Dal Database OLTP al Data Warehouse PostgreSQL

Una guida pratica completa per migrare e trasformare un database operazionale in un sistema analitico professionale

Obiettivi del Percorso

Migrazione DB

Imparare a migrare un database OLTP da SQLite a PostgreSQL mantenendo l'integrità dei dati

Schema a Stella

Strutturare un Data Warehouse professionale seguendo il modello dimensionale

Processi ETL

Popolare le dimensioni utilizzando dblink e costruire pipeline di trasformazione dati

Fact Table

Costruire tabelle dei fatti con chiavi surrogate e logiche di lookup avanzate

Il Database di Partenza: TechStore OLTP

Il database TechStore è un sistema OLTP (Online Transaction Processing) completo che gestisce tutte le operazioni di un negozio di tecnologia. Questo database, sviluppato durante il corso ITS, contiene sette tabelle principali che rappresentano l'ecosistema completo delle vendite.

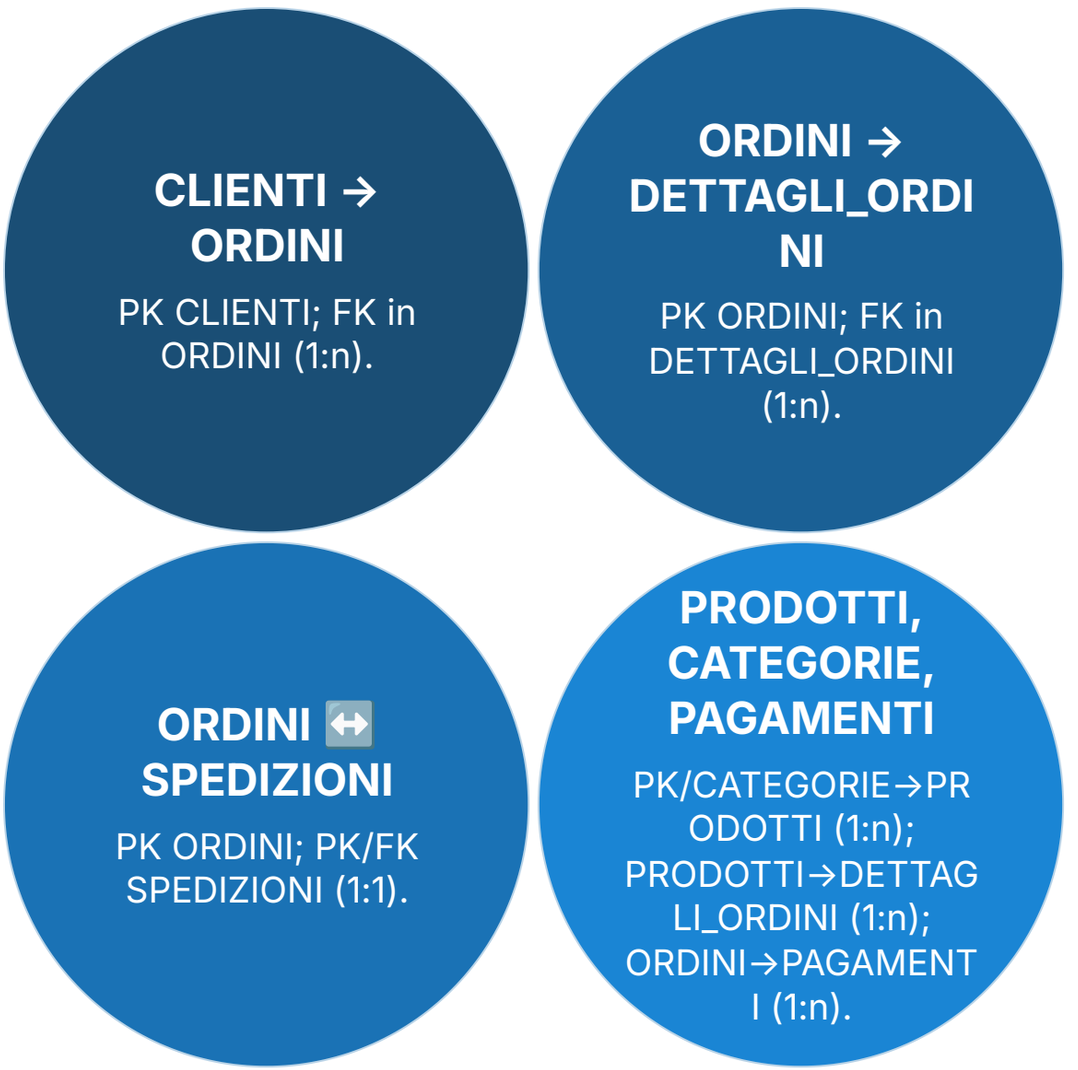
Il formato SQLite è ideale per applicazioni embedded e sviluppo, ma per un Data Warehouse professionale è necessario migrare verso PostgreSQL, un sistema enterprise-grade con funzionalità analitiche avanzate.

