

# Process Expla



## Transform

Processes and organizes extracted data so it is usable

# ETL e Slowly Changing Dimensions

## Gestione dei cambiamenti nel Data Warehouse con focus su SCD Type 2

Questa sessione vi guiderà attraverso i concetti fondamentali dell'ETL (Extract, Transform, Load) e delle Slowly Changing Dimensions, con particolare attenzione all'implementazione pratica usando Python e PostgreSQL. Imparerete a costruire pipeline ETL robuste e a gestire l'evoluzione dei dati nel tempo mantenendo un audit trail completo.

### Data Warehouse



Architettura e design

### Python + PostgreSQL



Stack tecnologico

# Il Processo ETL

L'ETL (Extract, Transform, Load) rappresenta il cuore pulsante di ogni Data Warehouse moderno. Questo processo trifase permette di integrare dati provenienti da sistemi eterogenei, trasformarli in formati consistenti e caricarli in un repository centralizzato ottimizzato per l'analisi. La qualità dell'ETL determina direttamente la qualità delle decisioni basate sui dati.



## Extract

Estrazione dati da fonti eterogenee

- Database relazionali (OLTP)
- File (CSV, JSON, XML)
- API REST e servizi web
- Stream di dati in tempo reale

## Transform

Trasformazione e pulizia dati

- Data cleansing e validazione
- Standardizzazione formati
- Arricchimento e calcoli derivati
- Aggregazione e denormalizzazione

## Load

Caricamento nel Data Warehouse

- Full load per setup iniziali
- Incremental load per aggiornamenti
- Upsert (insert/update combinati)
- Ottimizzazioni performance e batch

Il successo di un progetto di Data Warehouse dipende dalla robustezza e dall'efficienza del processo ETL. Una pipeline ben progettata deve essere scalabile, manutenibile e in grado di gestire volumi crescenti di dati mantenendo performance elevate e qualità dei dati costante.

## Extract: Estrazione Dati

La fase di estrazione rappresenta il primo step critico dell'ETL. La scelta della strategia e della tecnica corretta dipende da fattori come volume dei dati, frequenza di aggiornamento, impatto sui sistemi sorgente e requisiti di latenza. Una strategia di estrazione ben progettata minimizza l'impatto sui sistemi operazionali mantenendo la completezza e l'accuratezza dei dati.

## Strategie di Estrazione



Aspetto	Full Extraction	Incremental
Cosa estrae	Tutti i dati ogni volta	Solo modifiche recenti
Velocità	Lenta per grandi volumi	Veloce e efficiente
Complessità	Semplice da implementare	Richiede tracking changes
Tracking	Non richiesto	Timestamp o CDC necessari
Carico sistema	Elevato su OLTP	Ridotto e controllato
Quando usare	Piccoli volumi, setup iniziale	Grandi volumi, aggiornamenti frequenti

# Tecniche di Estrazione



## **Direct Database Connection**

Connessione diretta via JDBC/ODBC. Query SQL per estrarre dati. Semplice ma può impattare le performance del sistema sorgente durante l'estrazione.



## File-Based Extraction

Esportazione in file (CSV, JSON, XML). Disaccoppia i sistemi ma richiede gestione dello storage e del trasferimento dei file.



## API-Based Extraction

Chiamate REST API per sistemi SaaS. Flessibile e sicuro ma limitato da rate limits, paginazione e timeout delle API.



## Change Data Capture (CDC)

Cattura automatica modifiche dal transaction log. Fornisce dati real-time con minimo impatto, ma complesso da configurare e gestire.

## Transform: Trasformazione Dati

La fase di trasformazione è dove i dati grezzi diventano informazioni di valore. Questa fase critica include pulizia, standardizzazione, validazione, arricchimento e aggregazione. Trasformazioni ben progettate assicurano che i dati nel Data Warehouse siano accurati, consistenti e pronti per l'analisi. Ogni tipo di trasformazione ha uno scopo specifico e contribuisce alla qualità complessiva dei dati.

# 5. Aggregation

Aggregazioni per analisi: somme, medie, conteggi, min/max per gruppo con raggruppamenti multipli.

```
# Aggregazione vendite per categoria e mese  
vendite_agg = df.groupby(['categoria', 'mese']).agg({  
    'importo': ['sum', 'mean', 'count'],  
    'quantita': 'sum'  
})
```

L'aggregazione è fondamentale per:

## Riassumere grandi dataset

Consente di ridurre la quantità di dati, rendendoli più gestibili e comprensibili per l'analisi a livello superiore.

## Creare viste analitiche

Permette di generare prospettive di business specifiche, ideali per reportistica e dashboard, fornendo insight mirati.

## Migliorare le performance delle query

I dati pre-aggregati riducono il carico di lavoro del database durante le interrogazioni, velocizzando l'accesso alle informazioni.

## 4. Enrichment

L'arricchimento è un processo chiave nella fase di trasformazione, che aggiunge valore ai dati esistenti incorporando informazioni aggiuntive o derivate. Questo include calcoli complessi, segmentazione dei clienti basata su criteri specifici, lookup su tabelle dimensionali per ottenere dettagli contestuali e l'aggiunta di metadati per una migliore gestione e tracciabilità.

```
# Calcolo età da data di nascita  
df['eta'] = (pd.Timestamp.now() -  
    df['data_nascita']).dt.days // 365  
  
# Segmentazione clienti per valore  
df['segmento'] = df['valore'].apply(  
    lambda x: 'Premium' if x>10000  
    else 'Standard')
```

L'arricchimento è fondamentale per elevare il significato dei dati grezzi, trasformandoli in informazioni strategiche. Questo processo:

→ **Aggiunge contesto di business**

Rende i dati più comprensibili e rilevanti per le esigenze aziendali.

