

# Nowoczesna Platforma Danych Databricks

Databricks to ujednolicona platforma do zarządzania danymi i analityki, która łączy najlepsze cechy hurtowni danych z elastycznością jeziora danych, tworząc architekturę typu lakehouse. Umożliwia efektywne przetwarzanie danych na dużą skalę, analitykę i rozwój sztucznej inteligencji.

# Agenda

## Wprowadzenie

- Nowoczesna platforma danych Databricks
- Krzysztof Burejza (Inżynier Danych Azure)
- Osobiste pasje

## Architektura Lakehouse na Databricks

- Aktualne wyzwania w obszarze danych
- Ewolucja od magazynu danych do jeziora danych i do Lakehouse
- Przegląd i historia platformy Databricks

## Praca z danymi

- Odczytywanie danych (CSV, JSON, Parquet)
- Tworzenie i eksploracja ramek danych
- Przechowywanie i partycjonowanie

## Tabele Delta

- Podstawy Delta Lake
- Właściwości ACID
- Podróże w czasie
- Operacje DML
- Optymalizacja (OPTIMIZE, ZORDER, Liquid Clustering)

## Pozyskiwanie danych

- Tryb wsadowy a tryb strumieniowy
- COPY INTO
- Auto Loader
- Tryby wyzwalania

## Potoki Lakeflow

- Podejście deklaratywne
- Tabele strumieniowe a widoki zmaterializowane
- Oczekiwania dotyczące jakości danych
- Automatyzacja i DAG

## Orkiestracja

- Zadania Lakeflow (przepływy pracy Databricks)
- Typy i zależności zadań
- Wyzwalacze i parametryzacja
- Monitorowanie i alerty

## Katalog Unity

- Warstwa zarządzania
- Kontrola dostępu
- Szczegółowe zabezpieczenia (maskowanie kolumn, bezpieczeństwo na poziomie wiersza)
- Pochodzenie i audyt



Krzysztof Burejza

### ***Data Engineer Azure***

Inżynier danych z 10-letnim doświadczeniem w pracy z danymi, od SQL Server po platformy chmurowe takie jak Azure, Databricks i Microsoft Fabric.

Skoncentrowany na przekształcaniu surowych danych w cenne rozwiązania, budowaniu potoków danych, hurtowni danych i lakehouse'ów. Entuzjasta nowych technologii na styku danych i sztucznej inteligencji, które sprawiają, że inżynieria danych staje się jeszcze bardziej fascynującą.



# Przedstawmy się!

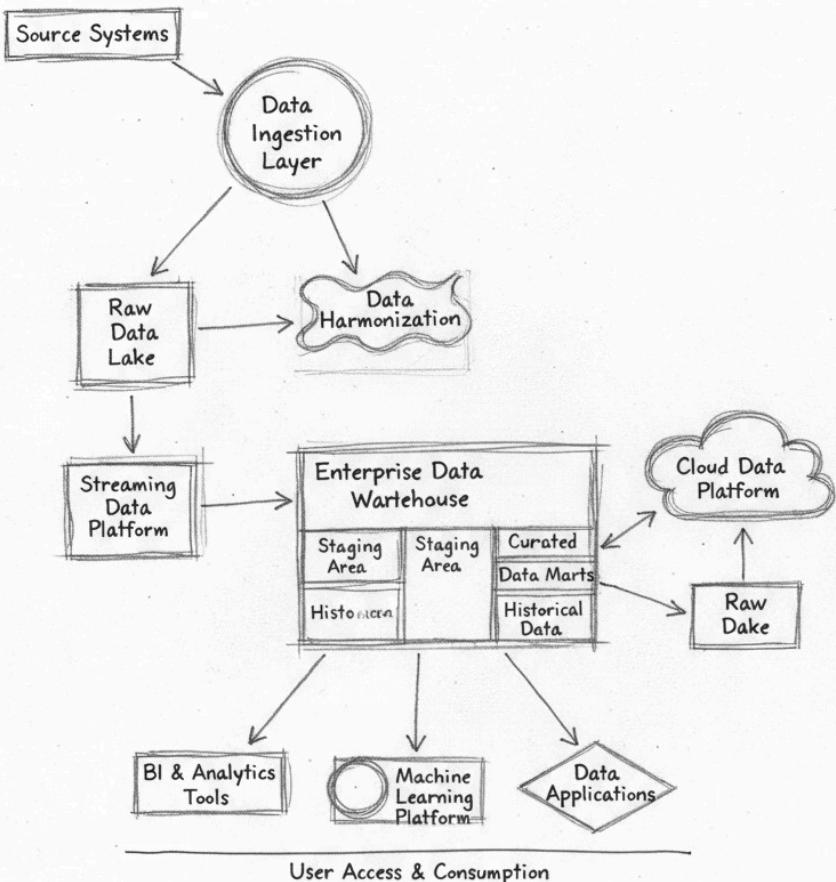
Abyśmy mogli lepiej się poznać i stworzyć angażującą atmosferę, prosimy o krótkie przedstawienie się. Podziel się z nami:

- Swoim imieniem i nazwiskiem
- Rolą lub stanowiskiem
- Krótkim doświadczeniem z platformą Databricks (jeśli posiadasz)

# Architektura Lakehouse

Nowoczesna architektura dla inżynierów danych

# Typowy Krajobraz Danych w Organizacji



## Magazyn Danych

Klasyczne rozwiązania, takie jak Serwer SQL, Oracle czy Snowflake. Dane ustrukturyzowane, wysokie koszty, przewidywalne obciążenie.

## Jeziorko Danych

Pliki w usłudze Azure Data Lake Storage lub AWS S3. Wysoka elastyczność, niskie koszty przechowywania, ale brak możliwości transakcyjnych.

## Narzędzia ETL/ELT

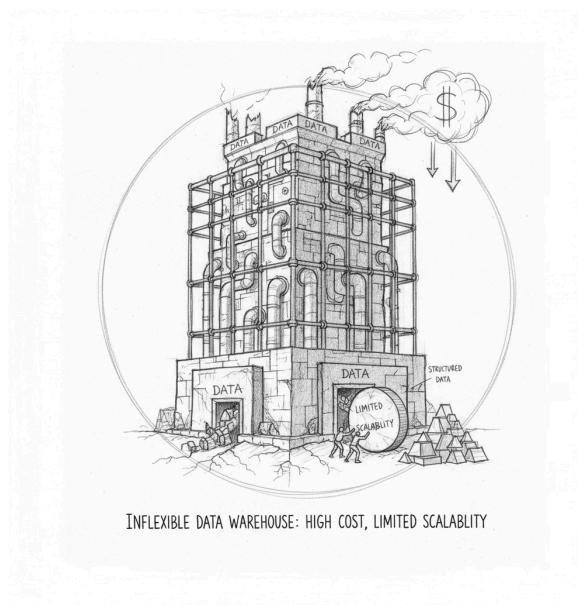
Zróżnicowany zestaw: SSIS, Azure Data Factory, Informatica, Apache Airflow. Każde z innym ekosystemem i kompetencjami.

## Środowiska Specjalistyczne

Oddzielne platformy do raportowania analityki biznesowej (BI) i projektów sztucznej inteligencji/uczenia maszynowego (AI/ML). Powielanie danych i wysiłku.

Nowoczesne organizacje utrzymują złożony ekosystem narzędzi do zarządzania danymi, co prowadzi do fragmentacji i zwiększenych kosztów operacyjnych.

# Obecny Świat Danych – Wyzwania



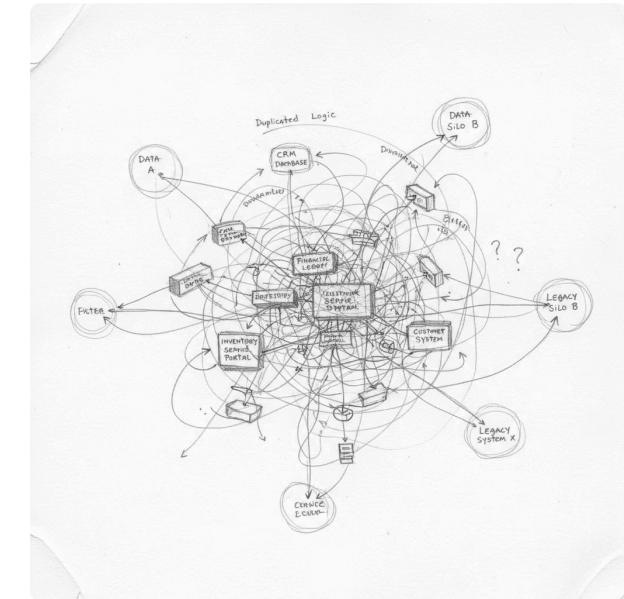
## Klasyczne Hurtownie Danych

- Wysokie koszty licencji i infrastruktury
- Ograniczona skalowalność
- Sztywne schematy
- Głównie dla danych ustrukturyzowanych



## Jeziorko Danych

- Tania pamięć masowa, ale "bagno danych"
- Brak transakcji ACID (Atomowość, Spójność, Izolacja, Trwałość)
- Problemy z zarządzaniem (Ład Danych)
- Trudne odkrywanie i pochodzenie danych



## Środowiska Rozproszone

- Oddzielne systemy dla Analityki Biznesowej i Uczenia Maszynowego
- Duplikacja danych i logiki
- Silosy organizacyjne
- Problemy z wydajnością w skali

# Wpływ sztucznej inteligencji i dużych modeli językowych na wymagania platform danych



## → **Dane niestrukturalne**

Modele sztucznej inteligencji wymagają dostępu do tekstu, obrazów, audio i wideo. Znacznie więcej niż tradycyjne tabele.

## → **Zunifikowana platforma**

Potrzeba jednego miejsca na surowe dane, cechy, modele uczenia maszynowego i metadane. Eliminowanie kopiowania między systemami.

## → **Zarządzanie i jakość**

Sztuczna inteligencja/duże modele językowe zwiększają problemy z jakością danych. Wzrost znaczenia pochodzenia, audytu i kontroli dostępu.

## → **Konwergencja hurtowni danych + jeziora danych + uczenia maszynowego**

Konieczność platformy łączącej możliwości hurtowni danych, elastyczność jeziora danych i wsparcie sztucznej inteligencji w jednym rozwiążaniu.

Era generatywnej sztucznej inteligencji zasadniczo zmienia oczekiwania wobec platform danych, wymuszając konsolidację i modernizację architektury.

# Databricks: Platforma Danych + AI

# Czym jest Databricks?

Databricks to ujednolicona platforma analityczna zbudowana na Apache Spark, stworzona przez założycieli tego projektu z UC Berkeley. Zapewnia kompleksowe rozwiązanie typu lakehouse, łączące:

- Inżynieria danych – przetwarzanie i przygotowanie danych
- Nauka o danych – zaawansowana analityka i uczenie maszynowe
- Analityka biznesowa – wizualizacje i raporty biznesowe

Platforma jest dostępna we wszystkich głównych chmurach publicznych (AWS, Azure, GCP), co pozwala na elastyczne wdrożenia zgodne z istniejącą infrastrukturą.



**databricks**



# Założenie Databricks

## 2013: Powstanie Databricks

Założyciele – twórcy Apache Spark z UC Berkeley, w tym Matei Zaharia, Ali Ghodsi, Ion Stoica – decydują się na komercjalizację technologii.

## Misja Firmy

Uproszczenie korzystania z Apache Spark w środowiskach produkcyjnych i w chmurze. Uczynienie zaawansowanej analityki dostępną dla szerszego grona organizacji.

Databricks zostało założone przez zespół, który nie tylko stworzył Sparda, ale nadal aktywnie rozwija zarówno sam silnik, jak i ekosystem wokół niego.

## Zespół założycielski



**Ali Ghodsi**  
Co-founder and Chief Executive Officer



**Ion Stoica**  
Co-founder and Executive Chairman



**Matei Zaharia**  
Co-founder and CTO



**Patrick Wendell**  
Co-founder and VP of Engineering



**Reynold Xin**  
Co-founder and Chief Architect



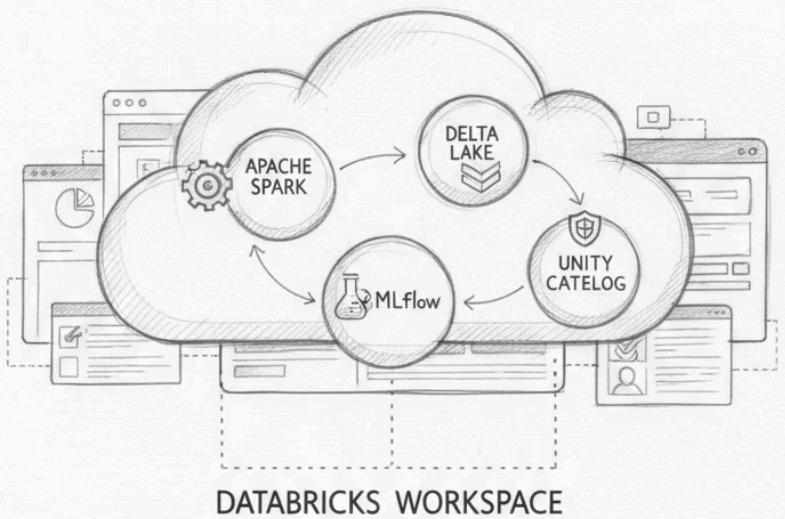
**Andy Konwinski**  
Co-founder and VP of Product Management



**Arsalan Tavakoli-Shiraji**  
Co-founder and SVP of Field Engineering

# Czym jest Databricks?

Databricks to chmurowa platforma Data + AI zbudowana na sprawdzonych technologiach open-source:



## Apache Spark

Rozproszone przetwarzanie danych w pamięci, standardowy silnik analityczny dla dużych zbiorów danych.

## Delta Lake

Otwarty format tabel z transakcjami ACID i możliwością podróży w czasie, podstawa Lakehouse.

## MLflow

Otwarta platforma do zarządzania cyklem życia modeli uczenia maszynowego (ML), od eksperymentów do wdrożenia.

## Unity Catalog

Centralny katalog metadanych z zaawansowanym zarządzaniem, śledzeniem pochodzenia danych i kontrolą dostępu.

Dostępny w głównych chmurach publicznych, w tym Azure Databricks – natywnie zintegrowany z ekosystemem Microsoft Azure.