Tabelle OLTP

- Clienti
- Prodotti
- Categorie
- Ordini
- Dettagli ordini
- Spedizioni
- Pagamenti

SQLite: Scelta del Database per la prima esercitazione

Per la nostra esercitazione, utilizziamo il database SQLite, un sistema di gestione di database leggero e ideale per lo sviluppo e la prototipazione. Anche se semplice, il suo schema è rappresentativo di un tipico ambiente OLTP (Online Transaction Processing) per un negozio di tecnologia, TechStore.



Questo schema ci fornisce una base solida per comprendere la struttura dei dati transazionali prima di trasformarla per l'analisi.

SQLite: Scelta del Database per la prima esercitazione

SQLite

Un database embedded, leggero e senza server, ideale per applicazioni locali, file-based e progetti di piccole dimensioni che non richiedono concorrenza multi-utente.

- Zero-configurazione, facile da implementare
- Database completo in un singolo file
- Ottimo per prototipazione, test e dispositivi mobili
- Performance limitate per carichi di lavoro intensivi

OLTP vs Data Warehouse

Database OLTP

Ottimizzato per transazioni veloci e frequenti. Schema normalizzato per evitare ridondanze. Focalizzato su INSERT, UPDATE, DELETE. Dati operativi correnti.

Data Warehouse

Ottimizzato per query analitiche complesse. Schema denormalizzato (star/snowflake). Focalizzato su SELECT aggregati. Dati storici e consolidati.

Preparazione Ambiente PostgreSQL

01

Creazione Database OLTP

Primo passo: creare il database PostgreSQL che ospiterà la replica del sistema operazionale

02

Installazione pgloader

Installare lo strumento di migrazione che convertirà automaticamente SQLite in PostgreSQL

03

Creazione Database DW

Creare il database separato che conterrà il Data Warehouse con schema dimensionale

04

Abilitazione dblink

Attivare l'estensione per collegare i due database e permettere l'ETL

Creazione del Database OLTP in PostgreSQL

Il primo passo è creare il database che ospiterà la replica del sistema operazionale. Da pgAdmin, connettersi al server PostgreSQL ed eseguire:

```
CREATE DATABASE techstore;
```

Questo comando crea un database vuoto chiamato `techstore` che sarà pronto a ricevere i dati dal database SQLite originale. È importante mantenere lo stesso nome per facilitare la tracciabilità e la documentazione del progetto.

> **Nota importante:** Assicurarsi che l'utente postgres abbia i privilegi necessari per creare database. In ambiente di produzione, valutare la creazione di utenti dedicati con permessi specifici.

Migrazione con pgloader

pgloader è uno strumento potente che automatizza la conversione tra diversi sistemi di database. Gestisce automaticamente le differenze di sintassi, i tipi di dati e le conversioni necessarie.

Installazione su sistemi Linux/Ubuntu:

```
sudo apt-get install pgloader
```

Comando di migrazione completo:

```
pgloader sqlite:///percorso/techstore_oltp.db \
postgresql://postgres:postgres@localhost:5432/techstore
```

pgloader analizzerà lo schema SQLite, creerà le tabelle corrispondenti in PostgreSQL con i tipi di dati appropriati, e copierà tutti i record mantenendo l'integrità referenziale. Il processo è completamente automatico e gestisce anche la conversione degli indici.

Creazione del Data Warehouse

Ora creiamo un database completamente separato per il Data Warehouse. Questa separazione è una best practice fondamentale nell'architettura dei dati.

```
CREATE DATABASE techstore_dw;
```

La separazione fisica permette di ottimizzare configurazioni, backup e sicurezza in modo indipendente per i due sistemi.

Vantaggi della Separazione

- Performance isolate per query analitiche
- Backup e recovery indipendenti
- Sicurezza e permessi differenziati
- Manutenzione semplificata

Abilitazione Extension dblink

Per iniziare a popolare il Data Warehouse, è fondamentale stabilire una connessione tra il database OLTP (operazionale) e il database DW (analitico). Questo è possibile grazie all'estensione **dblink** di PostgreSQL.

Cos'è dblink?

dblink è un modulo esterno di PostgreSQL che permette a un database PostgreSQL di connettersi a un altro database PostgreSQL e di eseguire query direttamente su di esso, come se i dati fossero locali. È uno strumento cruciale per l'implementazione di processi ETL (Extract, Transform, Load) che necessitano di accedere a dati residenti su database diversi.

> Immagina dblink come un **ponte** logico che collega due database PostgreSQL, consentendo un accesso e uno scambio fluido di informazioni per l'analisi.

Quando e Perché Usare dblink?

Query Incrociate

Quando hai più database separati e necessiti di eseguire query che uniscono dati provenienti da diverse fonti PostgreSQL.