→ **Crea metriche derivate**

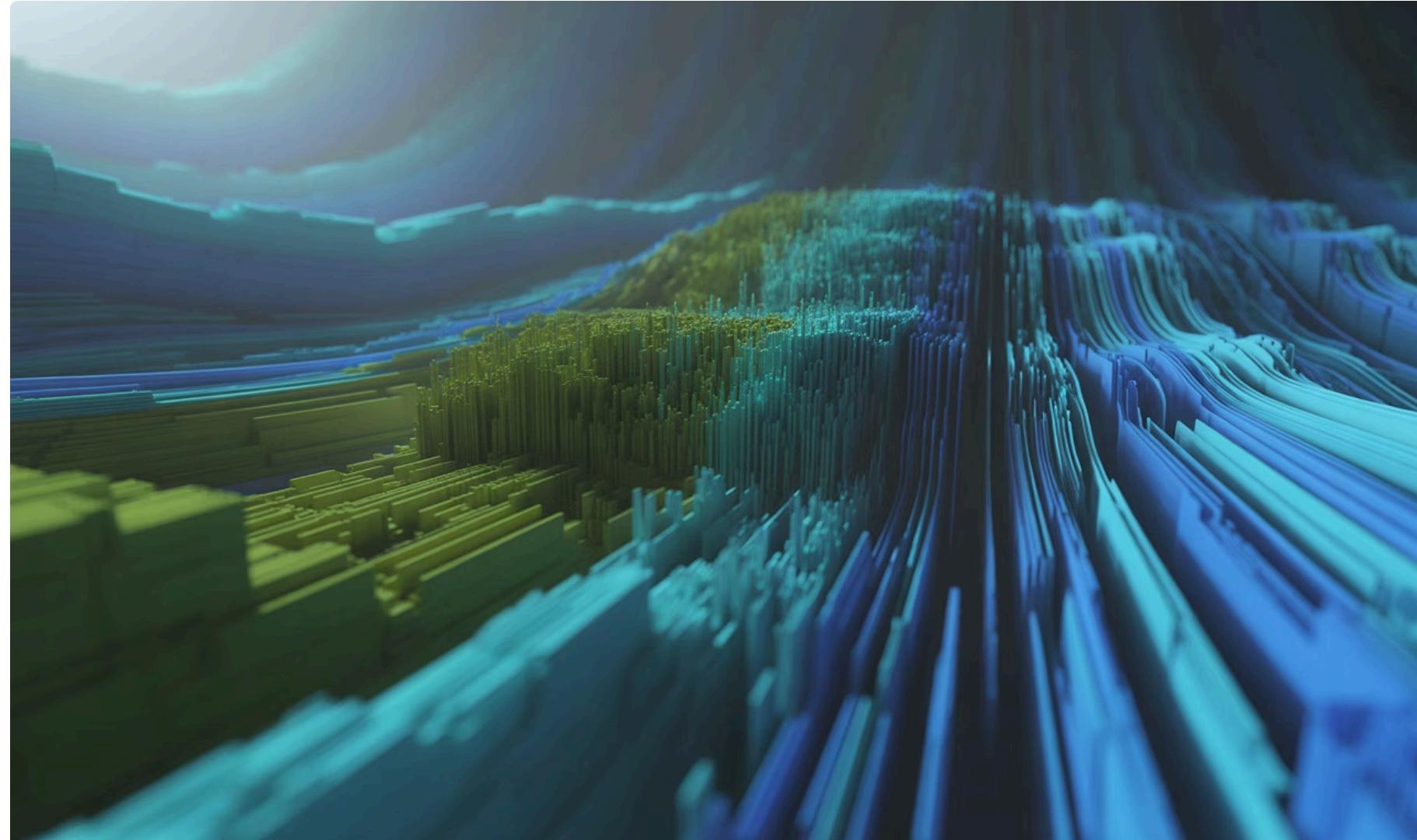
Genera nuovi indicatori e KPI essenziali per l'analisi e il reporting.

→ **Abilita segmentazione avanzata**

Permette di raggruppare i dati in categorie significative per analisi mirate.

→ **Supporta analisi e decisioni**

Fornisce una base dati più ricca e dettagliata per processi decisionali informati.



## 3. Validazione

La validazione dei dati è un passaggio cruciale all'interno della fase di trasformazione, garantendo che i dati siano accurati, consistenti e conformi alle regole di business. Questa fase si concentra sulla verifica di vari aspetti dei dati, inclusi range numerici, formati specifici come gli indirizzi email, l'integrità referenziale tra diverse entità e i vincoli di dominio.

```
# Validazione formato email con regex
df['valid'] = df['email'].str.match(
    r'^[\w\.-]+@[\\w\.-]+\.\w+$')
```

```
# Validazione range età ammissibile
df = df[(df['eta']>=0) & (df['eta']<=120)]
```

La validazione è fondamentale perché rafforza le regole di business, intercetta tempestivamente problemi di qualità dei dati e mantiene l'integrità referenziale, elementi indispensabili per garantire l'affidabilità delle analisi e delle decisioni basate sui dati nel Data Warehouse.

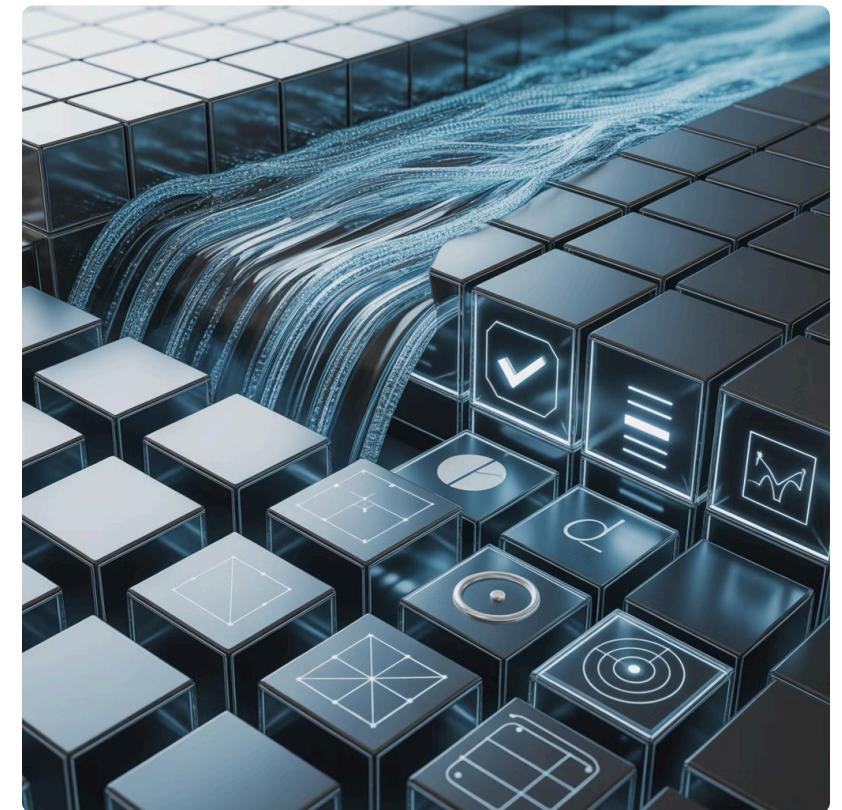
## 2. Standardization

Uniformazione dei formati: date, nomi, codici postali, numeri di telefono per garantire consistenza.

```
# Standardizzazione date in formato ISO  
df['data'] = pd.to_datetime(df['data'])  
  
# Title case per nomi di città  
df['citta'] = df['citta'].str.title()  
  
# Uppercase per codici  
df['cod'] = df['cod'].str.upper()
```

La standardizzazione è fondamentale per diversi motivi chiave nel processo di trasformazione dei dati:

- Permette confronti adeguati tra set di dati diversi.
- Facilita le operazioni di join tra tabelle, essenziali per l'integrazione dei dati.
- Assicura una reportistica coerente e affidabile, migliorando l'accuratezza delle analisi.



# 1. Data Cleansing

Pulizia dei dati: rimozione spazi, gestione valori NULL, eliminazione duplicati e correzione errori tipografici.



```
# Rimozione spazi e duplicati  
df['nome'] = df['nome'].str.strip()  
df.drop_duplicates(subset=['id'])
```

```
# Gestione NULL con valori di default  
df['email'].fillna('unknown@example.com')
```

- La pulizia dei dati è fondamentale per garantire l'accuratezza dei dati, prevenire errori di analisi e migliorare l'affidabilità del data warehouse.