# Architektura Databricks:

## Płaszczyzna kontrolna kontra Płaszczyzna danych

### Płaszczyzna kontrolna

#### Zarządzana przez Databricks:

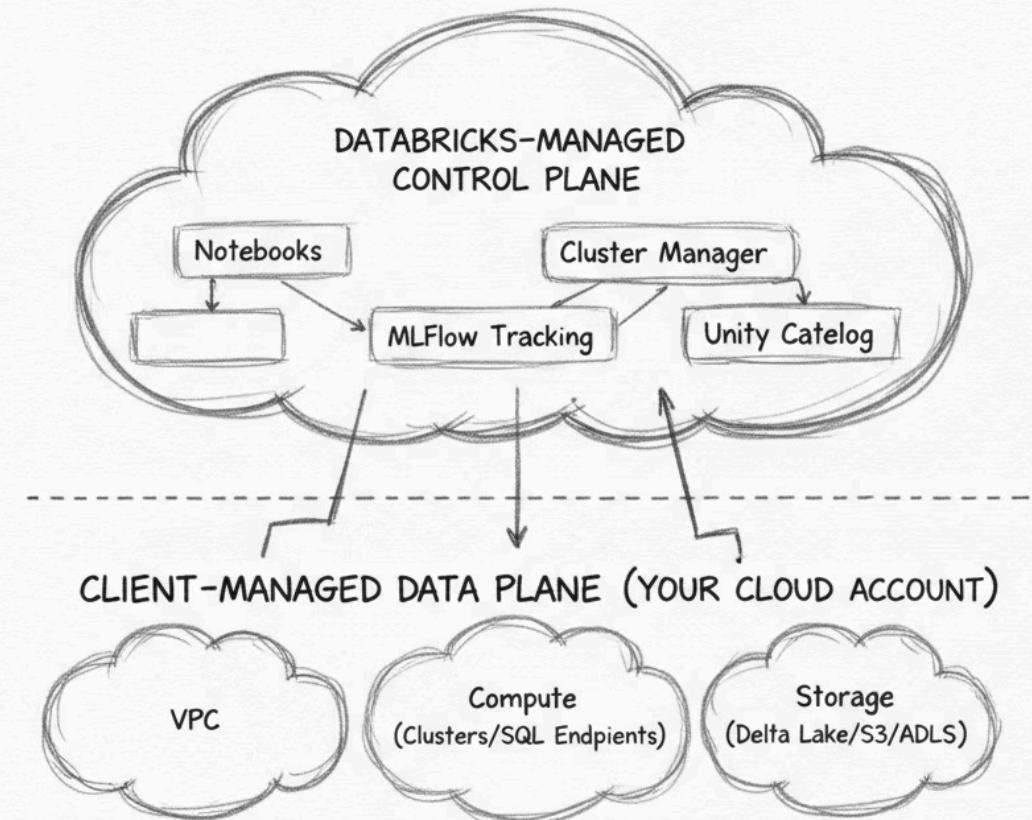
- Aplikacja internetowa – interfejs użytkownika
- Menedżer klastrów – zarządzanie klastrami
- Harmonogram zadań – planowanie zadań
- Serwer Notatników – środowisko programistyczne
- Bezpieczeństwo i Ład Korporacyjny – bezpieczeństwo
- Magazyn metadanych – metadane

### Płaszczyzna danych

#### Środowisko klienta:

- Klastry obliczeniowe (Apache Spark)
- Pamięć masowa klienta (S3/ADLS/GCS)
- Sieć (VPC/VNet)
- Grupy i polityki bezpieczeństwa

Komunikacja między warstwami odbywa się za pośrednictwem bezpiecznych wywołań API, zapewniając izolację danych klienta.



# Integracja Databricks z Chmurą

## Przechowywanie danych

Natywna integracja z Azure Data Lake Storage, AWS S3, Google Cloud Storage. Databricks odczytuje i zapisuje dane bezpośrednio w chmurze, bez pośredników.

## Bezpieczeństwo i tożsamość

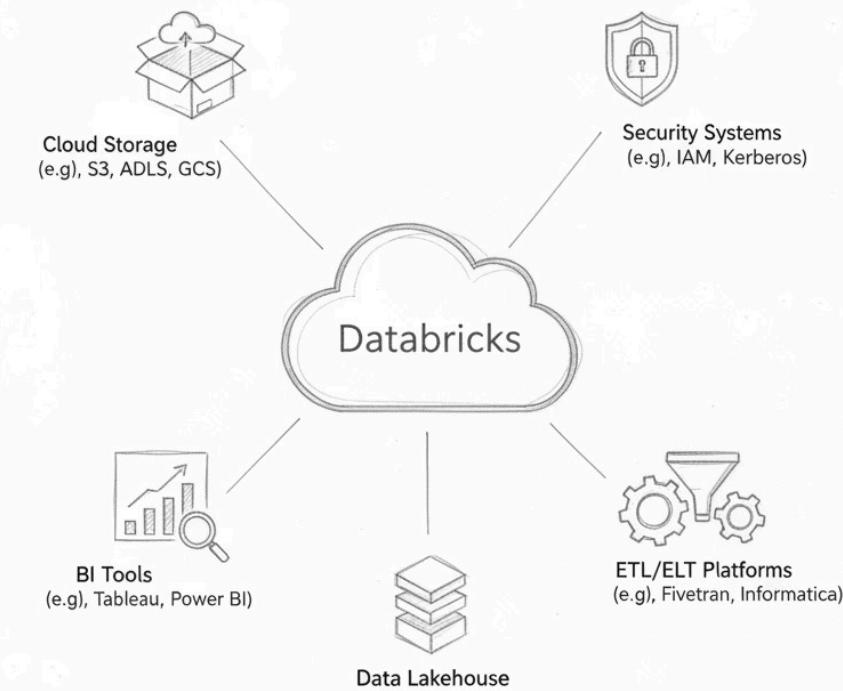
Azure Active Directory, AWS IAM, Google Cloud IAM do uwierzytelniania. SCIM provisioning i SSO dla przedsiębiorstw. Szyfrowanie w spoczynku i podczas przesyłania.

## Integracje BI

Łączniki dla Power BI, Tableau, Looker, Qlik. Sterowniki JDBC/ODBC dla dowolnych narzędzi SQL. Partner Connect do szybkiej konfiguracji.

## Integracje ETL/ELT

Współpraca z Fivetran, dbt, Azure Data Factory, Informatica, Talend. Auto Loader do przyrostowego importu plików z pamięci masowej w chmurze.



# **Wprowadzenie do Platformy Databricks**

Platforma Inteligencji Danych

# Przestrzeń robocza Databricks

## – Przegląd Interfejsu

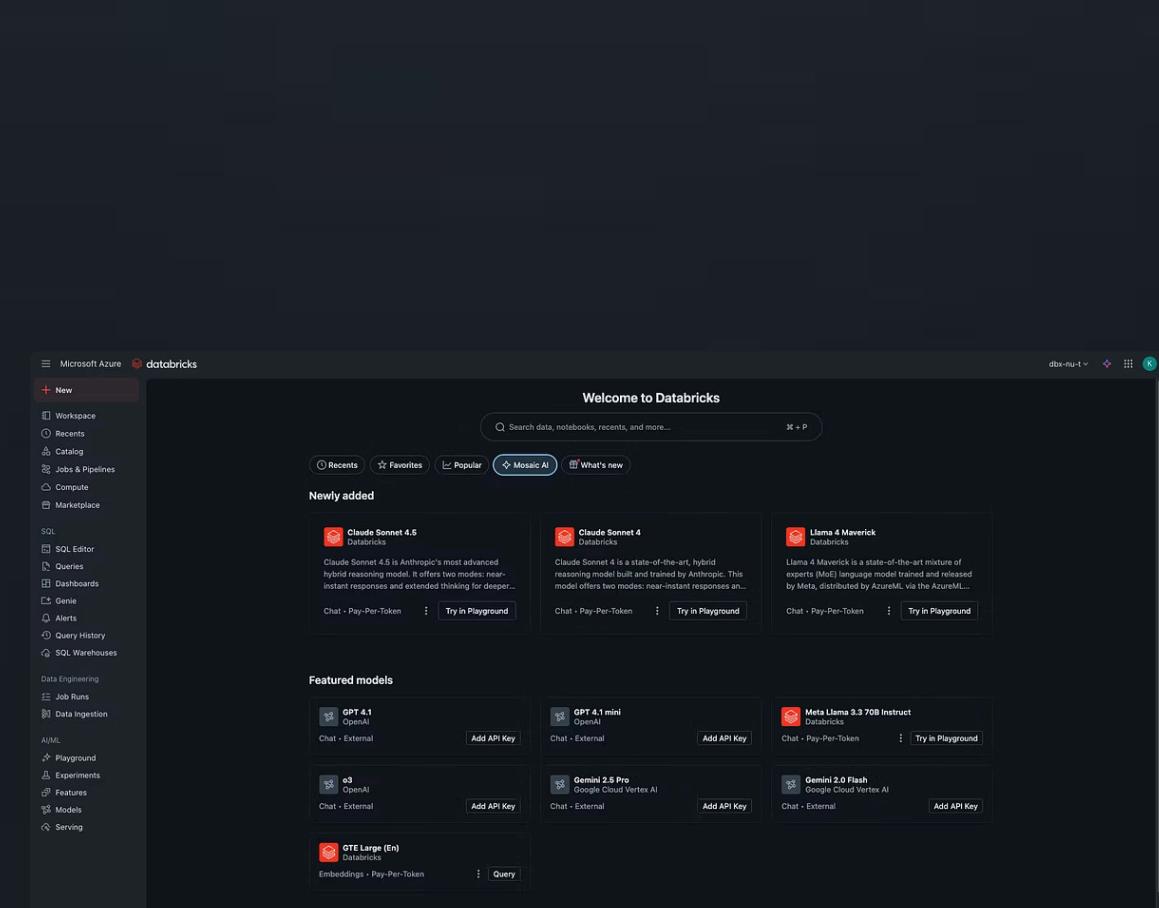
Przestrzeń robocza Databricks (Databricks Workspace) to ujednolicone środowisko pracy dla wszystkich ról związanych z danymi: inżynierów danych, analityków danych i naukowców danych. Interfejs internetowy integruje kod, dane, zasoby obliczeniowe (Compute) i orkiestrację w jednym miejscu.

### Główne Sekcje

- Przestrzeń robocza (Workspace):** foldery z notatnikami (notebooks), pulpitem nawigacyjnym, bibliotekami kodu
- Eksplorator Katalogu (Catalog Explorer):** przeglądanie obiektów Unity Catalog (katalogi, schematy, tabele)
- Dane (Data):** dostęp do tabel, widoków, funkcji i woluminów
- Zasoby obliczeniowe (Compute):** zarządzanie klastrami i magazynami SQL (SQL Warehouses)
- Przebiegi pracy (Workflows):** zadania (Jobs), Delta Live Tables, orkiestracja potoków danych

### Organizacja Pracy

- Foldery użytkownika (/Users/email@company.com) do pracy osobistej
- Foldery współdzielone (/Shared) do projektów zespołowych
- Repozytoria (Repositories) zintegrowane z Git do CI/CD i kontroli wersji
- Pulpity nawigacyjne do interaktywnych wizualizacji SQL



# Notatniki Databricks – Główne narzędzie pracy

Notatniki Databricks to interaktywne środowisko łączące kod, dokumentację i wyniki. Obsługują Python, Scala, SQL i R w jednym dokumencie, umożliwiając hybrydowe podejście do przetwarzania danych.

## Kluczowe funkcjonalności

- Podział komórek z różnymi językami (magiczne komendy)
- Natychmiastowa informacja zwrotna – wyniki wizualizowane bezpośrednio pod kodem
- Współpraca w czasie rzeczywistym (edycja kolaboracyjna)
- Kontrola wersji poprzez integrację z repozytoriami Git
- Parametryzacja i uruchamianie jako Zadania
- Eksport do formatów HTML, DBC, Jupyter

Notatniki to naturalne miejsce do eksploracji danych, prototypowania transformacji i dokumentowania logiki biznesowej.

The screenshot shows the Databricks workspace interface. The left sidebar lists 'workshops' with sub-folders: 01\_databricks\_lakehouse\_intro, 02\_data\_import\_exploration, 03\_basic\_transformations\_sql\_pyspark, 04\_data\_cleaning\_quality, and 05\_views\_workflows. The top navigation bar includes 'File', 'Edit', 'View', 'Run', 'Help', 'Python' (which is selected), 'Tabs: ON', and a 'main' tab. On the right, there are sections for 'BRONZE\_SCHEMA' and 'CATALOG', showing 'trainer\_bronze' and 'training\_catalog'. Below this, a section titled 'Konfiguracja' contains the text 'Import bibliotek i ustawienie zmiennych środowiskowych:' followed by a code snippet. Two code editor panes are visible at the bottom, both displaying identical setup code:

```
1 from pyspark.sql import functions as F
2 from pyspark.sql.types import *
3 import re
4
5 # Wyświetl kontekst użytkownika (zmienne z 00_setup)
6 print("==== Kontekst użytkownika ===")
7 print(f"Katalog: {CATALOG}")
8 print(f"Schema Bronze: {BRONZE_SCHEMA}")
9 print(f"Schema Silver: {SILVER_SCHEMA}")
10 print(f"Schema Gold: {GOLD_SCHEMA}")
11 print(f"Użytkownik: {raw_user}")
12
13 # Ustaw katalog jako domyślny
14 spark.sql(f"USE CATALOG {CATALOG}")
```

The second code editor pane shows the same code.

# Databricks Repozytoria – Integracja z Git

## Klonowanie repozytorium

Połączenie z GitHub, GitLab, Bitbucket, Azure DevOps.  
Importowanie istniejących projektów do Obszaru Roboczego.  
Pełna synchronizacja ze zdalnym repozytorium.

## Przebieg prac rozwojowych

Tworzenie gałęzi funkcji bezpośrednio w Databricks. Edytowanie notatników i zatwierdzanie zmian. Przełączanie między gałeziami dla różnych zadań.

## Przegląd kodu

Żądania scalenia w zewnętrznym systemie Git. Zespołowy przegląd kodu przed scaleniem. Integracja z potokami CI/CD.

## Wdrożenie

Automatyczne wdrożenia do produkcji po scaleniu. Wycofanie do poprzednich wersji w przypadku problemów. Pełna historia zmian i audyt.

Repozytoria umożliwiają profesjonalne zarządzanie kodem w zespołach, zapewniając kontrolę wersji, współpracę i możliwość stosowania standardowych praktyk DevOps.

# Woluminy – Standaryzowany dostęp do pamięci masowej

## Woluminy Katalogu Unity

Woluminy to zarządzane lokalizacje w pamięci masowej w chmurze (S3, ADLS, GCS) zarejestrowane w Katalogu Unity. Zapewniają bezpieczny, kontrolowany dostęp do niestrukturyzowanych plików.

### Typy woluminów:

- **Woluminy zarządzane:** Databricks zarządza cyklem życia plików, automatyczne czyszczenie po usunięciu (DROP)
- **Woluminy zewnętrzne:** wskazują na istniejącą pamięć masową, dane pozostają po usunięciu woluminu

Woluminy są częścią przestrzeni nazw Katalogu Unity (katalog.schemat.wolumin) i podlegają tym samym regułom uprawnień co tabele.

## DBFS – Warstwa kompatybilności

Databricks File System to abstrakcja nad pamięcią masową w chmurze, historycznie używana przed wprowadzeniem Katalogu Unity. Dostępna poprzez punkt montowania /dbfs/.

### Kiedy używać DBFS:

- Starszy kod wymagający ścieżek /dbfs/
- Pliki tymczasowe i pamięć podręczna
- Kompatybilność ze starymi notatnikami

### Kiedy unikać:

- Nowe projekty powinny używać Woluminów
- Dane produkcyjne – brak zarządzania w DBFS
- Udostępnianie danych między obszarami roboczymi

# Zalecane Wzorce Przechowywania Danych

## Pobieranie Surowych Danych

**Woluminy Zewnętrzne** do danych wejściowych z systemów źródłowych. Dane pozostają pod kontrolą inżynierów danych, niezależnie od obszaru roboczego. Łatwa integracja z istniejącymi jeziorami danych.

## Przetworzone Tabele

**Zarządzane Tabele Delta** w Katalogu Unity dla przetworzonych danych. Databricks zarządza cyklem życia, optymalizacją i zarządzaniem. Pełna kontrola dostępu i pochodzenia.

## Artefakty ML i Punkty Kontrolne

**Zarządzane Woluminy** dla modeli ML, punktów kontrolnych, logów. Automatyczne czyszczenie starych wersji. Integracja ze śledzeniem MLflow.