Accesso Remoto

Per leggere dati da un altro database PostgreSQL senza doverli replicare fisicamente, mantenendo le informazioni nella loro posizione originale.

Centralizzazione Dati

Per centralizzare l'accesso ai dati da più sistemi PostgreSQL, creando un punto unico di consultazione per l'analisi o l'integrazione.

Abilitazione Extension dblink

Per poter utilizzare `dblink` e stabilire connessioni tra i database, è necessario abilitare l'estensione nel database da cui si intende avviare le query (in questo caso, il nostro Data Warehouse `techstore_dw`).

```
CREATE EXTENSION dblink;
```

Una volta abilitata, la funzione `dblink()` diventa disponibile, permettendoti di eseguire query su un altro database specificando i dettagli di connessione (host, porta, nome database, utente, password) e la query da eseguire. I risultati vengono restituiti come se fossero una tabella locale, grazie all'alias che definisce la struttura delle colonne.

```
SELECT * FROM dblink('host=localhost dbname=altrodb user=utente password=pass',
                     'SELECT id, nome FROM clienti')
AS risultato(id INT, nome TEXT);
```

Dettagli Importanti

- Funziona solo tra PostgreSQL ↔️ PostgreSQL.
- È necessaria la conoscenza dello schema remoto (tipi e ordine colonne).
- Richiede configurazione di accesso (es. `pg_hba.conf`) per connessioni remote.
- L'esecuzione è sincrona e può bloccare l'applicazione se la query remota è lenta.

Sicurezza: Connessioni Registrate

Per evitare di esporre credenziali sensibili direttamente nelle query, è buona pratica registrare un nome di connessione utilizzando `dblink_connect()`:

```
SELECT dblink_connect('mia_connessione', 'host=... user=... dbname=...');
SELECT * FROM dblink('mia_connessione', 'SELECT ...') ...
```

Esempio Pratico

Consideriamo due database: `magazzino_db` (OLTP) e `fornitori_db` (esterno). Per leggere l'elenco dei fornitori da `fornitori_db`:

```
SELECT * FROM dblink('host=127.0.0.1 dbname=fornitori_db user=postgres password=miapassword',
                     'SELECT id, nome, partita_iva FROM fornitori')
AS fornitori_remoti(id INT, nome TEXT, partita_iva TEXT);
```

Questo comando permette di trattare i dati dei fornitori come se fossero una tabella temporanea nel nostro database corrente, abilitando query e analisi incrociate.

Abilitazione Extension dblink

come visto prima dblink è un'estensione PostgreSQL essenziale per l'ETL che permette di eseguire query su database esterni come se fossero locali. Connettersi al database `techstore_dw` ed eseguire:

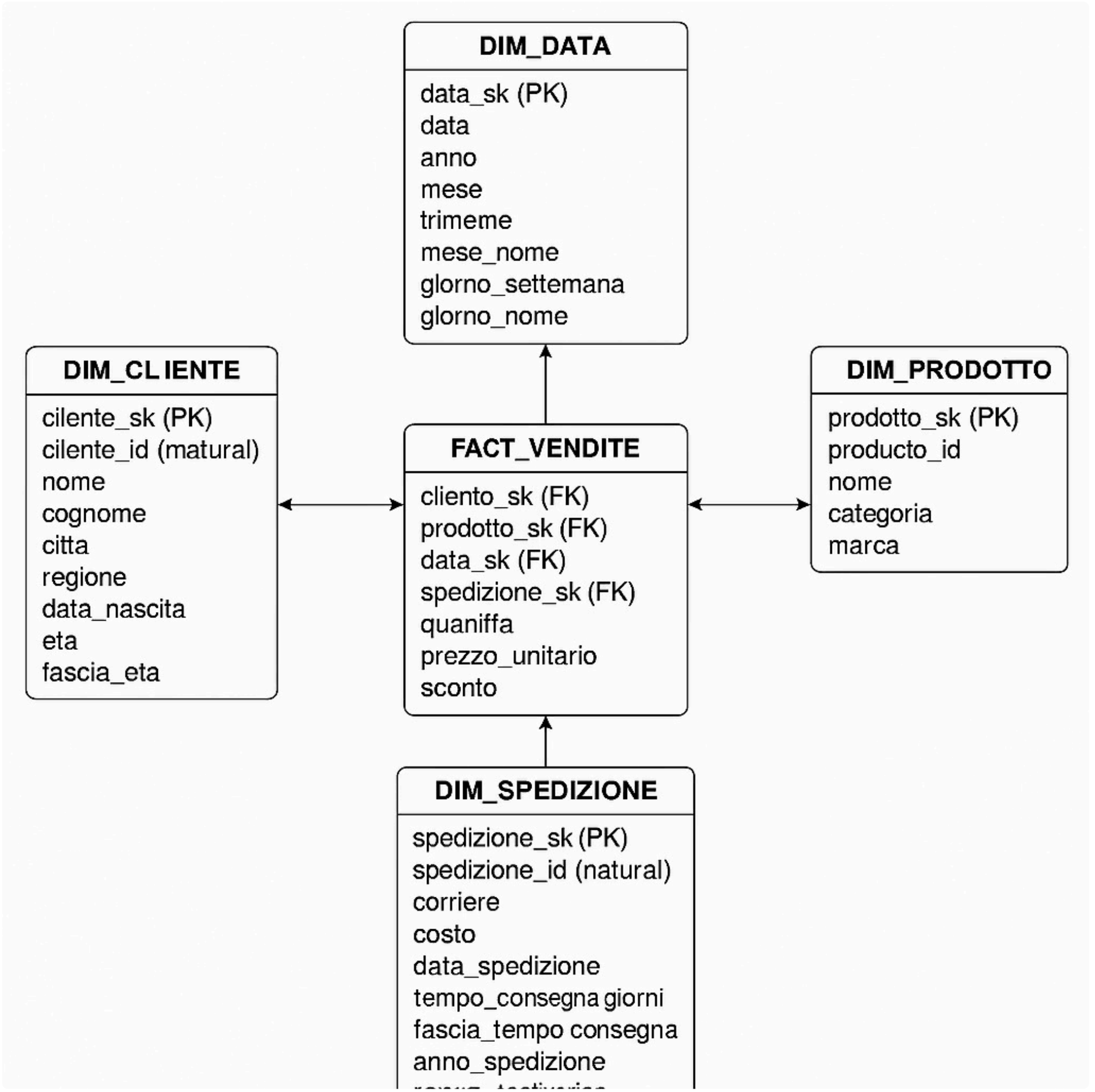
```
CREATE EXTENSION dblink;
```

Questa estensione sarà il ponte tra il database operazionale e il Data Warehouse, permettendo di estrarre dati dal sistema OLTP durante i processi di caricamento delle dimensioni e della fact table.

Importante: dblink deve essere installato nel database di destinazione (techstore_dw), non in quello sorgente. Verificare che l'estensione sia disponibile con SELECT * FROM pg_available_extensions;

Architettura Star Schema

Il nostro Data Warehouse seguirà il modello a stella (star schema), il design più popolare e performante per sistemi analitici. Al centro abbiamo la fact table con le misure quantitative, circondata da quattro dimensioni che forniscono il contesto.



DIM_DATA: La Dimensione Calendario

Ogni Data Warehouse professionale inizia con una dimensione temporale. Questa tabella pre-calcolata permette analisi temporali sofisticate senza overhead computazionale durante le query.

Perché è Fondamentale?

- Analisi per anno, mese, trimestre
- Confronti year-over-year
- Serie storiche e trend
- Calcoli su giorni lavorativi
- Analisi stagionali

La dimensione contiene ogni singolo giorno in un range di anni, pre-calcolando tutti gli attributi temporali necessari per le analisi di business.

Attributi Principali

- `data_sk`: Chiave surrogata
- `data`: Data effettiva
- `anno`, `mese`, `giorno`
- `trimestre`
- `mese_nome`, `giorno_nome`
- giorno_settimana

Creazione e Popolamento DIM_DATA

La creazione della dimensione calendario avviene in due fasi: definizione della struttura e popolamento automatico con generate_series.

```
CREATE TABLE dim_data (
  data_sk SERIAL PRIMARY KEY,
  data DATE UNIQUE,
  anno INT,
  mese INT,
  giorno INT,
  trimestre INT,
  mese_nome TEXT,
  giorno_settimana INT,
  giorno_nome TEXT
);
```

Il popolamento genera automaticamente tutti i giorni dal 2020 al 2026, calcolando tutti gli attributi derivati:

```
INSERT INTO dim_data (data, anno, mese, giorno, trimestre,
 mese_nome, giorno_settimana, giorno_nome)
SELECT
 d,
 EXTRACT(YEAR FROM d),
 EXTRACT(MONTH FROM d),
 EXTRACT(DAY FROM d),
 EXTRACT(QUARTER FROM d),
 TO_CHAR(d, 'Month'),
 EXTRACT(DOW FROM d),
 TO_CHAR(d, 'Day')
FROM generate_series('2020-01-01'::date,
 '2026-12-31'::date,
 '1 day'::interval) AS d;
```

Le Chiavi Surrogate: Concetto Fondamentale

Chiavi Naturali (OLTP)