# Data Quality: Le 6 Dimensioni

La qualità dei dati è fondamentale per il successo di qualsiasi progetto di Data Warehouse. Dati di bassa qualità portano a decisioni errate e perdita di fiducia nel sistema. Le sei dimensioni della qualità dei dati forniscono un framework completo per valutare e migliorare la qualità durante tutto il processo ETL. Ogni dimensione affronta un aspetto specifico della qualità e richiede controlli e validazioni dedicate.



## Accuracy

Correttezza e precisione dei dati rispetto alla realtà. I dati devono riflettere accuratamente le entità del mondo reale che rappresentano.

*Esempio:* Email valide con formato corretto, codici fiscali che passano algoritmi di validazione, indirizzi verificabili.



## Consistency

Coerenza dei dati tra fonti e sistemi diversi. Gli stessi dati devono avere gli stessi valori in tutti i sistemi.

*Esempio:* Lo stesso cliente deve avere indirizzo e telefono identici in CRM, ERP e Data Warehouse.



## Validity

Rispetto di regole di business e vincoli di dominio. I dati devono conformarsi alle regole definite per il loro utilizzo.

*Esempio:* Età negative, date future per eventi passati, prezzi negativi o zero violano vincoli di validità.



## Completeness

Assenza di valori mancanti nei campi obbligatori. Tutti gli attributi necessari per l'analisi devono essere presenti.

*Esempio:* NULL in campi critici come cliente\_id, data\_ordine o importo\_vendita compromettono le analisi.



## Timeliness

Aggiornamento tempestivo e disponibilità dei dati quando necessario. I dati devono essere freschi e rilevanti per le decisioni.

*Esempio:* Dati di vendita del giorno precedente non ancora caricati al mattino rendono impossibile il monitoraggio daily.



## Uniqueness

Assenza di duplicati per entità che devono essere uniche. Ogni entità reale deve essere rappresentata una sola volta.

*Esempio:* Lo stesso cliente inserito più volte con piccole variazioni nel nome causa conteggi errati e analisi distorte.

# Load: Caricamento Dati

La fase di caricamento è l'ultimo step del processo ETL e richiede particolare attenzione alle performance e all'integrità dei dati. Una strategia di load inefficiente può creare colli di bottiglia che impattano l'intero pipeline ETL. Le ottimizzazioni discusse qui possono migliorare drasticamente i tempi di caricamento, riducendo le finestre di manutenzione e permettendo aggiornamenti più frequenti del Data Warehouse.

## Strategie di Caricamento



## Ottimizzazioni Performance

**10-100x**

### Batch Insert

Inserimento di record multipli in un'unica transazione invece di singoli INSERT. Commit finale unico.

**2-5x**

### Disable Indexes

Disabilita temporaneamente gli indici durante caricamenti bulk, poi ricostruisce una volta terminato.

**5-10x**

### COPY Command

Usa il comando COPY nativo di PostgreSQL per caricamenti bulk ultra-veloci da file o buffer in memoria.

**3-8x**

### Parallel Loading

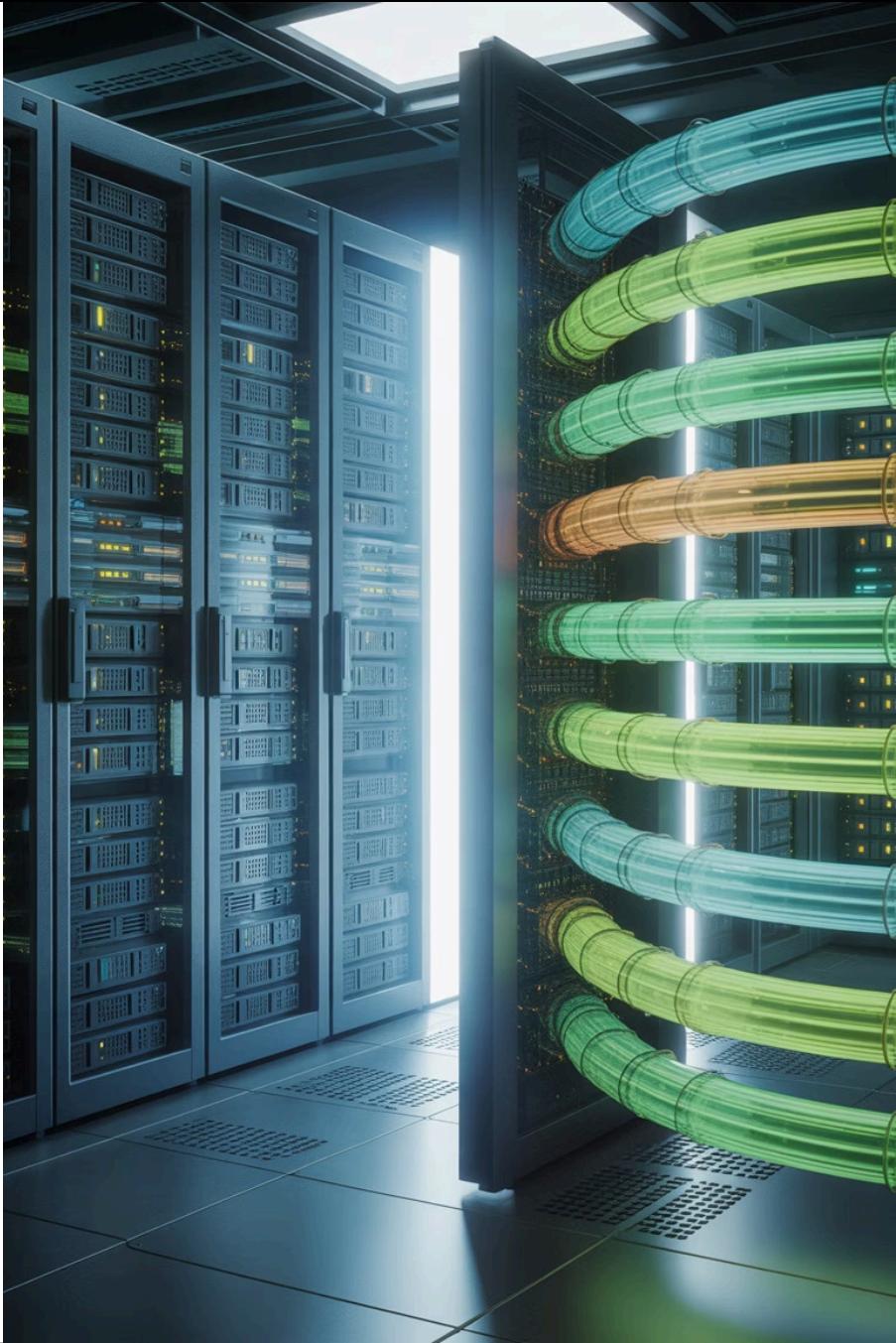
Carica partizioni diverse della tabella in parallelo su più core CPU per massimizzare il throughput.