## Tymczasowe i Pamięć Podręczna

**DBFS** dla plików tymczasowych w ramach sesji. Szybki dostęp lokalny. Brak wymagań dotyczących zarządzania dla danych krótkotrwałych.

Kluczowa zasada: używaj **Woluminów** Katalogu Unity i **Zarządzanych Tabel** dla wszystkich danych o długim cyklu życia, **DBFS** tylko dla danych efemerycznych.

# Namespaces

## Three-level namespace

Each object is addressed by its full path:

```
catalog.schema.table  
production.sales.orders  
dev.analytics.customer_360
```

Ability to set a default catalog and schema for a session (USE CATALOG, USE SCHEMA), which allows for the use of shorter names.

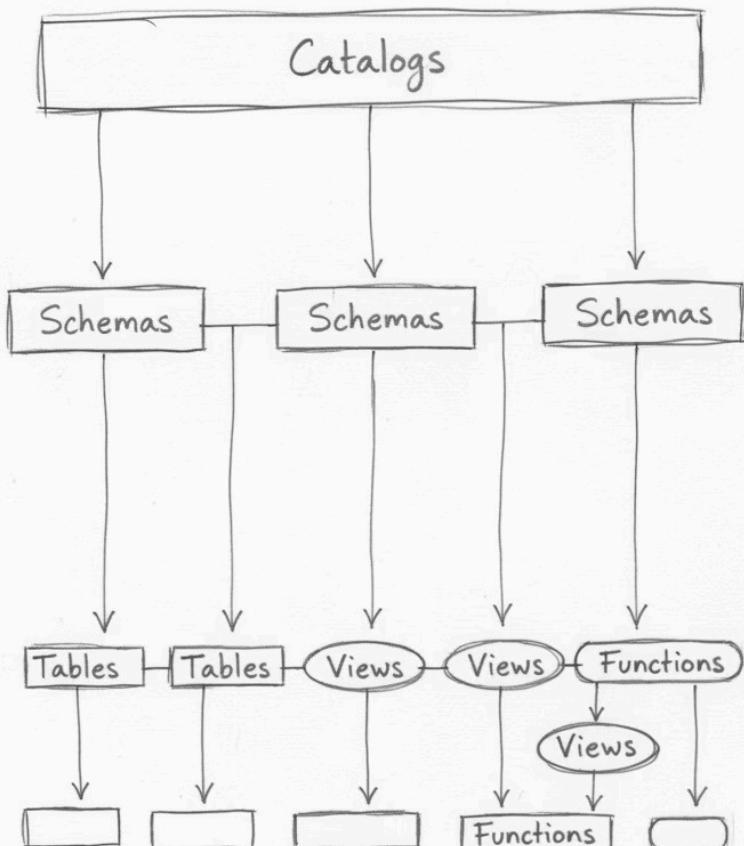
Elimination of naming conflicts between projects and environments. Clear separation of data domains.

The screenshot shows the Databricks Catalog Explorer interface. At the top, there's a navigation bar with 'Catalog' and other tabs like 'Serverless Starter Ware...', 'Serverless', and 'S'. Below the navigation is a search bar labeled 'Type to search...'. The main area is a tree view of namespaces:

- My organization
  - dbx\_weu\_trn
  - system
  - \_databricks\_internal
  - ai\_sandbox
- ecommerce\_platform\_trainer
  - bronze
    - Tables (19)
      - customers
      - customers\_batch
      - customers\_delta
      - customers\_raw
      - customers\_test\_clone
      - customers\_with\_validation
      - lakehouse\_demo
      - orders
      - orders\_autoloader
      - orders\_batch
      - orders\_demo
      - orders\_modern
      - orders\_raw
      - orders\_rescued

On the right side, there's a detailed view for the 'customers' table under the 'bronze' catalog. It includes an 'AI Suggested Description' section stating: 'The table contains customer demographic and registration information, such as first name, last name, email, phone, city, state, and country. It also includes registration date and customer segment. This data can be used for various analytical purposes.' Below this are buttons for 'Accept' and 'Edit'. Further down, there's a table showing the columns and their types:

Column	Type	Comment
customer_id	string	
first_name	string	
last_name	string	
email	string	
phone	string	
city	string	
state	string	
country	string	



# Katalog Unity – hierarchia obiektów

Katalog Unity wprowadza trójpoziomową przestrzeń nazw dla danych, umożliwiając lepszą organizację i kontrolę dostępu na różnych poziomach szczegółowości.

## Katalog

Najwyższy poziom organizacji. Zazwyczaj: jeden katalog na środowisko (dev, staging, prod) lub na domenę biznesową (sprzedaż, finanse, HR). Określa zasady dostępu i ogólne włascielstwo.

## Schemat (Baza danych)

Logiczne grupowanie powiązanych tabel i widoków. Odpowiednik schematu w SQL. Umożliwia oddzielenie warstw (bronze, silver, gold) lub podsystemów. Niezależne zarządzanie uprawnieniami dla każdego schematu.

## Tabela / Widok / Funkcja

Specyficzne obiekty danych. Tabele (zarządzane lub zewnętrzne), Widoki (logika biznesowa), Funkcje (UDF). Granularne uprawnienia SELECT, MODIFY, EXECUTE. Pełne śledzenie pochodzenia i audyt.

# Uprawnienia w Unity Catalog

## Model uprawnień

Uprawnienia są dziedziczone hierarchicznie:

- **KATALOG:** UŻYJ (USE), TWORZENIE SCHEMATU (CREATE SCHEMA)
- **SCHEMAT:** UŻYJ (USE), TWORZENIE TABELI (CREATE TABLE), TWORZENIE WIDOKU (CREATE VIEW)
- **TABELA:** WYBIERZ (SELECT), MODYFIKUJ (MODIFY), ODCZYTAJ METADANE (READ\_METADATA)

Podmiot zabezpieczeń (użytkownik, grupa, podmiot usługi) otrzymuje uprawnienia na każdym poziomie. Administrator może delegować własność i zarządzanie. Użyj komend GRANT/REVOKE w SQL do szczegółowej kontroli.

 **Zarządzanie i audyt:** Unity Catalog rejestruje wszystkie operacje dostępu (historia zapytań, dzienniki audytu). Automatyczne śledzenie pochodzenia danych – skąd dane pochodzą i kto ich używa. Integracja z zewnętrznymi narzędziami zarządzania (Collibra, Alation).

## Performance

 Machine learning ⓘ

Databricks runtime

17.3 LTS

Scala 2.13, Spark 4.0.0 ▾

 Photon acceleration ⓘ

Worker type

Min

Max

Standard\_D4ds\_v5

16 GB Memory, 4 Cores ▾

2

8

 Single node

# Obliczenia – typy klastrów Sparka

### Klaster Ogólnego Przeznaczenia

Interaktywne środowisko do rozwoju i eksploracji. Dzielone przez wielu użytkowników. Długi czas pracy, automatyczne skalowanie, ręczne zakończenie. Używane do notatników, zapytań doraźnych, tworzenia prototypów.

### Klaster Zadaniowy

Dedykowany klaster do konkretnego zadania. Automatyczne tworzenie przed wykonaniem, automatyczne usuwanie po zakończeniu. Izolacja zasobów dla każdego zadania. Optymalny do produkcji: niższe koszty, brak konfliktów.

### Klaster Jednowęzłowy

Jeden węzeł bez procesów roboczych, Spark działa w trybie lokalnym. Niskie koszty dla małych zadań i testów. Ograniczona skalowalność. Przydatny do rozwoju, weryfikacji koncepcji, lekkiego przetwarzania.

# Kluczowe parametry klastra

## Autoskalowanie

Automatycznie dodaje/usuwa węzły robocze w zależności od obciążenia. Konfiguracja minimalnej/maksymalnej liczby węzłów. Oszczędności kosztów w okresach niskiej aktywności. Reakcja na skoki użycia bez ręcznej interwencji.

## Automatyczne zakończenie

Automatyczne wyłączenie klastra po określonym czasie bezczynności (np. 30 minut). Kluczowe dla kontroli kosztów w środowiskach deweloperskich/testowych. Klastry produkcyjne często bez automatycznego zakończenia (zawsze dostępne).

## Typy instancji

Wybór typów maszyn wirtualnych w zależności od obciążenia:

- **Zoptymalizowane pod kątem obliczeń:** do zadań intensywnie wykorzystujących procesor
- **Zoptymalizowane pod kątem pamięci:** do dużych agregacji, pamięci podręcznej
- **Zoptymalizowane pod kątem pamięci masowej:** do potoków intensywnie wykorzystujących operacje wejścia/wyjścia
- **Instancje GPU:** do trenowania i wnioskowania ML

## Profile klastrów

Predefiniowane konfiguracje dla typowych zastosowań.

Standaryzacja klastrów w organizacji. Egzekwowanie zasad (np. minimalny rozmiar węzła, maksymalny czas działania).

# Silnik Photon – akcelerator SQL

Photon to natywny silnik Databricks napisany w C++, zaprojektowany w celu maksymalizacji wydajności operacji SQL i ramek danych. Działa jako zamiennik dla części tradycyjnego wykonawcy Sparka.

## Gdzie Photon przynosi korzyści

- Zapytania SQL z dużą liczbą złączeń i agregacji
- Skanowanie i filtrowanie dużych tabel Delta
- Pulpity nawigacyjne analizy biznesowej i interaktywna analityka
- Potoki ETL z dużą liczbą transformacji kolumnowych
- Operacje na typach tekstowych, datowych, dziesiętnych

Przyspieszenie do 2-5x w porównaniu ze standardowym Sparkiem, często bez zmian w kodzie. Automatyczne wykrywanie możliwości użycia Photon.

## Kiedy nie jest konieczny

- Proste SELECT \* lub minimalne przetwarzanie
- Ciężkie operacje tasowania (shuffle) (Photon ich nie przyspiesza)
- UDFy w Pythonie (nie podlegają Photonowi)
- Obciążenia z niskim wykorzystaniem procesora

Photon jest dostępny w klastrach zadań i magazynach SQL. Włączany poprzez wybranie środowiska uruchomieniowego obsługującego Photon. Dodatkowy koszt około 20% w porównaniu ze standardowym środowiskiem uruchomieniowym, ale zwrot z inwestycji szybko się zwraca przy intensywnych obciążeniach.

# Magazyn SQL – dedykowany silnik dla BI

Różnice w porównaniu do klastrów notesów

Magazyn SQL to środowisko obliczeniowe zoptymalizowane pod kątem zapytań SQL, z innym modelem kosztów i cyklu życia.

Podczas gdy notesy działają na klastrach ogólnego przeznaczenia, Magazyn SQL jest dedykowany dla:

- Paneli kontrolnych i analityki interaktywnej
- Integracji z narzędziami analityki biznesowej (Power BI, Tableau)
- Zapytań ad-hoc od analityków biznesowych
- Zaplanowanych zapytań i alertów

Model cenowy: rozliczanie co do sekundy podczas aktywności. Automatyczne skalowanie na zapytanie, nie na użytkownika. Wiele klastrów dla wysokiej współbieżności.



# Przypadki użycia SQL Warehouse

## Eksploracja ad-hoc

Analitycy biznesowi eksplorują dane bez konieczności uruchamiania notebooks. Szybkie zapytania, natychmiastowe wyniki. Niskie koszty dla sporadycznych analiz.

## Pulpity nawigacyjne i raportowanie

Pulpity nawigacyjne w czasie rzeczywistym w Databricks SQL lub zewnętrznych narzędziach. Wspólne zapytania, sparametryzowane raporty. Automatyczne odświeżanie i zaplanowane alerty.

## Integracja z narzędziami BI

Bezpośrednie połączenie z Power BI, Tableau, Looker za pośrednictwem JDBC/ODBC. Jedno źródło prawdy dla całej organizacji. Scentralizowane zarządzanie uprawnieniami w Unity Catalog.

- ❑ **Serverless SQL Warehouse:** Nowa opcja eliminująca zimny start. Natychmiastowe wykonywanie zapytań bez oczekiwania na uruchomienie klastra. Zarządzane przez Databricks – zero konfiguracji infrastruktury. Wyższy koszt za sekundę, ale zwrot z inwestycji przy niskiej częstotliwości użytkowania.

# Obliczenia Serverless – kiedy mają sens

## Zalety Serverless

- **Zerowy zimny start:** natychmiastowe wykonanie zapytania bez czekania na uruchomienie klastra
- **Zarządzana infrastruktura:** Databricks zarządza skalowaniem i dostępnością
- **Płać za użycie:** rozliczanie tylko za czas wykonania, bez kosztów przestoju
- **Autoskalowanie:** automatyczne dostosowanie zasobów do złożoności zapytania
- **Uproszczone operacje:** brak potrzeby dostrajania parametrów klastra

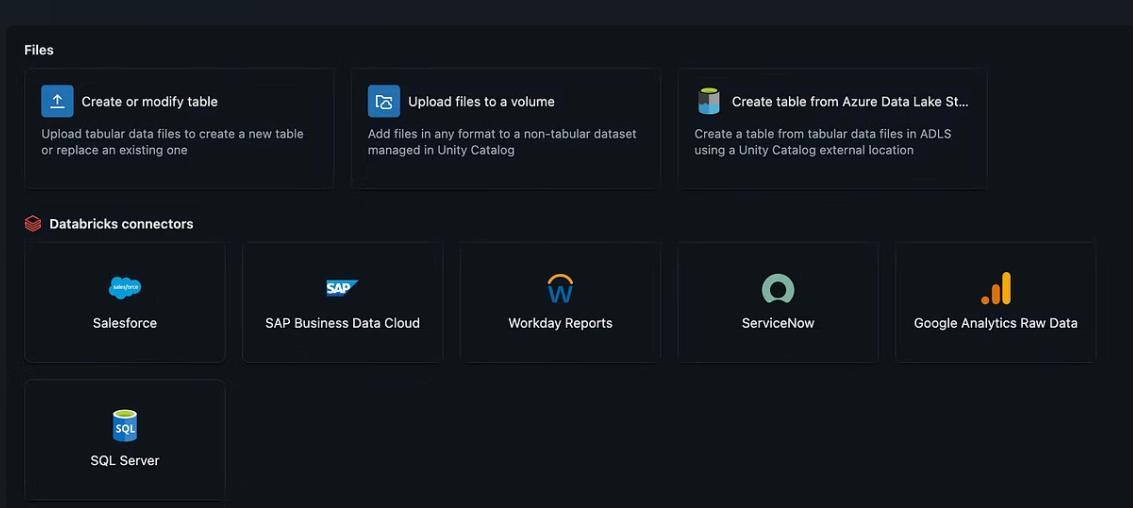
## Kiedy rozważyć Serverless

- Niska częstotliwość zapytań (kilka razy dziennie)
- Nieregularny wzorzec użycia (obciążenia szczytowe)
- Wymóg natychmiastowej odpowiedzi dla użytkowników końcowych
- Brak doświadczenia w zarządzaniu klastrami
- Środowiska deweloperskie/testowe z okazjonalnym użyciem

**Kiedy używać tradycyjnego klastra:** Ciągłe obciążenia (24/7), przewidywalne wzorce, potrzeba precyzyjnej kontroli, bardzo duże wolumeny danych (Serverless ma ograniczenia).

# Integracja z narzędziami BI

Databricks oferuje natywną integrację z popularnymi narzędziami BI za pośrednictwem złączy JDBC/ODBC oraz złączy partnerów. Unity Catalog zapewnia spójne uprawnienia i metadane dla wszystkich integracji.



## Power BI

Natywne złącze (tryb DirectQuery i Import). Jednokrotne logowanie za pośrednictwem Azure AD. Zabezpieczenia na poziomie wiersza z Unity Catalog. Automatyczne odświeżanie schematów i integracja przepływów danych.