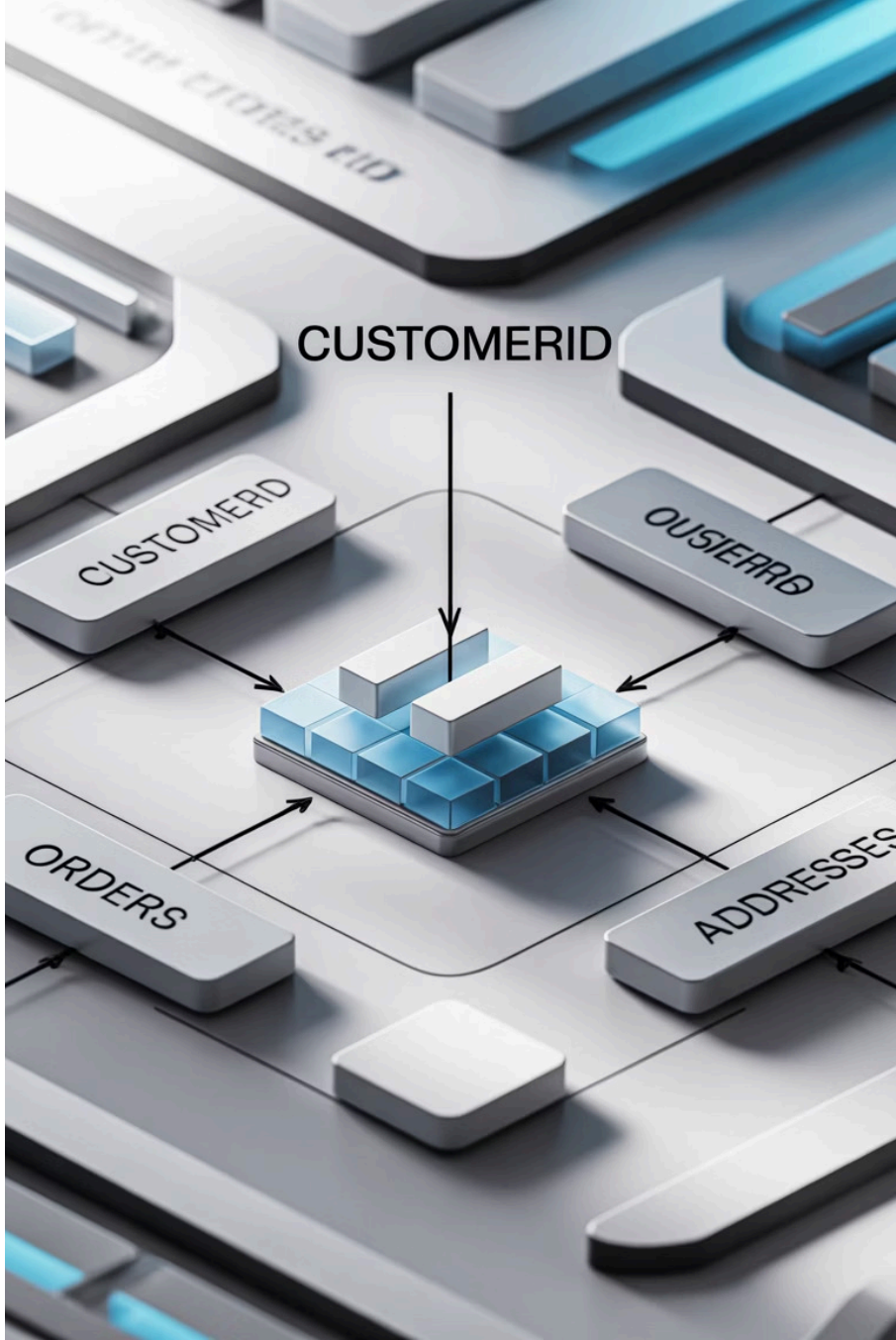
Nel database operazionale usiamo ID naturali: cliente_id, prodotto_id, ecc. Questi identificano le entità nel sistema transazionale.

Chiavi Surrogate (DW)

Nel Data Warehouse generiamo nuove chiavi (cliente_sk, prodotto_sk) indipendenti dal sistema sorgente. Questo garantisce stabilità e performance.

Vantaggi delle Surrogate

Isolamento dai cambiamenti OLTP, gestione di più sorgenti dati, performance ottimali con indici piccoli, supporto per Slowly Changing Dimensions.



DIM_CLIENTE: Progettazione

La dimensione cliente va oltre una semplice copia dell'anagrafica OLTP. Arricchiamo i dati con attributi analitici derivati che supportano segmentazione e analisi di marketing.

```
CREATE TABLE dim_cliente (
  cliente_sk SERIAL PRIMARY KEY,
  cliente_id INT UNIQUE,
  nome TEXT,
  cognome TEXT,
  citta TEXT,
  regione TEXT,
  data_nascita DATE,
  eta INT,
  fascia_eta TEXT
);
```

Notare la presenza sia di `cliente_sk` (chiave surrogata per il DW) che `cliente_id` (chiave naturale dall'OLTP). Quest'ultima mantiene il collegamento con il sistema sorgente ed è dichiarata UNIQUE per garantire che ogni cliente OLTP appaia una sola volta nella dimensione.