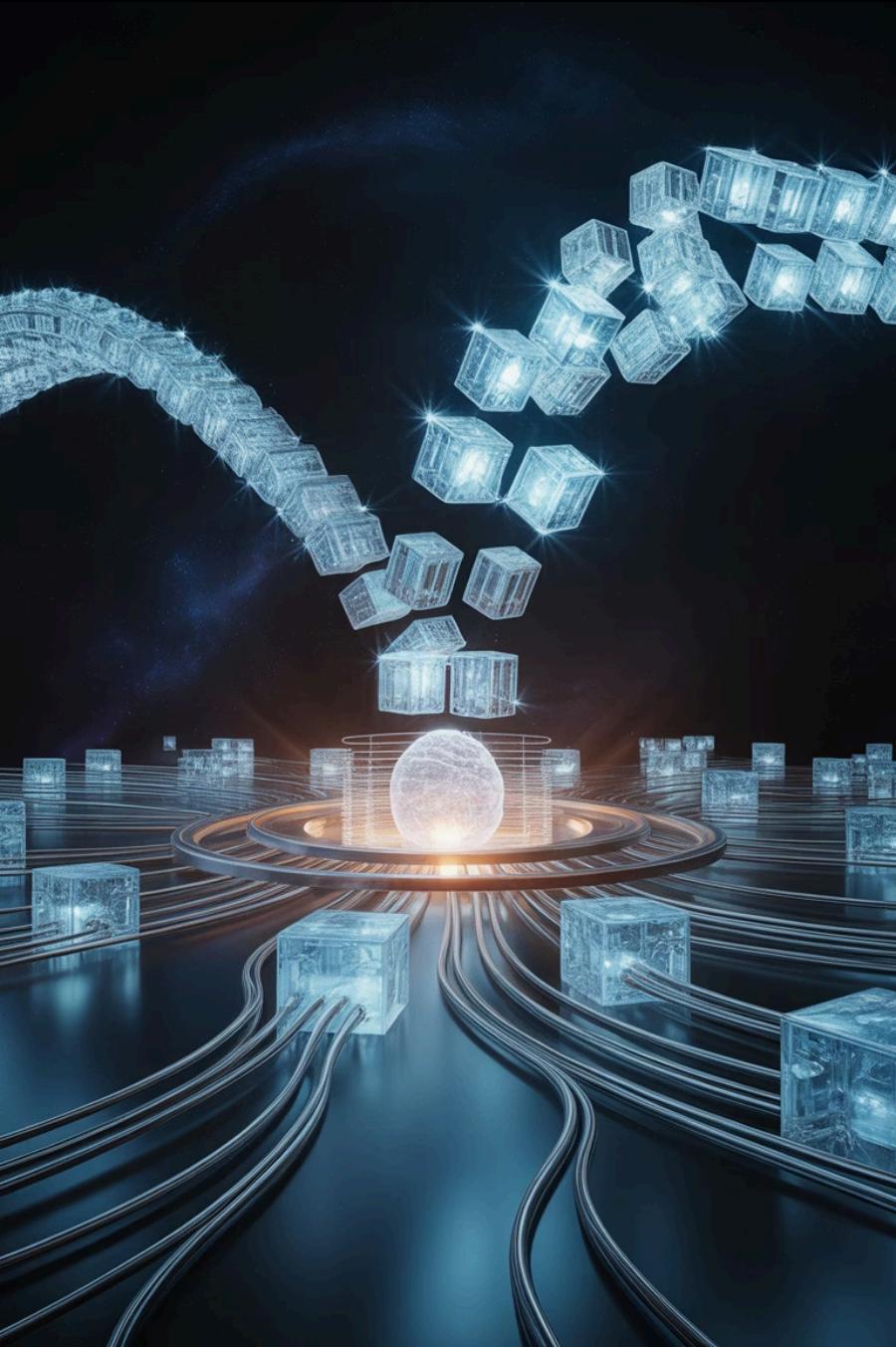
- **Pro Tip:** Combina multiple ottimizzazioni per ottenere speedup moltiplicativi. Ad esempio, batch insert + disable indexes può dare speedup di 50-500x rispetto a singoli INSERT con indici attivi.

## Full Load: Caricamento Completo

La strategia **Full Load** è il metodo più diretto per caricare i dati in un Data Warehouse. Implica la cancellazione completa del contenuto di una tabella di destinazione e il ricaricamento di tutti i dati sorgente da zero. Sebbene sia **semplice da implementare**, questa metodologia può essere molto **inefficiente** e richiede tempi lunghi per dataset di grandi dimensioni, rendendola adatta solo per tabelle piccole o aggiornamenti poco frequenti.

```
TRUNCATE TABLE dim_prodotto;  
INSERT INTO dim_prodotto  
SELECT * FROM staging.prodotto;
```





## Incremental Load: Caricamento Incrementale

La strategia di **Incremental Load** aggiorna il Data Warehouse caricando **solo i record nuovi o modificati** rispetto all'ultimo ciclo. Questa tecnica è cruciale per i sistemi con grandi volumi di dati, dove un full load sarebbe troppo costoso in termini di tempo e risorse. Garantisce aggiornamenti frequenti ed efficienti, mantenendo il Data Warehouse sempre rilevante.