## Tableau

Certyfikowane złącze Databricks dla Desktop i Server. Połączenie na żywo lub wyodrębnianie danych. Publikowanie w Tableau Online z osadzonymi poświadczaniami. Pełne przekazywanie zapytań SQL dla optymalnej wydajności.

## Looker / Qlik / ThoughtSpot

Połączenia JDBC/ODBC dla korporacyjnych platform BI. Niestandardowe definicje SQL i LookML/modeli. Integracja z Unity Catalog dla zarządzania i śledzenia pochodzenia danych.

# Notebooki – polecenia magiczne

Polecenia magiczne to specjalne dyrektywy w notebookach Databricks, które umożliwiają przełączanie języków, operacje na systemie plików i inne narzędzia. Każda komórka może używać innego języka, co pozwala na tworzenie hybrydowych potoków.

## %sql

Przełącza komórkę na SQL. Pełne wsparcie dla składni SQL Spark. Wyniki są automatycznie wizualizowane jako tabele. Możliwość tworzenia tymczasowych widoków dostępnych w PySpark.

## %python

Domyślny język w notebookach PySpark. Dostęp do sesji SparkSession jako spark. Pełne API DataFrame, ML, streaming. Importowanie zewnętrznych bibliotek.

## %scala

Przełącza na Scala (natywny język Spark). Wyższa wydajność dla niektórych operacji. Bezpośredni dostęp do wewnętrznych funkcji Spark.

## %r

Integracja z R do statystyk i zaawansowanej analityki. SparkR lub sparklyr do przetwarzania rozproszonego. Używane przez analityków danych preferujących ekosystem R.

## %md

Markdown do dokumentacji notebooków. Nagłówki, listy, tabele, obrazy. Interaktywne diagramy (Mermaid). Kluczowe dla czytelnych notebooków.

## %fs

Komendy systemu plików (ls, cp, rm). Operacje na DBFS, wolumenach, pamięci masowej w chmurze. Debugowanie ścieżek i zawartości plików.

# Tworzenie i eksploracja ramek danych

Podstawowe operacje na danych w Databricks

# Pliki w object storage – fizyczna reprezentacja danych

Dane w Databricks nie są przechowywane w tradycyjnej bazie danych. Zamiast tego, wszystkie dane znajdują się jako pliki w object storage (Azure Data Lake Storage, AWS S3 lub Google Cloud Storage).

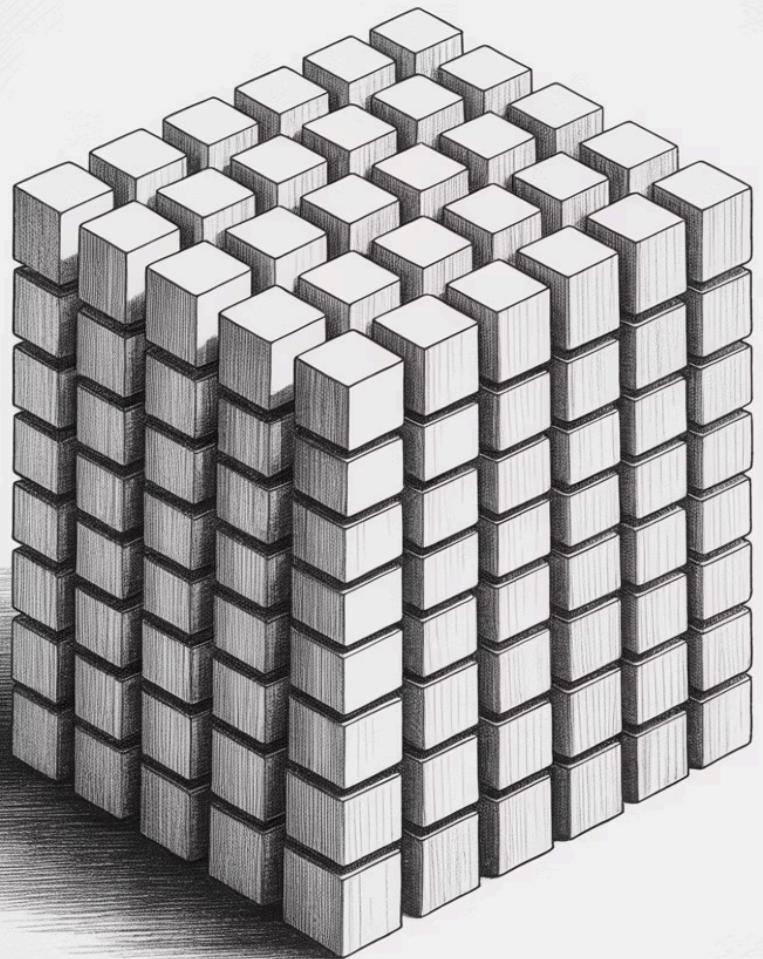
## Pliki surowe

- **CSV/JSON** – oparte na tekście, łatwe do importowania, ale nieefektywne
- Brak kompresji i optymalizacji
- Każde zapytanie musi przetwarzać cały plik
- Trudno wymusić schemat

## Pliki kolumnowe (zoptymalizowane)

- **Parquet** – kolumnowy, skompresowany, z metadanymi
- **Delta** – Parquet + delta log z transakcjami ACID
- Efektywne zapytania (odczytują tylko niezbędne kolumny)
- Kompresja i kodowanie wartości





# Format Kolumnowy: Wydajność w Praktyce

Format kolumnowy, taki jak **Apache Parquet**, jest standardem w Big Data, który przechowuje dane fizycznie zorganizowane według kolumn, nie wierszy. Ta pozornie prosta zmiana w organizacji danych ma ogromny wpływ na wydajność.

## Selektywne Odczytywanie

Odczytywanie tylko tych kolumn, które są potrzebne w zapytaniu – drastycznie redukując I/O

## Lepsza Kompresja

Dane w kolumnach są często podobne, co zwiększa wydajność algorytmów kompresji

## Optymalizacje Procesorowe

Umożliwia wektoryzowane przetwarzanie i wykorzystanie pamięci podręcznej procesora

**Delta Lake** rozszerza formaty kolumnowe o **metadane transakcyjne** i **dziennik zmian**, dodając warstwy zarządzania i spójności na poziomie tabeli.

# Współpraca SQL i PySpark

## Przekazywanie danych między językami

Tymczasowe widoki pełnią rolę mostu między SQL a API DataFrame.

Proces:

1. PySpark ładuje dane do DataFrame
2. `createOrReplaceTempView('view_name')`
3. Zapytanie SQL na tym widoku w następnej komórce
4. Wynik SQL również jako tymczasowy widok lub DataFrame

```
# Python
df.createOrReplaceTempView("customers_temp")
```

```
# SQL (następna komórka)
%sql
SELECT * FROM customers_temp
WHERE country = 'PL'
```

## Najlepsze praktyki

- Używaj SQL do prostych operacji `select`, `filter`, `aggregate` – jest jaśniejszy dla analityków
- Używaj PySpark do złożonej logiki, funkcji UDF, integracji ML
- Tymczasowe widoki jako kontrakty między etapami potoku
- Dokumentuj w Markdown, jakie dane są przekazywane
- Spójna konwencja nazewnictwa widoków (np. prefiks `temp_`)

Podejście hybrydowe maksymalizuje mocne strony obu języków i umożliwia współpracę między analitykami znającymi SQL a inżynierami Python.

# Ładowanie CSV – Podstawy

Format CSV (Comma-Separated Values) jest najpopularniejszym formatem wymiany danych. Databricks oferuje bogaty zestaw opcji do pracy z CSV, od prostych zastosowań po złożone scenariusze produkcyjne.

## Podstawowe Ładowanie

```
# Python
df_customers = (spark.read
    .option("header", "true") # pierwsza linia zawiera nazwy kolumn
    .option("inferSchema", "true") # Spark próbuje automatycznie
    wykryć typy danych
    .csv(f"{raw_data_path}/customers.csv")
)

display(df_customers)
```

**header=true:** pierwsza linia zawiera nazwy kolumn

**inferSchema=true:** Spark próbuje automatycznie wykryć typy danych

## Struktura Danych Klientów

Typowe kolumny w zbiorze danych klientów:

- customer\_id (liczba całkowita/ciąg znaków)
- first\_name, last\_name (ciąg znaków)
- email (ciąg znaków)
- country, city (ciąg znaków)
- signup\_date (data/znacznik czasu)
- total\_purchases (dziesiętne)

Pierwszy podgląd: `display(df)` w Databricks automatycznie renderuje ładną tabelę z opcjami sortowania i filtrowania.

# Eksploracja danych CSV

- **Szybkie statystyki**

```
df_customers.describe().show()
```

Zwraca liczebność, średnią, odchylenie standardowe, minimum, maksimum dla kolumn numerycznych. Pomaga szybko zidentyfikować wartości odstające i rozkład danych.

- **Liczba rekordów i unikalność**

```
df_customers.count() # całkowita liczba wierszy  
df_customers.select("customer_id").distinct().count()
```

Sprawdzenie, czy `customer_id` jest unikalny. Jeśli liczba unikalnych wartości < całkowitej liczby → mamy duplikaty.

- **Schemat i typy danych**

```
df_customers.printSchema()
```

Weryfikacja, czy `inferSchema` poprawnie wykryło typy. Typowe problemy: daty jako ciągi znaków, liczby jako ciągi znaków z powodu wartości `null` w pliku.

- **Rozkład wartości**

```
df_customers.groupBy("country").count().orderBy("count",  
ascending=False).show()
```

Kraje z największą liczbą klientów. Identyfikacja potencjalnych problemów z jakością danych (np. różne kodowanie dla tego samego kraju).

# Data Quality Report

?

	Name	Email	Date	Date	Product ID
1	John.Doe 1055-31		03-14-20		
	John.Doe 2026-85		2023-10-26	2023-10-26	P125-19
1	john.doe@email 2023-15	john.doe@email 2023-10-23	20/26-28 10/26/23	20/26-28 10/26/23	
2	John.Doe P126-19	invalid_email			
3	John.Doe 2026-38		11/22/98 21/26/26		
4	John.Doe P12545			10/26/23 41/20/35	
4	John.Doe 1026-19			11/26-24 19/2845	
	john.doe@email P12345			11/40-23 11/28-45	
	John.Doe P12345			P12345	
	P12345			P12325	

# Częste problemy z inferSchema

## Problemy z automatycznym wykrywaniem

- Wartości null w środku pliku:** Spark próbuje wykryć schemat tylko z pierwszych N wierszy. Jeśli pierwsze wiersze mają wartości, a później pojawiają się wartości null, typ może zostać niepoprawnie wykryty.
- Mieszane typy danych:** Kolumna głównie z liczbami, ale z okazjonalnymi tekstami → wykrywana jako string.
- Daty jako stringi:** inferSchema nie rozpoznaje formatów dat; wymagane jest ręczne rzutowanie.
- Wiodące zera:** np. kody pocztowe "01234" → wykrywane jako integer i tracą zera.

## Rozwiązanie

Zawsze weryfikuj schemat po załadowaniu:

```
df.printSchema()  
df.select("suspicious_column").show()  
( )
```

W przypadku środowisk produkcyjnych zalecane jest jawne definiowanie schematu zamiast inferSchema. Zapewnia to stabilność potoku i przewidywalne typy danych.

# Rozszerzone Opcje Czytnika CSV

Potoki produkcyjne wymagają precyzyjnej kontroli nad parsowaniem CSV. Databricks obsługuje szeroki zakres opcji dla przypadków brzegowych i niestandardowych formatów.

## Separatory i Cudzysłowy

```
.option("sep", ";") # separator inny niż przecinek  
.option("quote", "") # cudzysłów dla wartości  
.option("escape", "\\") # znak ucieczki
```

Użycie w Europie: średnik jako separator, przecinek jako separator dziesiętny.

## Wartości null

```
.option("nullValue", "NA")  
.option("emptyValue", "")
```

Różne systemy używają różnych tokenów dla wartości null: "NA", "NULL", "N/A", "-", pusty ciąg znaków. Jawna definicja zapobiega błędnej interpretacji.

## Kodowanie i BOM

```
.option("encoding", "UTF-8")  
.option("charset", "windows-1252")
```

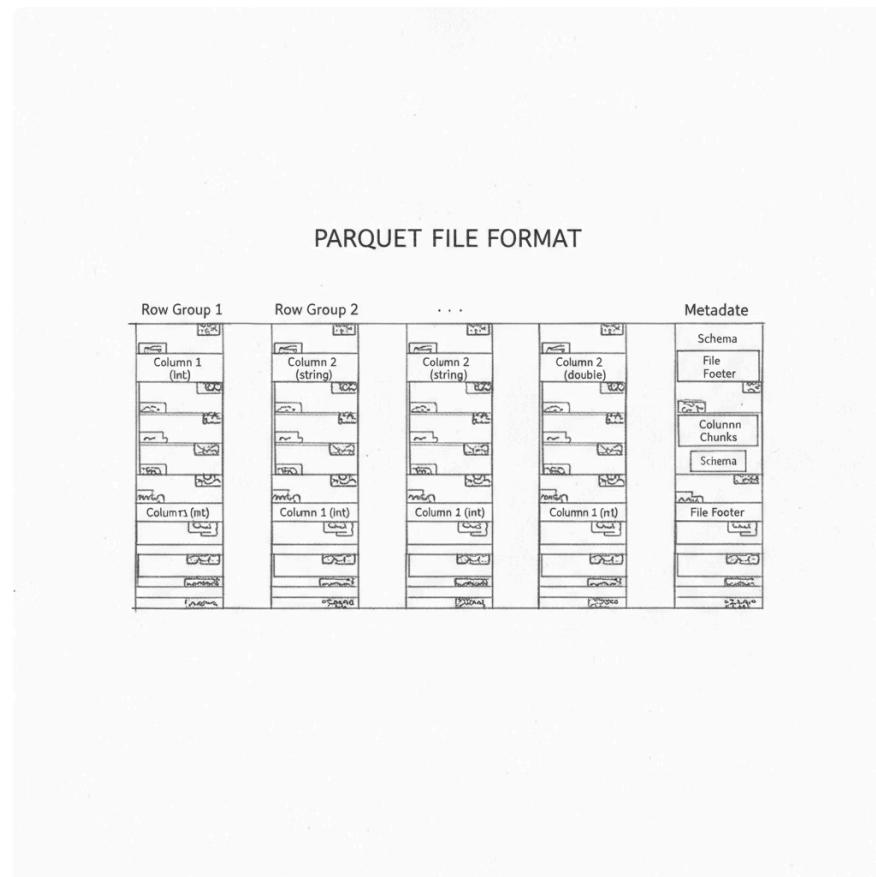
Pliki z systemu Windows mogą zawierać BOM (Byte Order Mark). BOM UTF-8 może uszkodzić pierwszą kolumnę. Użyj odpowiedniego zestawu znaków.

## Wieloliniowe i Uszkodzone Rekordy

```
.option("multiLine", "true")  
.option("mode", "PERMISSIVE")  
.option("columnNameOfCorruptRecord", "_corrupt_record")
```

`multiLine=true` dla plików CSV z nowymi liniami w wartościach. Tryb `PERMISSIVE`: zły rekord → null + zapisany w `\_corrupt\_record`.

# Import danych Parquet



## Ładowanie Parquet

```
df_products = (spark.read  
    .parquet(f"{raw_data_path}/products.parquet")  
)
```

```
df_products.printSchema()  
display(df_products)
```

Parquet przechowuje schemat w metadanych pliku, więc inferSchema nie jest potrzebne. Spark automatycznie odczytuje typy kolumn.