ETL: Caricamento DIM_CLIENTE

Fase 1: Import Base

Estraiamo i dati dal database OLTP usando dblink:

```
INSERT INTO dim_cliente
(cliente_id, nome, cognome,
 citta, regione)
SELECT *
FROM dblink(
  'dbname=techstore
   user=postgres
   password=postgres',
  'SELECT cliente_id, nome,
        cognome, citta, regione
   FROM clienti'
) AS t(cliente_id INT,
       nome TEXT,
       cognome TEXT,
       citta TEXT,
       regione TEXT);
```

Fase 2: Arricchimento

Aggiungiamo data di nascita casuale per simulazione:

```
UPDATE dim_cliente
SET data_nascita =
  date '1942-01-01' +
  (random() * 23725)::int;
```

Calcoliamo età e fascia demografica:

```
UPDATE dim_cliente
SET eta = EXTRACT(YEAR
 FROM age(CURRENT_DATE,
 data_nascita));

UPDATE dim_cliente
SET fascia_eta = CASE
 WHEN eta < 25 THEN '18-24'
 WHEN eta BETWEEN 25 AND 39
 THEN '25-39'
 WHEN eta BETWEEN 40 AND 59
 THEN '40-59'
 ELSE '60+'
END;
```

DIM_PRODOTTO: Catalogazione Analitica

La dimensione prodotto integra informazioni da prodotti e categorie, creando una vista denormalizzata ottimale per l'analisi. Questa denormalizzazione è intenzionale nel Data Warehouse.

```
CREATE TABLE dim_prodotto (
  prodotto_sk SERIAL PRIMARY KEY,
  product_id INT UNIQUE,
  nome TEXT,
  categoria TEXT,
  marca TEXT
);
```

Il caricamento richiede un JOIN tra le tabelle prodotti e categorie del sistema OLTP per ottenere il nome della categoria:

```
INSERT INTO dim_prodotto (product_id, nome, categoria, marca)
SELECT *
FROM dblink(
  'dbname=techstore user=postgres password=postgres',
  'SELECT p.prodotto_id,
          p.nome_prodotto,
          c.nome_categoria,
          p.marca
   FROM prodotti p
   JOIN categorie c ON p.categoria_id = c.categoria_id'
) AS t(product_id INT, nome TEXT, categoria TEXT, marca TEXT);
```

Questo approccio elimina la necessità di JOIN durante le query analitiche, migliorando drasticamente le performance.

DIM_SPEDIZIONE: Logistica e KPI

La dimensione spedizione non si limita ai dati grezzi, ma calcola KPI logistici essenziali per valutare l'efficienza operativa e la soddisfazione del cliente.

```
CREATE TABLE dim_spedizione (
  spedizione_sk SERIAL PRIMARY KEY,
  spedizione_id INT UNIQUE,
  corriere TEXT,
  costo NUMERIC(10,2),
  data_spedizione DATE,
  data_consegna DATE,
  tempo_consegna_giorni INT,
  fascia_tempo_consegna TEXT
);
```

Attributi Base

- Informazioni corriere
- Costo di spedizione
- Date spedizione/consegna

Attributi Derivati

- Tempo consegna in giorni
- Fascia temporale per KPI
- Analisi efficienza logistica

ETL DIM_SPEDIZIONE: Calcoli Derivati

Dopo il caricamento base dei dati via dblink, calcoliamo le metriche derivate che forniscono valore analitico:

```
INSERT INTO dim_spedizione
(spedizione_id, corriere, costo_spedizione,
 data_spedizione, data_consegna)
SELECT *
FROM dblink(
  'dbname=techstore user=postgres password=postgres',
  'SELECT s.spedizione_id, s.corriere, s.costo,
         s.data_spedizione, s.data_consegna
   FROM spedizioni s'
) AS t(spedizione_id INT, corriere TEXT, costo NUMERIC,
       data_spedizione DATE, data_consegna DATE);
```

Calcoliamo i giorni di consegna:

```
UPDATE dim_spedizione
SET tempo_consegna_giorni = data_consegna - data_spedizione;
```

Creiamo fasce per analisi KPI:

```
UPDATE dim_spedizione
SET fascia_tempo_consegna = CASE
 WHEN tempo_consegna_giorni <= 2 THEN '1-2 giorni'
 WHEN tempo_consegna_giorni BETWEEN 3 AND 5 THEN '3-5 giorni'
 WHEN tempo_consegna_giorni BETWEEN 6 AND 10 THEN '6-10 giorni'
 ELSE '10+ giorni'
END;
```