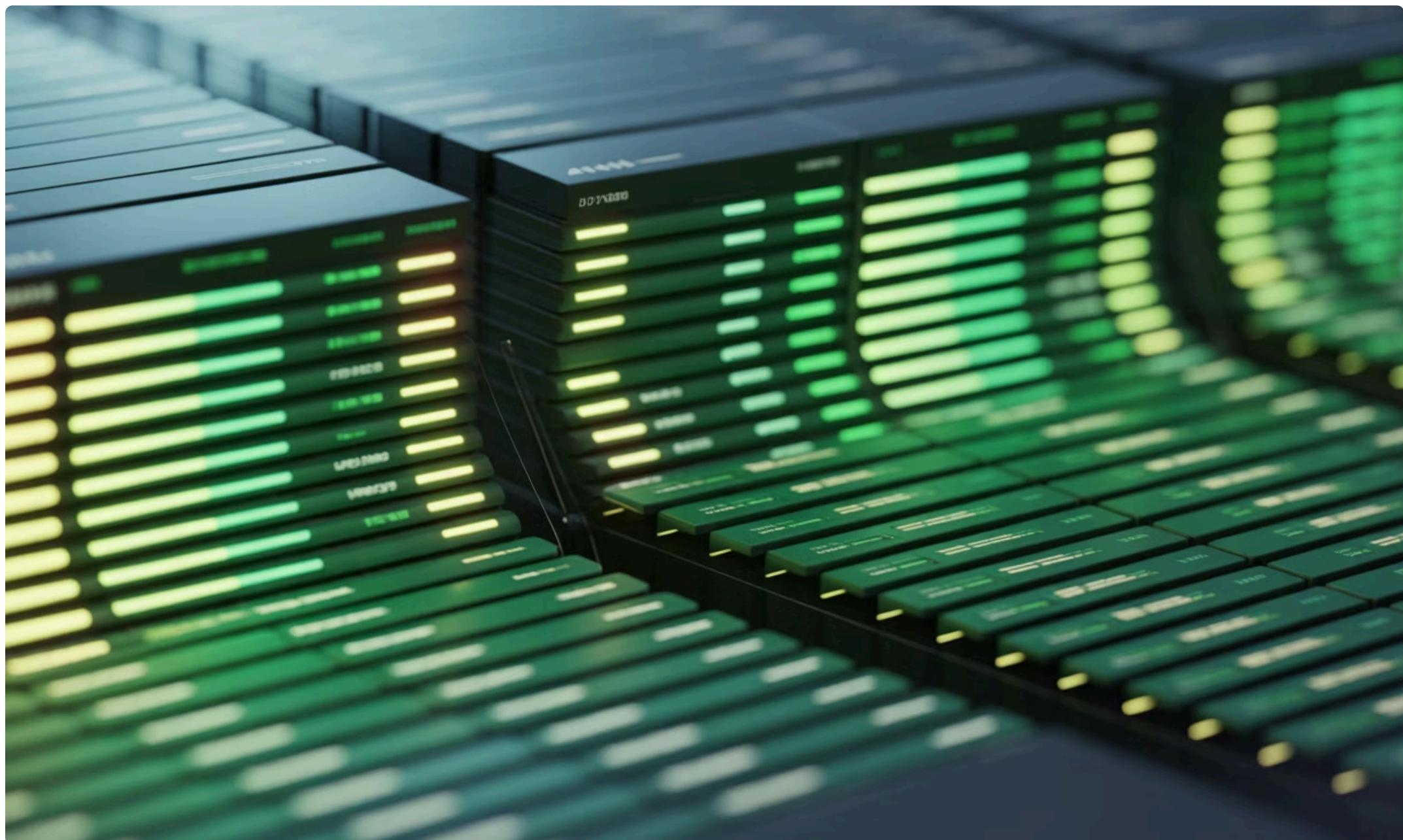
```
INSERT INTO dim_prodotto  
SELECT * FROM staging.prodotto  
WHERE modified_date > last_run;
```

- L'efficacia dell'Incremental Load dipende dalla capacità del sistema sorgente di tracciare le modifiche (e.g., tramite timestamp, versioning o log di transazione).

# Upsert (Insert/Update): La Migliore Pratica

La strategia **Upsert**, una combinazione di "Update" e "Insert", rappresenta la soluzione più flessibile ed efficiente per il caricamento dati nella maggior parte degli scenari ETL. Essa permette di **inserire nuovi record** quando non esistono nel Data Warehouse e di **aggiornare i record esistenti** in caso di modifiche. Questo approccio è fondamentale per mantenere l'accuratezza e la coerenza dei dati nel tempo senza la necessità di ricaricare l'intera tabella.

```
INSERT INTO dim_prodotto (prodotto_id, nome, prezzo_unitario, data_aggiornamento)
VALUES (:prodotto_id, :nome, :prezzo_unitario, NOW())
ON CONFLICT (prodotto_id)
DO UPDATE SET
    nome = EXCLUDED.nome,
    prezzo_unitario = EXCLUDED.prezzo_unitario,
    data_aggiornamento = NOW();
```



## Introduzione: Slowly Changing Dimensions (SCD)

Le **Slowly Changing Dimensions (SCD)** sono un concetto fondamentale nei Data Warehouse. Servono a gestire l'evoluzione nel tempo delle informazioni anagrafiche (clienti, prodotti, dipendenti, aziende) che risiedono nelle tabelle di dimensione.

"Se un dato cambia, come lo salvo in modo da mantenere la storia?"

Le dimensioni sono definite "Slowly Changing" perché, a differenza dei fatti che arrivano quotidianamente, i loro attributi cambiano con una frequenza minore. Esempi comuni includono un cliente che cambia indirizzo, un dipendente che assume un nuovo ruolo o un prodotto che viene riclassificato in una nuova categoria.

# Il Problema dei Cambiamenti

Nei sistemi operazionali (OLTP), quando un dato cambia, viene tipicamente sovrascritto con un semplice UPDATE. Questo approccio funziona per le operazioni quotidiane, ma crea seri problemi quando i dati vengono caricati in un Data Warehouse per analisi storiche. La perdita dello storico rende impossibile analizzare trend, comportamenti passati e l'evoluzione delle entità nel tempo. Le Slowly Changing Dimensions (SCD) risolvono questo problema fondamentale.

## Scenario Problema

Mario Rossi è un cliente "Standard" che spende 500€ all'anno. A gennaio 2024, dopo aver superato i 10.000€ di acquisti, viene promosso al segmento "Premium".

 **Prima (Dicembre 2023):**

cliente_id	nome	segmento	valore_annuo
1001	Mario Rossi	Standard	500€

```
UPDATE dim_cliente  
SET segmento = 'Premium',  
    valore_annuo = 12000  
WHERE cliente_id = 1001;
```

 **Dopo (Gennaio 2024):**

cliente_id	nome	segmento	valore_annuo
1001	Mario Rossi	Premium	12,000€

## Cosa Succede con UPDATE Semplice?

### Perdita Storico

Non sappiamo più che Mario era "Standard" prima di gennaio 2024. Lo storico del cliente è completamente perduto.

### Analisi Impossibili

Non possiamo calcolare le vendite per segmento nel 2023. I report retrospettivi sono inaccurati.

### Audit Trail Mancante

Nessuna traccia di quando è cambiato il segmento, chi ha fatto la modifica, o perché.

### Dati Retrospettivi Errati

Report storici mostrano Mario come "Premium" anche nel 2023, falsificando le analisi di trend.

"Senza storico dimensionale, il Data Warehouse perde la sua capacità più preziosa: permettere analisi temporali accurate. Le decisioni basate su dati storici incompleti o errati possono portare a strategie sbagliate e perdita di opportunità di business."

## Tipi di SCD: Panoramica

Le Slowly Changing Dimensions offrono diverse strategie per gestire i cambiamenti nelle dimensioni del Data Warehouse. La scelta del tipo appropriato dipende dai requisiti di business, dal volume dei cambiamenti attesi, dalle necessità di analisi storica e dai vincoli di storage e performance. Comprendere le caratteristiche di ogni tipo è essenziale per progettare un Data Warehouse efficace.

Tipo	Strategia	Quando Usare	Complessità	Storico
Type 0	Retain Original (Immutabile)	Dati che non devono mai cambiare: data di nascita, codice fiscale, ID univoci, timestamp creazione record	Bassa	N/A
Type 1	Overwrite (Sovrascrive)	Correzioni di errori, dati non critici per analisi storiche: indirizzo email, numero telefono, preferenze utente	Bassa	No
Type 2 ★	Add New Row (Nuova riga)	Quando serve storico completo per analisi trend: segmento cliente, prezzo prodotto, categoria, status contratto	Media	Completo
Type 3	Add New Attribute (Nuova colonna)	Tracciare solo il valore precedente, storico limitato: cambio categoria prodotto, riorganizzazione dipartimento	Bassa	Limitato
Type 4	History Table (Tabella separata)	Dimensioni molto grandi con pochi cambiamenti, necessità di separare dati current da historical per performance	Alta	Completo
Type 6	Hybrid (Combinazione 1+2+3)	Scenari complessi che richiedono sia storico completo che accesso rapido a valori correnti e precedenti	Molto Alta	Completo

**Nota:** SCD Type 2 è il più utilizzato nella pratica perché offre il miglior bilanciamento tra completezza dello storico, flessibilità di analisi e complessità implementativa. Rappresenta la scelta di default per la maggior parte degli attributi dimensionali che cambiano nel tempo.