## Zalety formatu kolumnowego

- **Kompresja:** 3-10x lepsza niż CSV/JSON
- **Wydajność:** odczyt tylko niezbędnych kolumn (column pruning)
- **Typy danych:** zachowywane natywnie, bez parsowania
- **Predicate pushdown:** filtrowanie na poziomie pliku

# Ręczne definiowanie schematu dla CSV

## Tworzenie StructType

```
from pyspark.sql.types import *

customers_schema = StructType([
    StructField("customer_id", IntegerType(), False),
    StructField("first_name", StringType(), True),
    StructField("last_name", StringType(), True),
    StructField("email", StringType(), True),
    StructField("country", StringType(), True),
    StructField("signup_date", DateType(), True),
    StructField("total_purchases", DecimalType(10,2), True)
])

df = spark.read.schema(customers_schema).csv(path)
```

## Zalety jawnego definicji

- **Stabilność:** Identyczny schemat przy każdym uruchomieniu, niezależnie od danych wejściowych
- **Wydajność:** Brak narzutu związanego z inferSchema (skanowanie danych)
- **Walidacja:** Spark zakończy się błędem, jeśli dane nie pasują do schematu
- **Dokumentacja:** Schemat = kontrakt danych, jasne oczekiwania
- **Łatwiejsze debugowanie:** Błędy rzutowania widoczne natychmiast

Produkcyjne potoki danych powinny zawsze używać jawnego schematu.

False/True w StructField to flaga wskazująca, czy pole może być nullem.

# Importowanie danych JSON

JSON (JavaScript Object Notation) to popularny format dla API i danych semi-strukturalnych. Databricks Spark obsługuje zarówno pojedyncze dokumenty JSON, jak i JSON Lines (JSON oddzielony znakami nowej linii).

## Ładowanie JSON

```
df_orders = (spark.read  
    .option("multiLine", "false")  
    .json(f"{raw_data_path}/orders.json")  
)
```

```
df_orders.printSchema()  
display(df_orders)
```

**multiLine=false:** każda linia to osobny dokument JSON (format JSON Lines)

**multiLine=true:** cały plik to pojedynczy dokument JSON (tablica lub zagnieżdżony obiekt)

## Specyfika formatu JSON

- **Zagnieżdżanie:** obiekty i tablice w strukturze danych
- **Brak sztywnego schematu:** różne dokumenty mogą mieć różne pola
- **Typy danych:** automatyczne wykrywanie (ciąg znaków, liczba, wartość logiczna, null)
- **Rozmiar:** JSON jest mniej wydajny niż Parquet (brak kompresji, format tekstowy)

Spark automatycznie analizuje zagnieżdżone struktury do StructType/ArrayType.

# Eksplorowanie zagnieżdzonych struktur JSON

JSON często zawiera zagnieżdżone obiekty i tablice. Spark reprezentuje je jako StructType (obiekt) i ArrayType (tablica). Dostęp do zagnieżdzonych pól wymaga specjalnej notacji.

## printSchema i eksploracja

```
df_orders.printSchema()

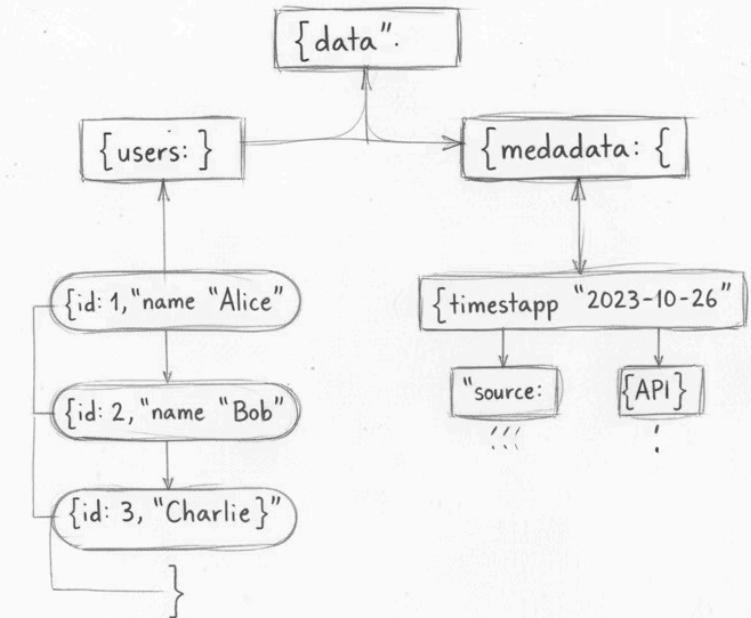
# Przykładowe wyjście:
# root
# |-- order_id: string
# |-- customer: struct
# |   |-- id: integer
# |   |-- name: string
# |-- items: array
# |   |-- element: struct
# |   |   |-- product_id: integer
# |   |   |-- quantity: integer
# |   |   |-- price: decimal(10,2)
```

## Notacja kropkowa i explode

```
# Dostęp do zagnieżdzonego pola
df_orders.select("customer.id",
                  "customer.name")

# Spłaszczenie tablicy
from pyspark.sql.functions import explode
df_items = df_orders.select(
    "order_id",
    explode("items").alias("item")
)
df_items.select("order_id",
                "item.product_id", "item.quantity")
```

explode: jeden wiersz na element tablicy.  
Zwiększa liczbę wierszy.



# DEMO

Zobaczmy Databricks w akcji!

# Transformacje Danych

W tej sekcji przyjrzymy się kluczowym operacjom transformacji danych w Databricks. Omówimy techniki takie jak filtrowanie, agregacje, łączenie zbiorów danych i czyszczenie danych, stosując podejścia PySpark i SQL.

# Transformacje Kolumn – wybór i projekcja

## Wybór i aliasy w PySpark

```
from pyspark.sql.functions import col

df_projection = df_customers.select(
    col("customer_id"),
    col("first_name").alias("fname"),
    col("last_name").alias("lname"),
    col("email"),
    col("country")
)
```

```
# Alternatywna składnia
df_projection2 = df_customers.select(
    "customer_id",
    "first_name",
    "country"
)
```

## Odpowiednik SQL

```
%sql
SELECT
    customer_id,
    first_name AS fname,
    last_name AS lname,
    email,
    country
FROM customers
```

Warstwa projekcji w potokach danych: wybieranie tylko niezbędnych kolumn zmniejsza zużycie pamięci i koszty transferu. Unikaj używania SELECT \* w środowisku produkcyjnym – jawną listą kolumn służy jako dokumentacja umowy.

# Dodawanie i usuwanie kolumn

## withColumn – tworzenie nowych kolumn

```
from pyspark.sql.functions import lit, expr,  
concat  
  
df_extended =  
df_customers.withColumn("country_code",  
lit("PL")) # Tworzy kolumnę z kodem kraju 'PL'  
df_extended =  
df_extended.withColumn("full_name",  
concat(col("first_name"), lit(" "),  
col("last_name"))) # Łączy imię i nazwisko w  
nową kolumnę  
df_extended =  
df_extended.withColumn("email_domain",  
expr("substring_index(email, '@', -1)")) #  
Wyodrębnia domenę e-mail z adresu e-mail
```

withColumn dodaje nową kolumnę lub nadpisuje istniejącą. Każde wywołanie zwraca nową, niezmienną DataFrame.

## Aktualizacja istniejących kolumn

```
from pyspark.sql.functions import upper, trim  
  
df_normalized =  
df_customers.withColumn("country",  
upper(trim(col("country")))) # Normalizuje  
wartości 'country' do wielkich liter i usuwa  
białe znaki  
df_normalized =  
df_normalized.withColumn("email",  
lower(col("email")))) # Normalizuje wartości  
'email' do małych liter
```

Wzorzec: normalizowanie wartości poprzez nadpisywanie kolumn. Zapewnia spójność przed operacjami łączenia i agregacji.

## drop – usuwanie kolumn

```
df_minimal =  
df_extended.drop("country_code",  
"email_domain") # Usuwa kolumny  
'country_code' i 'email_domain'
```

Usuwanie kolumn zmniejsza szerokość DataFrame. Przydatne po obliczeniach pośrednich, gdy tymczasowe kolumny nie są potrzebne w dalszej części przetwarzania.

# Wartości NULL w Agregacjach – Pułapki

Wartości NULL w agregacjach zachowują się nieintuicyjnie i mogą prowadzić do nieprawidłowych wyników biznesowych. Zrozumienie semantyki NULL przed agregacją jest kluczowe.

## Zachowanie funkcji agregujących

- **COUNT(\*)**: zlicza wszystkie wiersze, włączając NULL
- **COUNT(column)**: zlicza tylko wartości niebędące NULL
- **SUM/AVG**: ignorują NULL (nie traktują jako zero!)
- **MIN/MAX**: ignorują NULL

```
# Niebezpieczne
df.groupBy("customer_id").agg(sum("amount"))
# Jeśli klient ma tylko wartości NULL dla kwot →
wynik to NULL, nie 0!
```

## Rozwiążanie – użyj coalesce przed agregacją

```
from pyspark.sql.functions import coalesce, lit

df_safe = df.withColumn(
    "amount_clean",
    coalesce(col("amount"), lit(0))
)

df_agg = df_safe.groupBy("customer_id").agg(
    sum("amount_clean").alias("total_amount")
)
```

Zawsze weryfikuj semantykę NULL dla danych biznesowych: czy NULL = "brakujące dane" czy NULL = "zero"?

Name	Age	City	Score
Alice	25	New York	95
Bob	30	London	NULL
Charlie	NULL	Paris	72
David	40	NULL	Berlin
Eve		NULL	NULL

# Logika warunkowa – when/otherwise

Tworzenie kategorii biznesowych i flag w oparciu o wiele warunków. Funkcja `when()` w PySpark odpowiada instrukcji CASE WHEN w SQL.

## PySpark when/otherwise

```
from pyspark.sql.functions import when

df_tiered = df_customers.withColumn(
    "customer_tier",
    when(col("total_purchases") > 10000, "Diamond")
    .when(col("total_purchases") > 5000, "Platinum")
    .when(col("total_purchases") > 1000, "Gold")
    .otherwise("Silver")
)

df_flagged = df_tiered.withColumn(
    "is_vip",
    when(col("customer_tier").isin("Diamond", "Platinum"), True)
    .otherwise(False)
)
```

## Odpowiednik SQL

```
%sql
SELECT *,
CASE
    WHEN total_purchases > 10000 THEN 'Diamond'
    WHEN total_purchases > 5000 THEN 'Platinum'
    WHEN total_purchases > 1000 THEN 'Gold'
    ELSE 'Silver'
END AS customer_tier,
CASE
    WHEN customer_tier IN ('Diamond', 'Platinum') THEN true
    ELSE false
END AS is_vip
FROM customers
```

Kolejność warunków ma znaczenie – zwycięża pierwszy pasujący warunek. Testuj przypadki brzegowe (`NULL`, zero, wartości graniczne).

# Mapowanie wartości do kategorii

## Mapowanie słownikowe w PySpark

```
from pyspark.sql.functions import when

country_mapping = {
    "PL": "Poland",
    "US": "United States",
    "DE": "Germany",
    "FR": "France"
}

# Konwertuj na wyrażenie SQL
mapping_expr = " ".join([
    f"WHEN country = '{k}' THEN '{v}'"
    for k, v in country_mapping.items()
])

df_mapped = df.withColumn(
    "country_name",
    expr(f"CASE {mapping_expr} ELSE 'Other' END")
)
```

## Rozgłoś słownik dla dużych mapowań

```
from pyspark.sql.functions import broadcast

df_mapping = spark.createDataFrame(
    country_mapping.items(),
    ["code", "name"]
)

df_joined = df.join(
    broadcast(df_mapping),
    df.country == df_mapping.code,
    "left"
)
```

Dla złożonych mapowań (ponad 100 kategorii): użyj tabeli referencyjnej w Delta i dołącz zamiast CASE. Ułatwia to konserwację i aktualizację mapowań bez zmian w kodzie.

# Operacje tekstowe – Czyszczenie i normalizacja

## trim, upper, lower

```
from pyspark.sql.functions import  
trim, upper, lower  
  
df_clean = df.withColumn("country",  
upper(trim(col("country"))))  
df_clean =  
df_clean.withColumn("email",  
lower(trim(col("email"))))
```

Usuwa wiodące/końcowe białe znaki, normalizuje wielkość liter. Kluczowe przed opercjami łączenia danych – "Poland" != "Poland".

## concat i concat\_ws

```
from pyspark.sql.functions import  
concat, concat_ws, lit  
  
df_full = df.withColumn("full_name",  
concat(col("first_name"), lit(" "),  
col("last_name")))  
df_address =  
df.withColumn("full_address",  
concat_ws(", ", col("street"), col("city"),  
col("country")))
```

concat: łączy z literałami. concat\_ws: separator jest używany tylko między wartościami niepustymi.

## substring i split

```
from pyspark.sql.functions import  
substring, split  
  
df_domain =  
df.withColumn("email_domain",  
split(col("email"), "@").getItem(1))  
df_prefix =  
df.withColumn("country_prefix",  
substring(col("country"), 1, 2))
```

Wydobywa części ciągów znaków. split zwraca tablicę, getItem(idx) wybiera element z tej tablicy.

# Regex – Zaawansowane Operacje Tekstowe

## regexp\_extract – ekstrakcja wzorców

```
from pyspark.sql.functions import regexp_extract

# Ekstrakcja kodu pocztowego z adresu
df_zip = df.withColumn(
    "zip_code",
    regexp_extract(col("address"), r"(\d{2}-\d{3})", 1)
)

# Walidacja adresu e-mail
df_valid = df.withColumn(
    "is_valid_email",
    regexp_extract(col("email"), r"^\w\.-+@\w\.-+\.\w+$", 0) != ""
)
```

## regexp\_replace – czyszczenie

```
from pyspark.sql.functions import regexp_replace

# Usuwanie znaków innych niż alfanumeryczne
df_clean = df.withColumn(
    "phone_clean",
    regexp_replace(col("phone"), r"[\^d]", "")
)

# Normalizacja spacji
df_normalized = df.withColumn(
    "text_clean",
    regexp_replace(col("text"), r"\s+", " ")
)
```

Kiedy **regex jest dobrym narzędziem**: wykrywanie formatu (e-mail, telefon, kody pocztowe), ekstrakcja wzorców (ID, kody), czyszczenie znaków specjalnych.  
Kiedy **unikać**: proste operacje (trim, upper są wystarczające), ścieżki krytyczne dla wydajności (regex jest kosztowny), złożone parsowanie (użyj dedykowanych bibliotek).

# Filtrowanie – funkcje `filter()` i `where()` ze złożoną logiką

Filtrowanie wierszy na podstawie warunków to podstawowa operacja w każdym potoku danych. Spark oferuje elastyczne API dla prostych i złożonych predykatów.

## Proste Filtrowanie

```
# PySpark
df_pl = df_customers.filter(col("country") == "PL")
df_active = df_customers.where(col("is_active") == True)
df_high_value = df_customers.filter(col("total_purchases") > 5000)
```

```
# SQL
%sql
SELECT * FROM customers
WHERE country = 'PL'
AND is_active = true
AND total_purchases > 5000
```

filter() i where() to aliasy – identyczna funkcjonalność.