FACT_VENDITE: Il Cuore del Data Warehouse

La Fact Table

La fact table è dove risiedono le misure quantitative del business. Ogni riga rappresenta un evento di vendita, collegato alle dimensioni tramite foreign key verso le chiavi surrogate.

```
CREATE TABLE fact_vendite (
  vendita_sk SERIAL PRIMARY KEY,
  cliente_sk INT REFERENCES dim_cliente(cliente_sk),
  prodotto_sk INT REFERENCES dim_prodotto(prodotto_sk),
  data_sk INT REFERENCES dim_data(data_sk),
  spedizione_sk INT REFERENCES dim_spedizione(spedizione_sk),
  quantita INT,
  ordine_id INT,
  prezzo_unitario NUMERIC(10,2),
  sconto NUMERIC(5,2),
  ricavi NUMERIC(12,2)
);
```

Struttura: chiavi surrogate verso le dimensioni + misure additive (quantità, prezzi, ricavi). Le foreign key garantiscono l'integrità referenziale e supportano query efficienti tramite star join.

ETL Fact Table: Architettura Logica

EXTRACT

Estrarre dati grezzi dal database OLTP tramite dblink, unendo ordini, dettagli e spedizioni

TRANSFORM

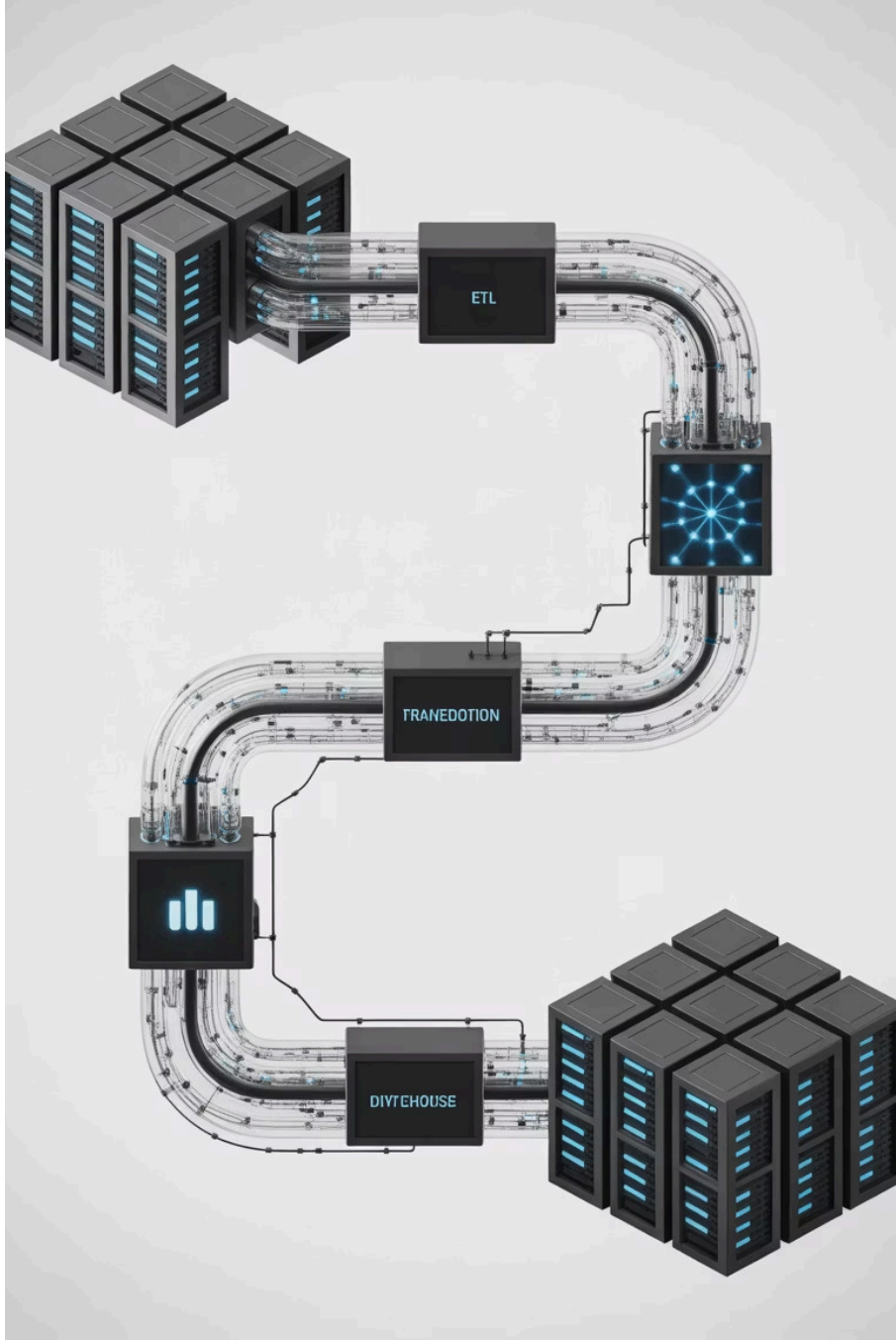
Eseguire lookup sulle dimensioni per mappare chiavi naturali a chiavi surrogate