## SCD Type 0 e Type 1

I tipi SCD 0 e 1 rappresentano le strategie più semplici per gestire i cambiamenti dimensionali. Type 0 tratta i dati come completamente immutabili - una volta inseriti, non possono essere modificati. Type 1 invece permette modifiche ma non mantiene alcuno storico, sovrapponendo semplicemente i nuovi valori. Entrambi hanno casi d'uso specifici dove la loro semplicità è un vantaggio.

### Type 0: Retain Original (Immutable)



I dati non cambiano mai. Una volta inseriti nel Data Warehouse, rimangono immutabili per sempre. Qualsiasi tentativo di modifica viene rifiutato dal sistema.

#### Struttura Tabella:

cliente_id	nome	data_nascita	codice_fiscale
1001	Mario Rossi	1980-05-15	RSSMRA80E15...

#### 💡 Quando usare:

- Dati che per natura non devono cambiare: data di nascita, luogo di nascita
- Codici identificativi univoci: codice fiscale, numero passaporto
- Timestamp di creazione record e metadati di sistema
- Dati legali o regolamentati che richiedono immutabilità

#### ✓ Vantaggi

- Semplicità massima: nessuna logica ETL complessa
- Integrità garantita: impossibile alterare i dati
- Performance ottimali: nessun overhead di gestione storico

#### ✗ Svantaggi

- Impossibile correggere errori una volta inseriti
- Non adatto per dati che possono legittimamente cambiare
- Richiede processi di data governance molto rigorosi

### Type 1: Overwrite (Sovrascrive)



I dati vengono sovrascritti quando cambiano. Il valore vecchio è completamente perduto e sostituito dal nuovo. Nessuno storico viene mantenuto.

#### Esempio UPDATE:

cliente_id	nome	email	telefono
1001	Mario Rossi	mario.rossi@nuovo.it	333-1234567

```
UPDATE dim_cliente  
SET email = 'mario.rossi@nuovo.it',  
    telefono = '333-1234567'  
WHERE cliente_id = 1001;
```

#### 💡 Quando usare:

- Correzioni di errori di data entry o tipo
- Dati di contatto che non richiedono storico: email, telefono
- Attributi non rilevanti per analisi temporali
- Dati che cambiano raramente e lo storico non aggiunge valore

#### ✓ Vantaggi

- Implementazione semplicissima: semplice UPDATE
- Nessun aumento di storage nel tempo
- Query sempre sulla versione corrente (nessun filtro)

#### ✗ Svantaggi

- Perdita totale dello storico dei cambiamenti
- Impossibile fare analisi "as-was" nel passato
- Nessun audit trail per compliance o debugging

## SCD Type 2: Add New Row

# Il Più Usato

SCD Type 2 è la strategia più popolare e potente per gestire i cambiamenti dimensionali. Invece di sovrascrivere i dati esistenti, crea una nuova riga per ogni cambiamento, mantenendo intatto lo storico completo. Questo permette analisi sofisticate "point-in-time" e fornisce un audit trail completo di tutti i cambiamenti. È la scelta di default per la maggior parte degli attributi dimensionali che evolvono nel tempo.

### Definizione

Quando un attributo cambia, invece di sovrascrivere (Type 1), si **inserisce una nuova riga** con il nuovo valore, mantenendo la riga vecchia come storico. Ogni riga rappresenta una "versione" della dimensione valida in un periodo temporale specifico.

### Concetto Chiave

Ogni riga rappresenta una **versione temporale** della dimensione valida in un determinato periodo. Metadati aggiuntivi (data\_inizio, data\_fine, is\_current) permettono di tracciare quando ogni versione era attiva e quale rappresenta lo stato corrente dell'entità.



#### Storico Completo

Tutti i valori storici sono preservati per analisi di trend e pattern temporali. Nessun dato viene mai perso o sovrascritto.



#### Audit Trail

Fornisce traccia completa di tutti i cambiamenti con timestamp esatti, essenziale per compliance normativa e debugging.



#### Point-in-Time Analysis

Permette analisi "as-was": come erano esattamente i dati in qualsiasi data del passato, ricostruendo snapshot storici precisi.



#### Compliance

Soddisfa requisiti normativi stringenti come GDPR, SOX, audit finanziari che richiedono storico immutabile e tracciabile.

## SCD Type 2: Struttura Tabella

Per implementare correttamente SCD Type 2, la tabella dimensionale richiede campi metadati specifici oltre agli attributi business standard. Questi campi addizionali permettono di tracciare lo storico, identificare univocamente ogni versione, e determinare quale versione è corrente. La corretta progettazione di questi metadati è cruciale per l'integrità e le performance delle query temporali.

01

### Surrogate Key (SK)

Chiave artificiale univoca per ogni versione della dimensione. Tipicamente un intero auto-incrementante. Esempio: cliente\_sk è diverso per ogni versione dello stesso cliente.

03

### data\_inizio (Start Date)

Timestamp che indica quando questa specifica versione è diventata valida ed effettiva nel sistema. Segna l'inizio del periodo di validità della riga.

05

### is\_current (Flag)

Flag booleano che indica se questa è la versione corrente (TRUE) o una versione storica (FALSE). Facilita le query per ottenere solo i dati attuali.

02

### Natural Key (NK)

Chiave business originale dal sistema sorgente OLTP. Rimane costante tra versioni diverse della stessa entità. Esempio: cliente\_id è lo stesso per tutte le versioni di Mario Rossi.

04

### data\_fine (End Date)

Timestamp che indica quando questa versione è stata sostituita da una nuova versione. NULL per la versione corrente ancora attiva. Segna la fine del periodo di validità.

06

### versione (Optional)

Numero sequenziale che indica quale versione è questa riga (1, 2, 3, ...). Opzionale ma utile per debugging e comprensione dell'evoluzione.

## Esempio Completo: dim\_cliente con SCD Type 2

cliente_sk	cliente_id	nome	segmento	valore_annuo	data_inizio	data_fine	is_current	versione
1	1001	Mario Rossi	Standard	500€	2023-01-01	2024-01-15	FALSE	1
2	1001	Mario Rossi	Premium	12,000€	2024-01-16	NULL	TRUE	2

 **Best Practice:** Usa sempre una Surrogate Key separata come primary key invece della Natural Key. Questo permette multiple versioni della stessa entità business e semplifica le relazioni con le fact table, che referenziano sempre la versione corretta tramite la SK.

