eşleşmeyen tedarikçi %2'den az": lambda df: df["supplier_id"].isna().mean() < 0.02,
}

def validate(df: pd.DataFrame) -> None:
    failed = [name for name, check in CHECKS.items() if not check(df)]
    if failed:
        raise ValueError("Veri kalitesi kontrolü başarısız: " + "; ".join(failed))
```

3. Load: yıldız şeması ve tekrar çalıştırılabilirlik

Analitik katmanda bir olgu tablosu ve onu çevreleyen boyut tabloları kullanıyorum. Bu yapı hem SQL sorgularını hem de Power BI modelini basitleştirir.

```

CREATE TABLE dim_supplier (
    supplier_id    text PRIMARY KEY,
    supplier_name  text NOT NULL,
    country        text,
    is_local       boolean
);

CREATE TABLE dim_category (
    category_id    text PRIMARY KEY,
    category_name  text NOT NULL,
    parent_name    text
);

CREATE TABLE fact_po_line (
    po_number      text,
    po_line        int,
    order_date     date NOT NULL,
    supplier_id    text REFERENCES dim_supplier,
    category_id    text REFERENCES dim_category,
    quantity       numeric(18,3) NOT NULL,
    unit_price     numeric(18,4) NOT NULL,
    currency       char(3) NOT NULL,
    fx_rate        numeric(18,6) NOT NULL,
    amount_try     numeric(18,2) NOT NULL,
    promised_date  date,
    received_date  date,
    PRIMARY KEY (po_number, po_line)
);

```

Yükleme işlemi *idempotent* olmalıdır: pipeline aynı gün iki kez çalışırsa veri iki kez yazılmamalıdır. Bunun için PostgreSQL'in `ON CONFLICT` yapısını kullanıyorum. Yüklemeye önce geçici bir tabloya yapılır, sonra tek bir işlemde ana tabloya aktarılır:

```

INSERT INTO fact_po_line AS f
SELECT * FROM stg_po_line
ON CONFLICT (po_number, po_line) DO UPDATE SET
    quantity      = EXCLUDED.quantity,
    unit_price    = EXCLUDED.unit_price,
    amount_try    = EXCLUDED.amount_try,
    received_date = EXCLUDED.received_date
WHERE (f.quantity, f.unit_price, f.received_date)
      IS DISTINCT FROM (EXCLUDED.quantity, EXCLUDED.unit_price, EXCLUDED.received_date);

```

Orkestrasyon: Airflow ile her sabah

Üç adımı Airflow'da bir DAG olarak bağlıyorum. Bir adım hata verirse sonraki adımlar çalışmaz ve ekip bir uyarı alır. Böylece kimse yarım yüklenmiş bir raporu görmez.

```

from airflow.decorators import dag, task
import pendulum

@dag(schedule="0 6 * * 1-5", start_date=pendulum.datetime(2025, 1, 1, tz="Europe/Istanbul"),
      catchup=False, tags=["procurement"])
def procurement_spend():

    @task(retries=2)
    def extract(ds=None) -> str:
        return str(extract_purchase_orders(latest_export(), date.fromisoformat(ds)))

    @task
    def transform(raw_path: str) -> str:
        df = pd.read_parquet(raw_path)
        df = to_try(assign_suppliers(cast_types(df)))
        validate(df)
        out = raw_path.replace("/raw/", "/clean/")
        df.to_parquet(out, index=False)
        return out

    @task
    def load(clean_path: str) -> None:
        upsert_fact_po_line(pd.read_parquet(clean_path))

    load(transform(extract()))

procurement_spend()

```

Pipeline'dan çıkan metrikler

Temiz bir olgu tablosu olunca, eskiden günler alan sorular birkaç satır SQL ile cevaplanır:

- **Tedarikçi bazında harcama ve yoğunlaşma:** Harcamanın yüzde kaçını ilk 10 tedarikçiye?
- **Birim fiyat değişimi:** Aynı malzemenin fiyatı son 12 ayda nasıl değişti?
- **Zamanında teslim oranı:** Hangi tedarikçi söz verdiği tarihte teslim ediyor?
- **Yerli tedarik payı:** Harcamanın ne kadarını yerli tedarikçilere gidiyor? Bu metrik savunma sanayii projelerinde özellikle önemlidir.

```
-- Tedarikçi bazında zamanında teslim oranı (son 12 ay)
SELECT s.supplier_name,
       count(*) AS lines,
       round(avg((f.received_date <= f.promised_date)::int) * 100, 1) AS on_time_pct,
       sum(f.amount_try) AS spend_try
FROM fact_po_line f
JOIN dim_supplier s USING (supplier_id)
WHERE f.received_date IS NOT NULL
      AND f.order_date >= current_date - interval '12 months'
GROUP BY s.supplier_name
HAVING count(*) >= 10
ORDER BY spend_try DESC;
```

Öğrendiklerim

1. **Ana veri, koddan önce gelir.** En iyi pipeline bile kötü tedarikçi ana verisini düzeltemez. Eşleştirme ve inceleme süreci, teknik kısım kadar önemlidir.
2. **Ham veriyi sakla.** Ham katman, geçmiş bir günü yeniden çalıştırmayı ve hatanın kaynağını bulmayı mümkün kılar.
3. **Kontroller, sessiz hatalardan iyidir.** Yanlış bir rapor, hiç olmayan bir rapordan daha zararlıdır. Pipeline şüpheli veride durmalıdır.
4. **Endüstri mühendisliği bakışı fark yaratır.** Süreci uçtan uca görmek, hangi metriğin karar için gerçekten önemli olduğunu seçmeyi kolaylaştırır. Veri mühendisliği bu bakışı ölçeklenebilir hale getirir.

Bir sonraki adımda dönüşüm katmanını dbt'ye taşımayı ve veri kalitesi testlerini SQL tarafında tanımlamayı planlıyorum. Sorularınız ve önerileriniz için LinkedIn üzerinden bana yazabilirsiniz.

Not: Yazıdaki kod ve tablo yapıları örnek amaçlıdır. Gerçek şirket verisi veya sistem bilgisi içermez.