## Złożone Warunki

```
# Logika AND (&), OR (|), NOT (~)
df_complex = df_customers.filter(
    (col("country") == "PL") &
    (col("total_purchases") > 1000)
)

df_or = df_customers.filter(
    (col("country") == "PL") |
    (col("country") == "DE")
)

df_not = df_customers.filter(
    ~col("email").isNull()
)
```

**Uwaga:** Używaj nawiasów! Bez nich kolejność operatorów może prowadzić do nieoczekiwanych wyników.

# Przypadek Biznesowy – Złożone Filtry

## Przypadek użycia: aktywni klienci premium z USA

```
# PySpark  
# Filtrowanie klientów z USA, którzy są w tierach "Diamond" lub "Platinum", są aktywni,  
# mają podany adres e-mail i dokonali zakupów o wartości powyżej 10000.  
df_target_segment = df_customers.filter(  
    (col("country") == "US") &  
    (col("customer_tier").isin("Diamond", "Platinum")) &  
    (col("is_active") == True) &  
    (col("email").isNotNull()) &  
    (col("total_purchases") > 10000)  
)  
  
print(f"Target segment size: {df_target_segment.count()}")
```

Ten segment może być wykorzystany do: kampanii marketingowych e-mail, spersonalizowanych ofert, programów lojalnościowych, analizy retencji. Zawsze dokumentuj logikę biznesową stojącą za filtrami – kod sam się nie wyjaśnia.

# Podstawy sortowania – orderBy i sort

## Metody sortowania w PySpark

PySpark oferuje dwie główne metody sortowania: `orderBy()` i `sort()`. Obie funkcje są semantycznie równoważne i pozwalają na sortowanie DataFrame'a według jednej lub wielu kolumn.

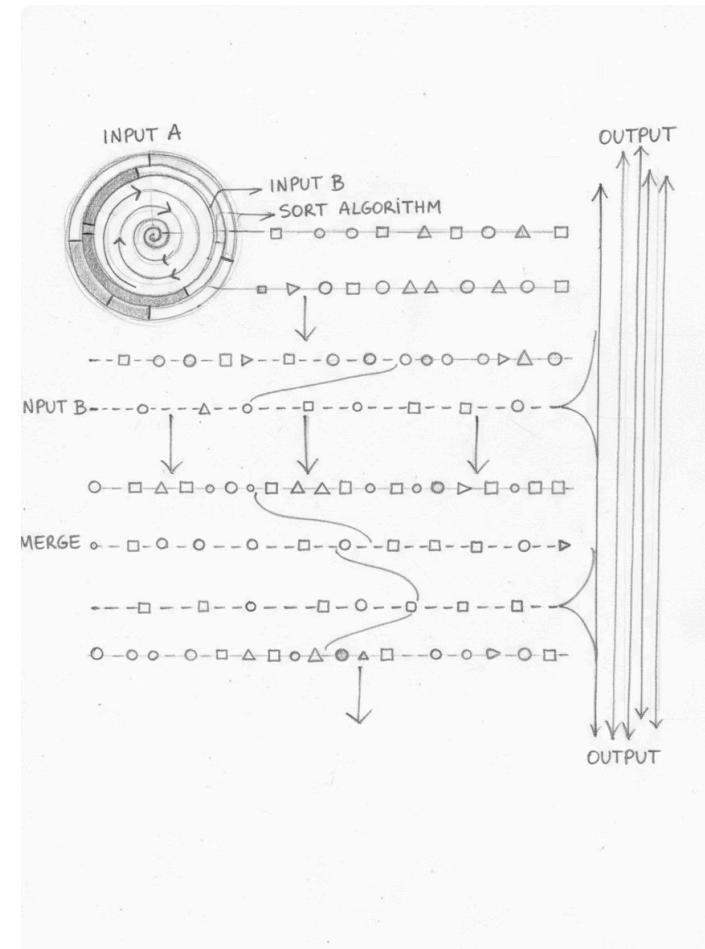
Sortowanie rosnące jest domyślne, ale można je jawnie określić za pomocą metody `asc()`. Do sortowania malejącego użyj `desc()`.

```
df.orderBy("amount", ascending=False)  
df.sort(col("date").desc())
```

W SQL składnia jest bardziej konwencjonalna, używając klauzuli ORDER BY z opcjonalnymi słowami kluczowymi ASC lub DESC.

### Uwaga dotycząca wydajności

Sortowanie wymaga przetasowania danych między partycjami, co znaczco wpływa na czas wykonania zapytania, zwłaszcza dla dużych zbiorów danych.



# Stronicowanie wyników – limit i offset

Stronicowanie jest kluczowe dla interfejsów API zwracających duże zbiory danych oraz dla interaktywnych raportów. W środowisku rozproszonym wdrożenie stronicowania wymaga dobrze przemyślanej strategii.

## Podejście SQL

Standardowa składnia SQL wykorzystuje LIMIT i OFFSET:

```
SELECT * FROM orders  
ORDER BY order_date DESC  
LIMIT 100 OFFSET 200
```

Zwraca 100 rekordów, zaczynając od 201. pozycji.

## Implementacja PySpark

PySpark oferuje metodę `limit()`, ale brakuje natywnego `offset`:

```
df.orderBy("date").limit(100)
```

Dla `offset` należy użyć funkcji okienkowych lub zapisać do tabeli tymczasowej i użyć SQL.

### Ostrzeżenie

OFFSET wymaga przetwarzania wszystkich poprzedzających rekordów, co czyni go kosztownym dla dużych wartości `offset`. Rozważ alternatywne podejścia, takie jak stronicowanie oparte na kursorach.

# Wpływ sortowania na plan wykonania

## Analiza bez sortowania

Zapytanie bez klauzuli ORDER BY może wykonywać się równolegle w ramach każdej partycji bez konieczności wymiany danych między węzłami.

## Sortowanie przez tasowanie (shuffle sort)

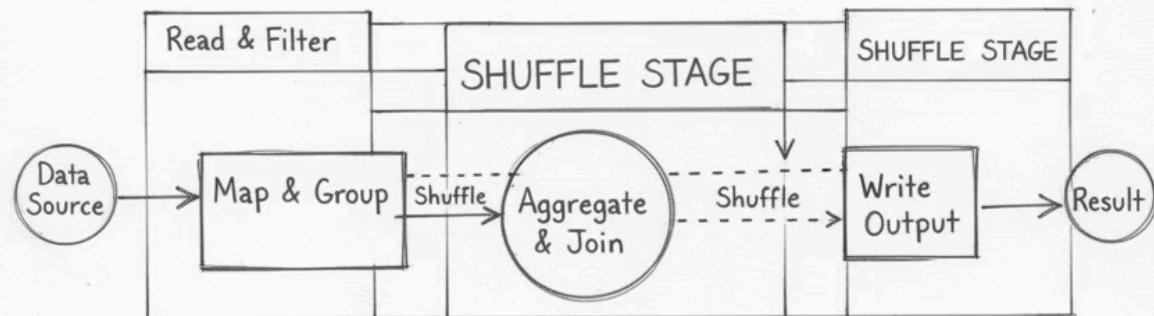
Globalne sortowanie wymaga tasowania (shuffle) – wszystkie dane muszą być ponownie dystrybuowane między wykonawcami na podstawie klucza sortującego.

## Koszty operacji

Tasowanie generuje ruch sieciowy, zapisy na dysku i zwiększa zużycie pamięci. Może być wąskim gardłem dla dużych zbiorów danych.

Użyj `df.explain()` lub interfejsu Spark UI, aby wyświetlić fizyczny plan i zidentyfikować etapy tasowania. Opcja `explain(mode='formatted')` dostarcza szczegółowych informacji o kosztach.

## SPARK EXECUTION PLAN



# Podstawowa składnia groupBy + agg

Agregacje są fundamentem analizy danych. Operacja `groupBy` grupuje rekordy według wartości w określonych kolumnach, a następnie funkcje agregujące obliczają statystyki dla każdej grupy.

## Typowe funkcje agregujące

- **sum()** – suma wartości liczbowych
- **avg() / mean()** – średnia arytmetyczna
- **count()** – liczba rekordów (lub niepustych wartości)
- **min() / max()** – wartości skrajne
- **stddev()** – odchylenie standardowe
- **collect\_list()** – lista wartości w grupie

## Przykład PySpark

```
df.groupBy("country", "product")
  .agg(
    sum("amount").alias("total_sales"),
    avg("quantity").alias("avg_qty"),
    count("order_id").alias("order_count")
  )
```

Ten kod grupuje zamówienia według kraju i produktu, obliczając całkowitą sprzedaż, średnią ilość i liczbę zamówień.

# Obsługa wartości NULL w agregacjach

Wartości NULL wymagają szczególnej uwagi w agregacjach, ponieważ większość funkcji agregujących je ignoruje, co może prowadzić do nieoczekiwanych wyników.

## Domyślne zachowanie

Funkcje takie jak `sum()`, `avg()` pomijają wartości `NULL`. `count(col)` liczy tylko wartości niebędące `NULL`, `count(*)` liczy wszystkie rekordy.

## Wartości odstające

Ekstremalne wartości mogą silnie wpływać na średnie. Rozważ użycie median lub percentylów jako bardziej odpornych miar.

## Strategie obsługi

Przed agregacją: wypełnij `NULLe` sensowną wartością (`fillna`). Podczas agregacji: użyj `coalesce(col, 0)`. Po agregacji: oznacz grupy z dużą liczbą wartości `NULL`.

# PySpark vs SQL – Różnice w Składni Agregacji

Operacja	PySpark	SQL
Grupowanie	<code>df.groupBy("col")</code>	<code>GROUP BY col</code>
Suma	<code>sum("amount")</code>	<code>SUM(amount)</code>
Zliczanie	<code>count("*")</code>	<code>COUNT(*)</code>
Średnia	<code>avg("price")</code>	<code>AVG(price)</code>
Wiele Agregacji	<code>.agg(sum(...), avg(...))</code>	<code>SUM(...), AVG(...)</code>
Alias	<code>.alias("name")</code>	<code>AS name</code>

Obie składnie są semantycznie równoważne i generują identyczne plany wykonania. Wybór zależy od preferencji zespołu i kontekstu projektu.

# Zaawansowane Agregacje – Wiele Agregacji Jednocześnie

W praktycznych scenariuszach często musimy obliczać wiele metryk dla tych samych grup. PySpark pozwala na eleganckie wyrażenie takich zapytań.

## Używanie słownika w agg()

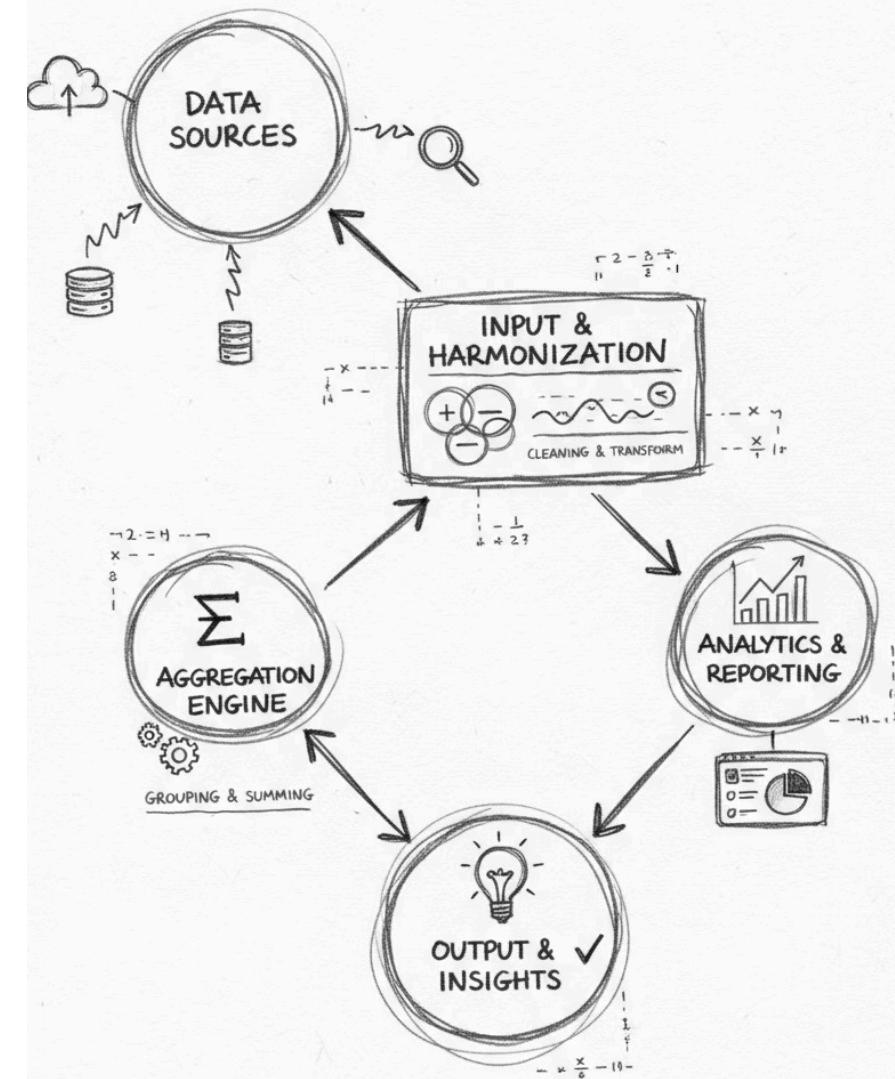
```
df.groupBy("customer_id").agg({  
    "amount": "sum",  
    "order_id": "count",  
    "discount": "avg"  
})
```

Zwięzła notacja, ale nazwy kolumn są automatyczne (np. "sum(amount)").

## Jawne Aliases (pseudonimy)

```
df.groupBy("customer_id").agg(  
    sum("amount").alias("total"),  
    count("order_id").alias("orders"),  
    avg("discount").alias("avg_disc")  
)
```

Pełna kontrola nad nazwami wynikowych kolumn – preferowane podejście.



# Filtrowanie wyników agregacji – HAVING kontra filter

## Klauzula HAVING w SQL

W SQL filtrowanie grup odbywa się za pomocą klauzuli HAVING:

```
SELECT country, SUM(sales)
      FROM orders
 GROUP BY country
 HAVING SUM(sales) > 100000
```

HAVING operuje na wynikach agregacji, WHERE na danych wejściowych.

## Filtrowanie w PySpark po agregacji

W PySpark nie ma klauzuli HAVING – stosujemy filter do wyniku groupBy:

```
df.groupBy("country")
 .agg(sum("sales").alias("total"))
 .filter(col("total") > 100000)
```

Kolejność operacji jest jawną i naturalną.

- WHERE/filter przed groupBy zmniejsza ilość danych wejściowych. Filter po agregacji zmniejsza liczbę grup w wyniku. Obie optymalizacje są wartościowe.

# Te same transformacje w obu podejściach

## API PySpark DataFrame

```
result = (  
    df.filter(col("status") == "active")  
    .groupBy("category")  
    .agg(sum("revenue").alias("total"))  
    .orderBy(col("total").desc())  
)
```

## SQL

```
SELECT category,  
       SUM(revenue) as total  
  FROM df  
 WHERE status = 'active'  
 GROUP BY category  
 ORDER BY total DESC
```

- Programistyczne, z typowaniem
- Łatwe do testowania jednostkowego
- Integracja z kodem Python

- Deklaratywne, znane analitykom
- Czytelne dla użytkowników nietechnicznych
- Standard w ekosystemie BI

Oba podejścia generują identyczny plan wykonania w Spark – różnica leży w sposobie wyrażania intencji.