## SCD Type 2: Esempio Timeline

Visualizzare l'evoluzione temporale di una dimensione SCD Type 2 aiuta a comprendere come le diverse versioni si susseguono nel tempo. Ogni versione ha un periodo di validità ben definito, e la transizione tra versioni avviene in modo preciso e senza sovrapposizioni. Questo esempio mostra il ciclo di vita completo di un cliente che passa da segmento Standard a Premium.

### Scenario: Evoluzione Cliente TechStore

Mario Rossi è cliente TechStore dal 1 gennaio 2023 con segmento "Standard". Durante il primo anno spende regolarmente circa 500€. Il 16 gennaio 2024, dopo aver superato i 10.000€ di spesa cumulativa, il sistema lo promuove automaticamente al segmento "Premium".



### Versione 1 (Storica)

Campo	Valore
cliente_sk	1
cliente_id	1001
nome	Mario Rossi
segmento	Standard
valore_annuo	500€
data_inizio	2023-01-01
data_fine	2024-01-15
is_current	FALSE
versione	1

Periodo validità: 380 giorni (tutto il 2023)

### Versione 2 (Corrente)

Campo	Valore
cliente_sk	2
cliente_id	1001
nome	Mario Rossi
segmento	Premium
valore_annuo	12,000€
data_inizio	2024-01-16
data_fine	NULL
is_current	TRUE
versione	2

Periodo validità: Aperto (ancora attiva)

## SCD Type 2: Algoritmo di Implementazione

L'implementazione di SCD Type 2 richiede un algoritmo preciso che confronta i dati in ingresso con lo stato corrente della dimensione e gestisce correttamente sia gli inserimenti di nuove entità che gli aggiornamenti di entità esistenti. Questo processo in 5 passi garantisce l'integrità dello storico e la corretta tracciabilità temporale.



### 1. Cerca il Record Corrente

Trova la versione attualmente attiva del record nella tabella dimensionale usando la natural key (chiave business) e il flag is\_current.

```
SELECT * FROM dim_cliente  
WHERE cliente_id = '1001'  
AND is_current = TRUE
```

Se non trovi nessun record, significa che è una nuova entità da inserire (vai al passo 5).



### 2. Confronta gli Attributi

Confronta i valori correnti nella dimensione con i nuovi valori provenienti dal sistema sorgente. Identifica se ci sono cambiamenti negli attributi tracciati.

```
IF (old_segmento != new_segmento OR  
old_valore != new_valore):  
# Cambiamento rilevato!  
process_change()  
ELSE:  
# Nessun cambiamento, skip  
continue
```

Solo gli attributi configurati come "tracked" vengono confrontati. Attributi Type 1 vengono semplicemente aggiornati.



### 3. Chiudi il Record Corrente

Imposta data\_fine alla data odierna (o ieri) e cambia is\_current da TRUE a FALSE per la versione corrente. Questo "chiude" il periodo di validità.

```
UPDATE dim_cliente  
SET data_fine = '2024-01-15',  
is_current = FALSE  
WHERE cliente_sk = 1
```

Usa CURRENT\_DATE - 1 per data\_fine se vuoi che il nuovo record parta da oggi, oppure CURRENT\_DATE se i periodi devono toccarsi.



### 4. Inserisci Nuova Versione

Crea una nuova riga con una nuova surrogate key, i nuovi valori degli attributi, data\_inizio impostata a oggi, data\_fine NULL e is\_current TRUE.

```
INSERT INTO dim_cliente (  
cliente_id, nome, segmento,  
valore_anno, data_inizio,  
data_fine, is_current, versione  
) VALUES (  
'1001', 'Mario Rossi', 'Premium',  
12000, '2024-01-16',  
NULL, TRUE, 2  
)
```



### 5. Aggiorna Fact Table

Le nuove transazioni nella fact table devono referenziare la nuova surrogate key (cliente\_sk = 2). Le transazioni storiche mantengono i loro riferimenti alle vecchie SK.

```
-- Nuove transazioni usano nuovo SK  
INSERT INTO fact_vendite  
(data, cliente_sk, ...)  
VALUES ('2024-01-17', 2, ...)
```

Questo preserva l'integrità referenziale storica: ogni fatto punta alla versione della dimensione valida al momento della transazione.

## SCD Type 2: Query Analitiche

Interrogare correttamente una dimensione SCD Type 2 richiede la comprensione di come filtrare le versioni basandosi sui campi temporali e sui flag. Le tre pattern di query più comuni coprono la maggior parte delle esigenze analitiche: snapshot corrente, point-in-time storico e storico completo di un'entità. Padroneggiare queste tecniche è essenziale per sfruttare appieno il potere delle SCD Type 2.

### 1 Query Versione Corrente

Ottieni solo i record attuali, ignorando tutto lo storico. Questo fornisce uno snapshot della dimensione "oggi", con una sola riga per ogni entità business.

```
SELECT
    cliente_id,
    nome,
    segmento,
    valore_annuo,
    data_inizio
FROM dim_cliente
WHERE is_current = TRUE;
```

-- Risultato: Una riga per cliente (solo versione attuale)

**Caso d'uso:** Dashboard real-time, report operativi, query semplici che non necessitano di contesto storico.

### 2 Query Point-in-Time (Snapshot Storico)

Ricostruisci lo stato della dimensione esattamente come era in una data specifica del passato. Essenziale per analisi "as-was" e comparazioni temporali.

```
SELECT
    cliente_id,
    nome,
    segmento,
    valore_annuo
FROM dim_cliente
WHERE '2023-12-31' BETWEEN data_inizio
    AND COALESCE(data_fine, '9999-12-31');
```

-- Risultato: Una riga per cliente (com'era a quella data)

**Caso d'uso:** Analisi retrospettive, compliance audit, comparazione "before/after", ricostruzione report storici.

**Trucco:** COALESCE(data\_fine, '9999-12-31') gestisce i record correnti (data\_fine NULL) trattandoli come validi fino al futuro lontano.

### 3 Query Storico Completo

Ottieni tutte le versioni storiche di un cliente specifico per analizzare l'evoluzione nel tempo e identificare pattern di comportamento.

```
SELECT
    cliente_sk,
    cliente_id,
    nome,
    segmento,
    valore_annuo,
    data_inizio,
    data_fine,
    is_current,
    versione
FROM dim_cliente
WHERE cliente_id = '1001'
ORDER BY data_inizio;
```

-- Risultato: Tutte le versioni in ordine cronologico

**Caso d'uso:** Analisi evoluzione singolo cliente, debugging problemi dati, audit trail dettagliato, customer journey analysis.

- Performance Tip: Crea indici su (cliente\_id, is\_current) per query correnti e su (data\_inizio, data\_fine) per query point-in-time. Per dimensioni molto grandi, considera la partitioning per data\_inizio per migliorare ulteriormente le performance delle query temporali.

## SCD Type 2: Vantaggi e Svantaggi

Come ogni scelta architettonale, SCD Type 2 presenta trade-off significativi. Comprendere profondamente vantaggi e svantaggi è essenziale per decidere quando utilizzare questa strategia e quando optare per alternative più semplici. La decisione dipende dai requisiti specifici di business, vincoli di storage, competenze del team e complessità accettabile.