CALCULATE

Calcolare misure derivate come ricavi applicando formule business (quantità × prezzo × sconto)

LOAD

Caricare i record trasformati nella fact table con tutte le chiavi e misure corrette



Step 1: Estrazione Dati OLTP

Il primo step dell'ETL crea una Common Table Expression (CTE) che estrae e unisce i dati dalle tabelle operazionali:

```
WITH oltp AS (
  SELECT *
  FROM dblink(
    'dbname=techstore user=postgres password=postgres',
    '
    SELECT
      o.ordine_id,
      o.cliente_id,
      o.data_ordine,
      s.spedizione_id,
      d.prodotto_id,
      d.quantita,
      d.prezzo_unitario,
      d.sconto
    FROM ordini o
    JOIN dettagli_ordini d ON o.ordine_id = d.ordine_id
    JOIN spedizioni s ON s.ordine_id = o.ordine_id
    '
  ) AS t(
    ordine_id INT,
    cliente_id INT,
    data_ordine DATE,
    spedizione_id INT,
    prodotto_id INT,
    quantita INT,
    prezzo_unitario NUMERIC,
    sconto NUMERIC
  )
)
```

Questo step crea una vista virtuale chiamata "oltp" contenente tutti i dati necessari per popolare la fact table, già pre-joinati nel sistema sorgente.

Step 2: Lookup delle Chiavi Surrogate

Il cuore della trasformazione: mappiamo ogni chiave naturale OLTP alla corrispondente chiave surrogata del Data Warehouse. Questo avviene tramite JOIN con le tabelle dimensionali:

Mapping delle Chiavi

- `cliente_id → cliente_sk`
- `prodotto_id → prodotto_sk`
- `data_ordine → data_sk`
- `spedizione_id → spedizione_sk`

Codice Lookup

```
FROM oltp o
JOIN dim_cliente dc
  ON dc.cliente_id =
    o.cliente_id
JOIN dim_prodotto dp
  ON dp.product_id =
    o.prodotto_id
JOIN dim_data dd
  ON dd.data =
    o.data_ordine
JOIN dim_spedizione ds
  ON ds.spedizione_id =
    o.spedizione_id
```

Ogni JOIN risolve una chiave naturale alla sua surrogata, creando il collegamento tra fact e dimensioni.

Step 3: Calcolo Misure e Caricamento

L'ultimo step calcola le misure derivate e carica il risultato finale nella fact table:

```
INSERT INTO fact_vendite (
  cliente_sk, prodotto_sk, data_sk, spedizione_sk,
  quantita, prezzo_unitario, sconto, ricavi, ordine_id
)
SELECT
  dc.cliente_sk,
  dp.prodotto_sk,
  dd.data_sk,
  ds.spedizione_sk,
  o.quantita,
  o.prezzo_unitario,
  o.sconto,
  (o.quantita * o.prezzo_unitario * (1 - o.sconto/100.0)) AS ricavi,
  o.ordine_id
FROM oltp o
JOIN dim_cliente dc ON dc.cliente_id = o.cliente_id
JOIN dim_prodotto dp ON dp.product_id = o.prodotto_id
JOIN dim_data dd ON dd.data = o.data_ordine
JOIN dim_spedizione ds ON ds.spedizione_id = o.spedizione_id;
```

La formula dei ricavi applica la logica business: `quantità × prezzo unitario × (1 - sconto%)`, trasformando dati operazionali in metriche analitiche.

Data Warehouse Completo: Panoramica

4

Dimensioni

Tabelle dimensionali completamente popolate e arricchite

1

Fact Table

Tabella dei fatti con tutte le misure e collegamenti

100%

ETL Completo

Pipeline di trasformazione con lookup e calcoli business

Il Data Warehouse è ora operativo e pronto per supportare analisi business avanzate, dashboard interattive e reporting strategico. La struttura star schema garantisce query performanti anche su volumi di dati elevati.

Best Practices e Prossimi Passi

Automatizzazione ETL

Implementare job schedulati per aggiornamenti incrementali del DW, evitando di ricaricare tutto ogni volta

Ottimizzazione Performance

Creare indici sulle foreign key della fact table e sulle colonne più interrogate delle dimensioni

Data Quality

Implementare controlli di qualità e validazione durante l'ETL per garantire integrità dei dati

Visualizzazione

Collegare strumenti BI come Tableau, Power BI o Metabase per creare dashboard e report

SCD Type 2

Implementare Slowly Changing Dimensions per tracciare lo storico delle modifiche nelle dimensioni

Tabelle Aggregate

Creare viste materializzate pre-aggregate per query frequenti su grandi volumi di dati