# Wersjonowanie Kodu i Testowanie

Aspekt	Notebooki PySpark	Skrypty SQL
Kontrola Wersji	Repozytorium Git, zmiany na poziomie komórek	Repozytorium Git, standardowe porównanie tekstowe
Testy Jednostkowe	pytest, łatwe mockowanie	Wymaga frameworka testowego (dbt, itp.)
CI/CD	Automatyzacja przez CLI Databricks	Wdrożenie poprzez migracje SQL
Przegląd kodu	Pull requesty, porównania notebooków	Standardowe PR dla plików SQL
Dokumentacja	Markdown w notebooku	Komentarze lub dokumenty zewnętrzne

Podejście hybrydowe: logika w funkcjach PySpark (testowalna), orkiestracja w SQL (dostępna dla BI).

# Zalecenia: Kiedy preferować SQL

## Warstwy semantyczne i raportowanie

Widoki SQL są naturalnym interfejsem dla narzędzi BI (Tableau, Power BI). Używaj SQL do warstw prezentacji w Lakehouse.

## Analiza ad-hoc przez analityków

Analitycy biznesowi znają SQL lepiej niż Python. SQL w Databricks SQL Editor jest dostępny bez znajomości Sparka.

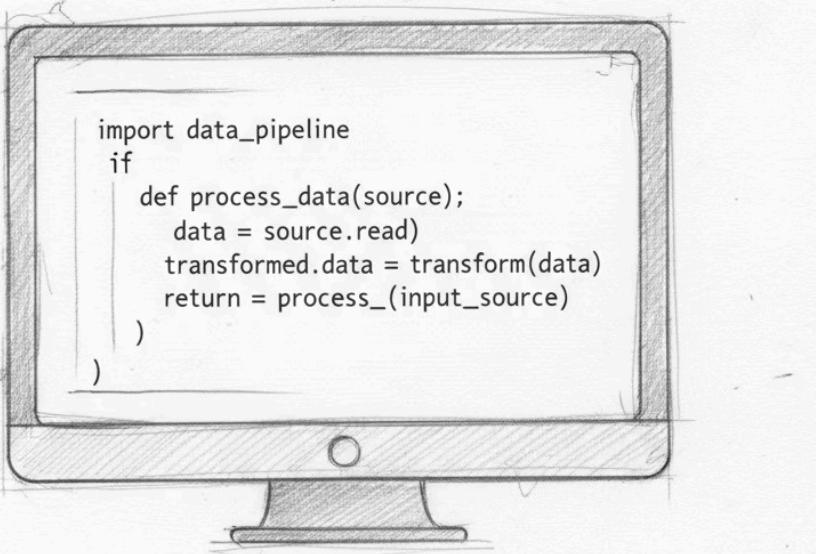
## Prosta logika transformacji

Dla prostych filtrów, łączeń i agregacji, SQL jest zwięzły i czytelny. Niepotrzebnie nie dodawaj złożoności.

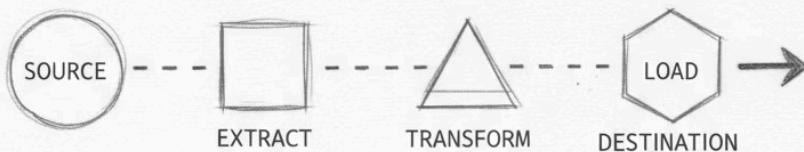
## Wspólne definicje biznesowe

Scentralizowane widoki z regułami biznesowymi (KPI, metryki) w Unity Catalog są dostępne dla wszystkich zespołów.





```
import data_pipeline
if
    def process_data(source);
        data = source.read)
        transformed.data = transform(data)
        return = process_(input_source)
    )
)
```



# Zalecenia: Kiedy preferować PySpark

## Złożona logika transformacji

Funkcje UDF, niestandardowa logika, złożone warunki, parsowanie JSON – łatwiej w Pythonie niż w SQL.

## Potoki wymagające testów

Produkcyjne potoki ETL/ELT z testami jednostkowymi, testami integracyjnymi i kontrolą jakości – PySpark + pytest.

## Integracja z ML

Inżynieria cech dla modeli ML, wywołania MLlib, scoring – naturalnie w PySpark.

## Parametryzacja i ponowne użycie

Funkcje w Pythonie z parametrami, ponowne użycie w różnych projektach, współdzielone moduły w przestrzeni roboczej.

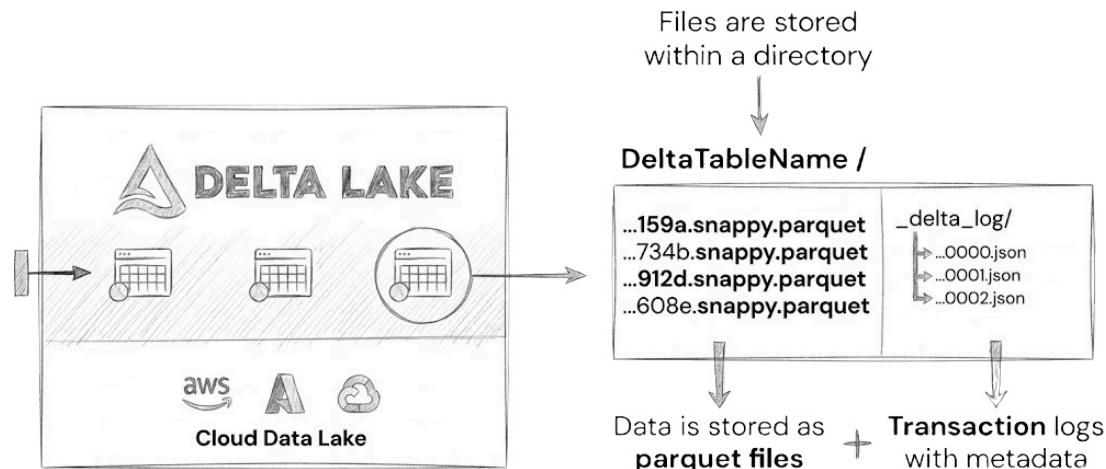
# DEMO

Zobaczmy Databricks w akcji!

# Delta Tables

Szczegółowa analiza Delta Tables

# Delta Log – Serce Transakcyjności



Delta Log to uporządkowana sekwencja plików JSON przechowywanych w katalogu `_delta_log`. Każdy plik reprezentuje pojedynczą transakcję i zawiera pełne informacje o zmianach w tabeli.

## Struktura wpisu Delta Logu:

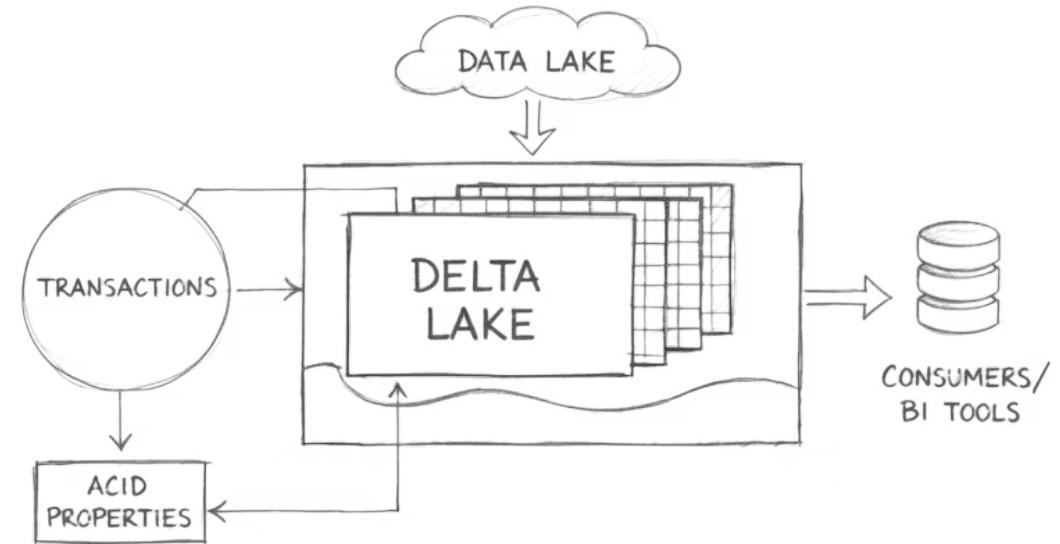
- Lista dodanych plików Parquet z metadynamami
- Lista usuniętych plików
- Operacje schematu (dodawanie/usuwanie kolumn)
- Metadane operacji (sygnatura czasowa, użytkownik, operacja)
- Statystyki danych (wartości min/max dla pomijania danych)

Każda operacja odczytu rozpoczyna się od odczytania Delta Logu w celu ustalenia bieżącego stanu tabeli i listy aktywnych plików.

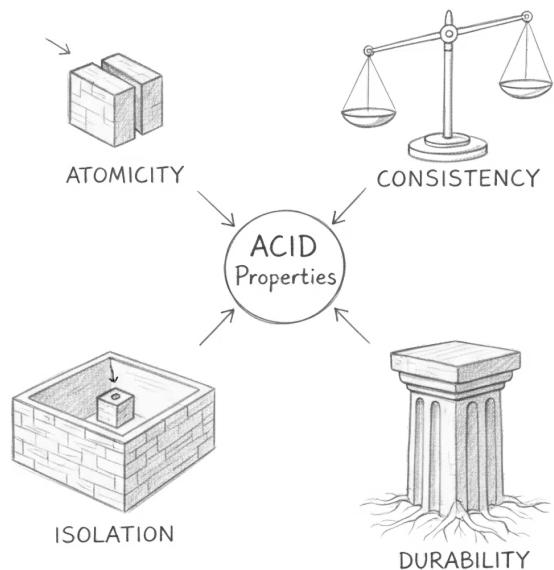
# Czym jest Delta Lake?

Delta Lake to otwarta warstwa przechowywania danych, która wprowadza niezawodność do Data Lakes. Łączy w sobie najlepsze cechy tradycyjnych baz danych i Data Lakes, zapewniając:

- **ACID Transactions**: Zapewnia atomowość, spójność, izolację i trwałość operacji, co gwarantuje integralność danych.
- **Schema Enforcement**: Pomaga utrzymać jakość danych poprzez blokowanie zapisów, które nie są zgodne ze schematem tabeli.
- **Data Versioning**: Umożliwia dostęp do historycznych wersji tabel, ułatwiając audyt, wycofywanie zmian i odtwarzanie danych.
- **Scalable Metadata**: Obsługuje tabele o bilionach partycji i plików.
- **Unified Streaming and Batch Processing**: Pozwala na korzystanie z tego samego Data Lake do przetwarzania danych w czasie rzeczywistym i batchu.



# Właściwości ACID w Data Lake



## Atomowość

Wszystkie operacje są atomowe – albo całkowity sukces, albo całkowite wycofanie. Brak częściowych zapisów lub niespójnych stanów.

## Spójność

Wymuszanie schematu gwarantuje, że każda transakcja pozostawia tabelę w prawidłowym stanie zgodnym z zdefiniowanym schematem.

## Izolacja

Izolacja możliwa do serializacji – współbieżne operacje nie zakłócają się nawzajem. Czytelnicy zawsze widzą spójny zrzut.

## Trwałość

Zatwierdzone transakcje są trwałe i przetrwają awarie systemu dzięki Delta Log przechowywanemu w pamięci obiektowej.

# DESCRIBE HISTORY – Audit transakcji

```
DESCRIBE HISTORY bronze_transactions;
```

-- Wynik pokazuje pełną historię operacji:

	version	timestamp	operation	operationParameters
0		2024-01-15 10:00:00	CREATE	{mode: ErrorIfExists}
1		2024-01-15 11:30:00	WRITE	{mode: Append}
2		2024-01-15 14:00:00	MERGE	{predicate: ["id"]}
3		2024-01-15 16:45:00	DELETE	{predicate: ["status='invalid'"]}
4		2024-01-15 18:00:00	OPTIMIZE	{predicate: []}

Komenda DESCRIBE HISTORY ujawnia pełną ścieżkę audytu wszystkich operacji wykonanych na tabeli Delta. Każda wersja jest niezmienną migawką, która może być używana do podróży w czasie (time travel), analizy kryminalistycznej lub odzyskiwania po awarii.

Historia transakcji jest automatycznie zarządzana przez Delta Lake i przechowywana zgodnie z konfiguracją okresu przechowywania (domyślnie 30 dni).

# Wymuszanie Schematu – Kontrola Jakości

**Wymuszanie Schematu** to mechanizm ochronny, który blokuje zapisy niezgodne z zdefiniowanym schematem tabeli. Delta Lake automatycznie waliduje każdy wiersz danych przed zapisem, sprawdzając typy danych, możliwość przyjmowania wartości NULL i inne ograniczenia.

## Typowe Scenariusze Blokowania

- Niezgodność typów danych (ciąg znaków zamiast liczby całkowitej)
- Brak wymaganych kolumn
- Próba zapisu wartości NULL do kolumny NOT NULL
- Nieprawidłowy format daty lub znacznika czasu

## Korzyści

- Wczesne wykrywanie błędów w potoku danych
- Zapobieganie propagacji błędnych danych
- Redukcja późniejszych kosztów naprawy
- Utrzymanie spójności referencyjnej

### SCHEMA EVOLUTION: ADDING COLUMNS

1. OLD SCHEMA	
id	name
1	John

2. ADD NULLABLE COLUMN		
id	email	
1	John@doe.com	
2		

3. BACKFILL & NOT NULL			
id	email		
1	John@doe.com		
2			

```
CREATE TABLE users (
    id INT,
    name TEXT );
```

```
ALTER TABLE users
ADD COLUMN
email TEXT;
```

```
UPDATE users SET email = '...'

ALTER TABLE users = ....
ALTER TABLE ALTER COLUMN
SET NOT NULL;
```

## Ewolucja Schematu – Bezpieczna Ewolucja

```
# Próba zapisu z nową kolumną - BŁĄD
df_new.write.format("delta").mode("append").save(path)
# AnalysisException: Wykryto niezgodność schematu
```

```
# Bezpieczne dodawanie nowej kolumny
df_new.write \
    .format("delta") \
    .mode("append") \
    .option("mergeSchema", "true") \
    .save(path)
# SUKCES - dodano nową kolumnę, stare rekordy mają wartość NULL
```

Ewolucja Schematu pozwala na kontrolowane zmiany w strukturze tabeli w czasie. Opcja `mergeSchema=true` umożliwia dodawanie nowych kolumn bez przebudowy całej tabeli. Stare rekordy automatycznie otrzymują wartość `NULL` dla nowych kolumn.

**Ograniczenia:** Ewolucja Schematu nie pozwala na usuwanie kolumn, zmianę typów danych ani zmianę możliwości przyjmowania wartości `NULL`. Te operacje wymagają pełnej migracji tabeli.



# Podróże w Czasie

Podróże w czasie przez historię danych

# Time Travel – Odczyt Historyczny

## Składnia Zapytań Historycznych

```
-- Odczyt według numeru wersji
```

```
SELECT * FROM table_name VERSION AS OF 0;
```

```
-- Odczyt według znacznika czasu
```

```
SELECT * FROM table_name  
TIMESTAMP AS OF '2024-01-15 10:00:00';
```

```
-- W API DataFrames
```

```
df = spark.read \  
.format("delta") \  
.option("versionAsOf", 0) \  
.load(path)
```

Time Travel to unikalna funkcja Delta Lake, która umożliwia odczyt danych z dowolnego punktu w historii tabeli. Każda wersja tabeli to niezmienna migawka, która pozostaje dostępna zgodnie z polityką retencji.