### ✓ Vantaggi



### ✗ Svantaggi



#### Storico Completo

Mantiene tutte le versioni storiche dei dati per analisi temporali complete. Nessuna informazione viene mai persa o sovrascritta.



#### Point-in-Time Analysis

Permette di analizzare i dati esattamente come erano in qualsiasi momento del passato. Essenziale per analisi retrospettive accurate.



#### Audit Trail

Fornisce tracciabilità completa di tutti i cambiamenti per conformità normativa, audit finanziari e debugging di problemi storici.



#### Trend Analysis

Abilità analisi sofisticate di trend, pattern e comportamenti nel tempo. Fondamentale per predictive analytics e machine learning.



#### Integrità Referenziale

Le fact table mantengono riferimenti corretti alle versioni storiche tramite surrogate keys, garantendo coerenza temporale perfetta.



#### Compliance

Soddisfa requisiti stringenti di normative come GDPR, SOX, HIPAA che richiedono storico immutabile e audit trail completo.



#### Storage Maggiore

Richiede significativamente più spazio disco per memorizzare tutte le versioni storiche. Per dimensioni con cambiamenti frequenti, lo storage può crescere rapidamente (2-10x rispetto a Type 1).



#### Complessità ETL

Il processo ETL è sostanzialmente più complesso: richiede logica per confrontare versioni, chiudere record correnti e inserire nuove versioni. Più punti di failure potenziali.



#### Performance Query

Query più lente su dimensioni grandi perché devono scansionare più righe (tutte le versioni). Gli indici sono più grandi e complessi. Impatto 2-5x su alcune query.



#### Complessità Query

Le query SQL sono più complesse: richiedono filtri su is\_current, gestione date con BETWEEN, COALESCE per data\_fine NULL. Curva di apprendimento più ripida per analyst.



#### Gestione Errori

Correggere errori storici è molto più difficile: non si può semplicemente fare UPDATE. Servono procedure complesse per "riscrivere la storia" mantenendo consistenza.



#### Manutenzione

Richiede monitoraggio continuo: crescita storage, performance query, integrità date. Necessità di procedure di cleanup e archiving per dimensioni molto attive.

"SCD Type 2 è la scelta giusta quando il valore delle analisi temporali supera il costo di complessità e storage. Per la maggior parte delle dimensioni business-critical (clienti, prodotti, fornitori), questo trade-off è favorevole. Per dimensioni con alta frequenza di cambiamento o basso valore analitico, considera alternative più semplici."

## SCD Type 3 e Type 4

Oltre a SCD Type 2, esistono altre strategie per gestire i cambiamenti dimensionali in scenari specifici. Type 3 e Type 4 offrono approcci alternativi con trade-off diversi in termini di completezza dello storico, complessità e performance. Comprendere quando utilizzare queste strategie alternative amplia il toolkit del data engineer.

### SCD Type 3: Add Previous Value Column



Aggiunge colonne specifiche per memorizzare il valore precedente di un attributo. Limita lo storico a un numero fisso di versioni (tipicamente solo valore corrente + precedente). Non mantiene timestamp dei cambiamenti.

#### Struttura Esempio:

cliente_id	nome	segmento_corrente	segmento_precedente	data_cambio
1001	Mario Rossi	Premium	Standard	2024-01-16
2005	Laura Bianchi	VIP	Premium	2023-11-20

#### ✓ Quando usare Type 3:

- Serve tracciare solo il valore precedente, non tutto lo storico completo
- Cambiamenti rari e prevedibili (es: una sola promozione di segmento)
- Analisi semplici "before/after" senza necessità di trend temporali
- Semplicità implementativa è prioritaria rispetto a completezza storico

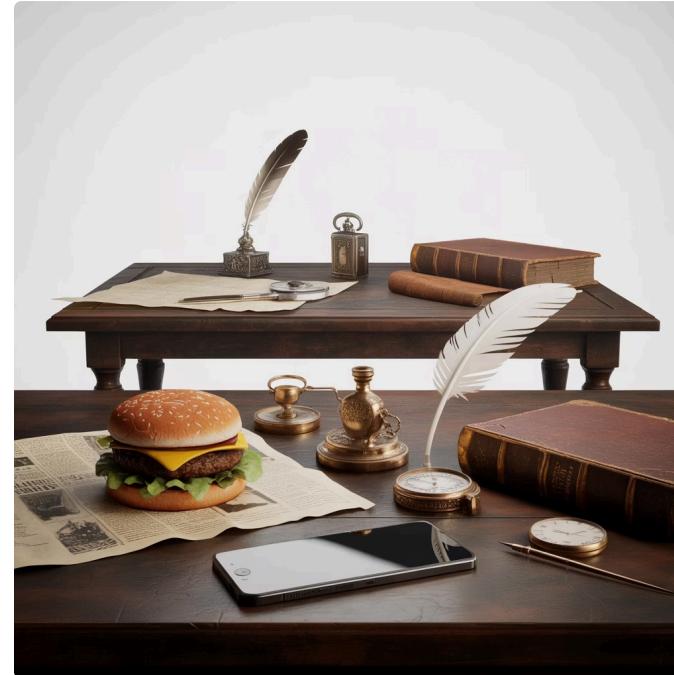
#### Vantaggi:

- Molto semplice: solo poche colonne extra, nessuna complessità di versioning
- Query facili: accesso diretto a valori correnti e precedenti
- Performance ottimali: una sola riga per entità, nessun join temporale

#### Svantaggi:

- Storico limitato: solo una versione precedente, tutto il resto è perso
- Inflessibile: per tracciare più valori servono più colonne (es: segmento\_penultimo)
- Nessun audit trail completo: timestamp limitato, nessuna traccia di tutti i cambiamenti

### SCD Type 4: History Table



Separa fisicamente i dati correnti (dimension table veloce e snella) dai dati storici (history table separata). La dimension table mantiene solo l'ultima versione, mentre la history table contiene tutte le versioni precedenti.

#### Struttura a Due Tabelle:

##### dim\_cliente (corrente - veloce):

cliente_id	nome	segmento	valore_annuo
1001	Mario Rossi	Premium	12,000€

##### dim\_cliente\_history (storico - completo):

cliente_id	segmento	valore_annuo	data_inizio	data_fine
1001	Standard	500€	2023-01-01	2024-01-15

#### ✓ Quando usare Type 4:

- Dimensioni molto grandi (milioni di righe) con cambiamenti frequenti
- Query correnti devono essere ultra-veloci, query storiche possono essere più lente
- Separazione fisica tra "hot data" (corrente) e "cold data" (storico)
- Necessità di archiviare/comprimere dati storici separatamente

#### Vantaggi:

- Performance eccellenti su query correnti: tabella principale piccola e veloce
- Flessibilità storage: possiamo archiviare/comprimere history table separatamente
- Storico completo mantenuto: stesso livello di dettaglio di Type 2

#### Svantaggi:

- Complessità architetturale: gestione di due tabelle sincronizzate
- Query point-in-time complesse: richiedono UNION tra corrente e history
- ETL più complesso: logica per decidere quando spostare dati in history