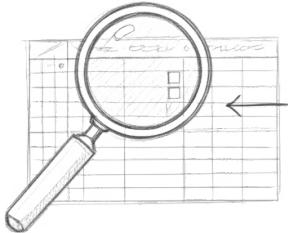
## Zastosowania Biznesowe

- Audyt regulacyjny i zgodność
- Analiza zmian danych w czasie
- Odtwarzanie przeszłych raportów
- Porównywanie wersji przed/po
- Testowanie logiki transformacji
- Odzyskiwanie danych po błędnych operacjach

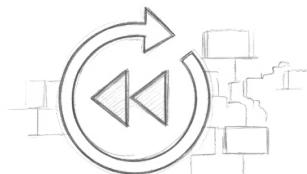
❑ Time Travel działa bez dodatkowych kosztów przechowywania — wykorzystuje te same pliki Parquet co bieżąca wersja

# Time Travel – Praktyczne Zastosowania

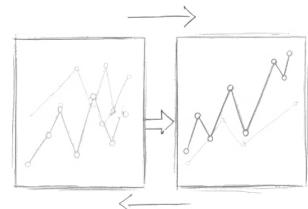
Funkcjonalność Time Travel w Delta Lake jest szeroko stosowana w codziennych operacjach na danych, umożliwiając nie tylko audyt, ale także szybką reakcję na błędy i analizę trendów.



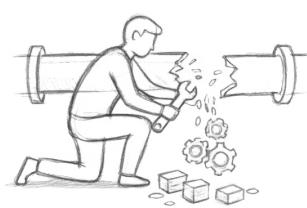
DATA AUDITING



ROLLBACKS



VERSION COMPARISONS



DATA PIPELINE  
DEBUGGING

## Przypadki Użycia:

Audytowanie zmian danych

Przywracanie stanu po błędnych operacjach

Porównywanie stanów danych między wersjami

Debugowanie potoków danych

## Praktyczne Przykłady:

-- Porównanie danych między wersjami  
`SELECT COUNT(*) as current_count FROM sales_table;`  
`SELECT COUNT(*) as yesterday_count FROM sales_table TIMESTAMP AS OF '2024-01-14';`

-- Znajdowanie różnic  
`SELECT * FROM sales_table EXCEPT SELECT * FROM sales_table VERSION AS OF 1;`

## Ograniczenia:

- Wpływ na wydajność przy bardzo starych wersjach
- Koszty przechowywania wersji historycznych
- Okres retencji (domyślnie 30 dni)

# Odzyskiwanie po Awarii – RESTORE TABLE

```
-- Przypadkowe usunięcie danych
```

```
DELETE FROM customer_orders WHERE order_date > '2024-01-01';
```

```
-- Usunięto 10 000 wierszy!
```

```
-- Weryfikacja szkód
```

```
SELECT COUNT(*) FROM customer_orders; -- 5 000 (było 15 000)
```

```
-- Szybkie przywrócenie do wersji sprzed operacji DELETE
```

```
RESTORE TABLE customer_orders TO VERSION AS OF 2;
```

```
-- Tabela przywrócona!
```

```
SELECT COUNT(*) FROM customer_orders; -- 15 000 ✓
```

Polecenie `RESTORE TABLE` umożliwia natychmiastowe przywrócenie tabeli do dowolnej historycznej wersji. W przeciwieństwie do klasycznego tworzenia kopii zapasowych/przywracania, operacja `RESTORE` działa w ciągu kilku sekund, ponieważ nie kopiuje danych – jedynie aktualizuje wskaźniki w `Delta Log`.

`RESTORE` jest kluczowym narzędziem do odzyskiwania po awarii w potokach produkcyjnych, gdzie błędne transformacje mogą wpływać na krytyczne procesy biznesowe.

# Operacje DML – UPDATE i DELETE

Delta Lake wprowadza natywne wsparcie dla operacji UPDATE i DELETE w Data Lake – funkcjonalność niedostępna w tradycyjnych formatach big data, takich jak Parquet czy ORC. Operacje te są atomowe, transakcyjne i nie wymagają przepisywania całej tabeli.

## UPDATE – Modyfikacja Rekordów

```
UPDATE silver_customers
SET status = 'inactive',
    updated_at = current_timestamp()
WHERE last_purchase_date < '2023-01-01'
    AND status = 'active';

-- Zaktualizowano 1,234 wierszy
```

## DELETE – Usuwanie Rekordów

```
DELETE FROM bronze_events
WHERE event_timestamp < current_date() - INTERVAL 90 DAYS
    AND event_type = 'debug';

-- Usunięto 5,678 wierszy
```

Pod spodem Delta Lake przepisuje tylko te pliki Parquet, które zawierają zmodyfikowane rekordy, minimalizując I/O i utrzymując wydajność nawet dla dużych tabel.

# MERGE INTO – Upsert w Data Lake

```
MERGE INTO silver_products AS target
USING bronze_products_updates AS source
ON target.product_id = source.product_id
WHEN MATCHED AND source.price != target.price THEN
    UPDATE SET
        target.price = source.price,
        target.updated_at = current_timestamp()
WHEN NOT MATCHED THEN
    INSERT (product_id, name, price, created_at)
        VALUES (source.product_id, source.name, source.price, current_timestamp())
WHEN NOT MATCHED BY SOURCE AND target.is_active = true THEN
    UPDATE SET target.is_active = false;
```

Operacja MERGE INTO to najbardziej zaawansowana funkcjonalność DML w Delta Lake. Łączy ona operacje INSERT, UPDATE i DELETE w jedną atomową transakcję, umożliwiając implementację złożonych scenariuszy, takich jak Change Data Capture (CDC), deduplikacja i synchronizacja danych źródłowych w Data Lake.

# DEMO

Zobaczmy Databricks w akcji!

# Optymalizacja

Wydajność i skalowanie w Data Lake.

# Analiza Zapytań – EXPLAIN

```
query = """
SELECT customer_id, sum(amount)
FROM gold_sales
WHERE order_date >= '2024-01-01'
  AND region = 'EMEA'
GROUP BY customer_id
"""

spark.sql(query).explain(mode="extended")
```

Polecenie `explain()` ujawnia fizyczny plan wykonania zapytania przez Spark.

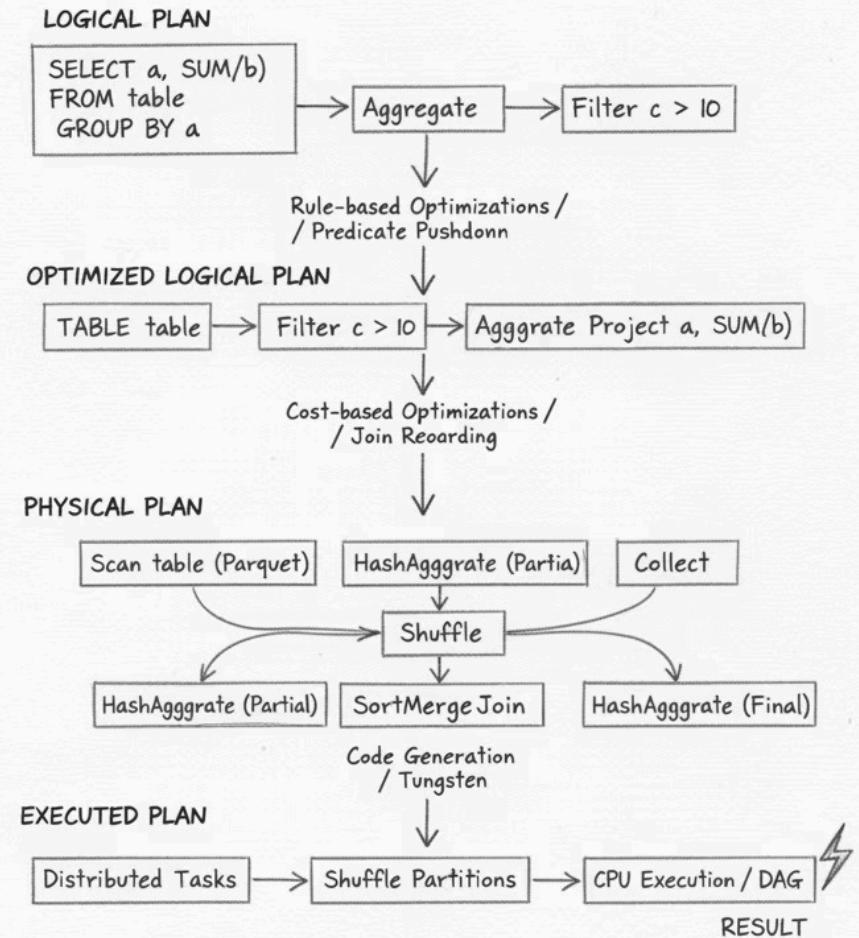
Kluczowe elementy do zidentyfikowania:

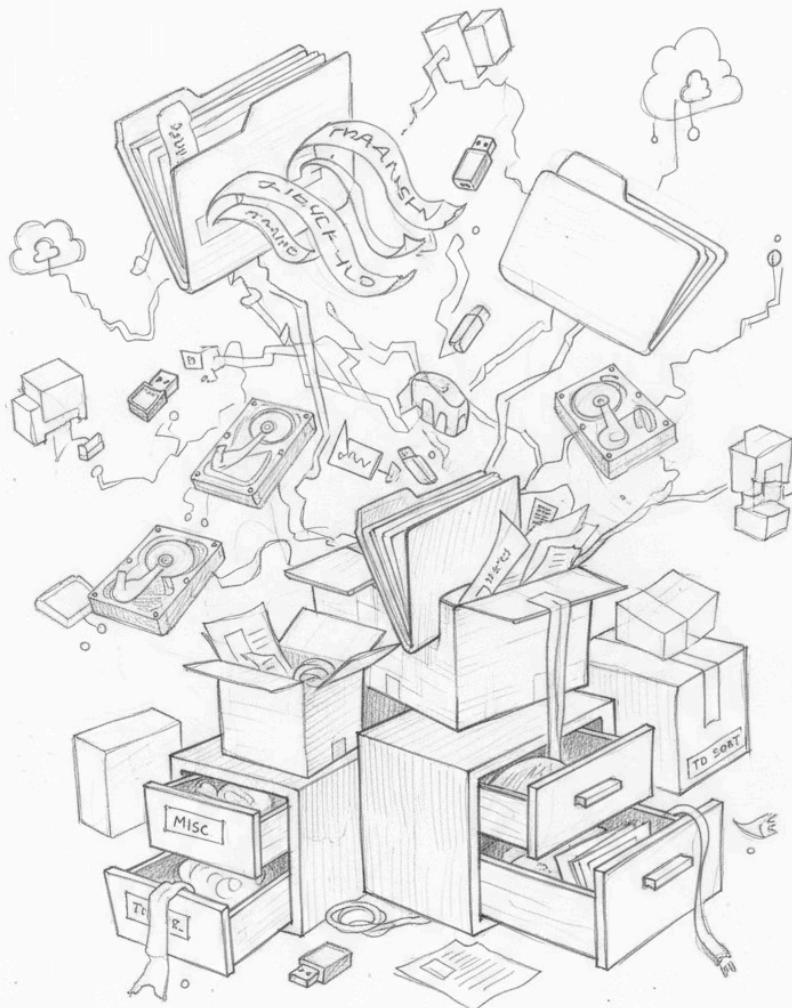
- **Predicate Pushdown:** Czy filtry docierają do miejsca przechowywania danych?
- **Column Pruning:** Czy odczytujemy tylko niezbędne kolumny?
- **Partition Pruning:** Czy pomijamy niepotrzebne partycje?
- **Shuffle:** Czy występuje kosztowna wymiana danych?

## ☐ Etapy Planu Fizycznego

Plan składa się z następujących etapów:

- FileScan – odczyt z Delta
- Filter – ewaluacja predykatów
- Project – selekcja kolumn
- Exchange – operacja shuffle (KOSZTOWNA)
- HashAggregate – agregacja





# Problem Małych Plików

Problem Małych Plików to klasyczny zabójca wydajności w Data Lakes. Spark działa optymalnie z plikami o rozmiarze od 128 MB do 1 GB. Tysiące małych plików (<10 MB) powoduje:

## Problemy z Wydajnością

- Narzut na otwieranie każdego pliku
- Nieefektywne wykorzystanie równoległości
- Wolniejsze wyszukiwanie metadanych w Delta Log
- Gorsza kompresja (każdy plik osobno)
- Mniejsza skuteczność pomijania danych

## Przyczyny małych plików

- Strumieniowanie micro-batch z wysoką częstotliwością
- Częste operacje INSERT bez kompresji
- MERGE INTO na małych partiach danych
- Wysoka kardynalność partycji
- Brak Auto Compaction

Rozwiązanie: Regularnie uruchamiaj OPTIMIZE lub włącz Auto Compaction.

# OPTIMIZE – Kompaktowanie

-- Ręczne kompaktowanie

```
OPTIMIZE gold_sales;
```

-- Skompaktowano 1247 plików do 23 plików

-- Kompaktowanie ZORDER (omówione na następnym slajdzie)

```
OPTIMIZE gold_sales ZORDER BY (order_date, region);
```

-- Automatyczne kompaktowanie (włącz na tabeli)

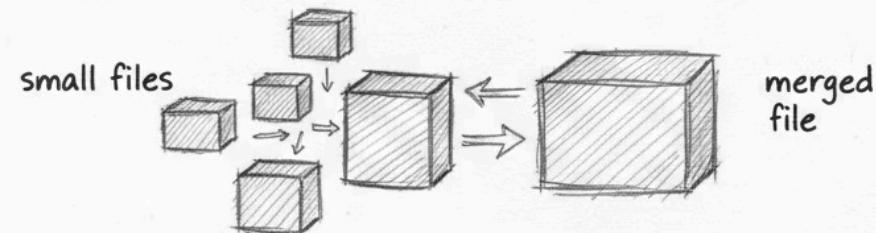
```
ALTER TABLE gold_sales SET TBLPROPERTIES (
  'delta.autoOptimize.optimizeWrite' = 'true',
  'delta.autoOptimize.autoCompact' = 'true'
);
```

-- Monitorowanie statusu plików

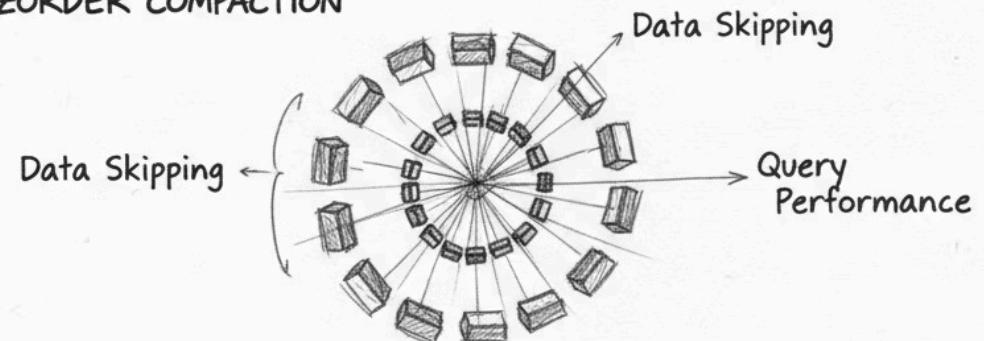
```
DESCRIBE DETAIL gold_sales;
```

-- liczba plików: 23, rozmiar w bajtach: 15.4GB

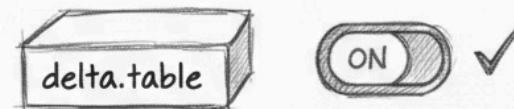
## MANUAL COMPACTION



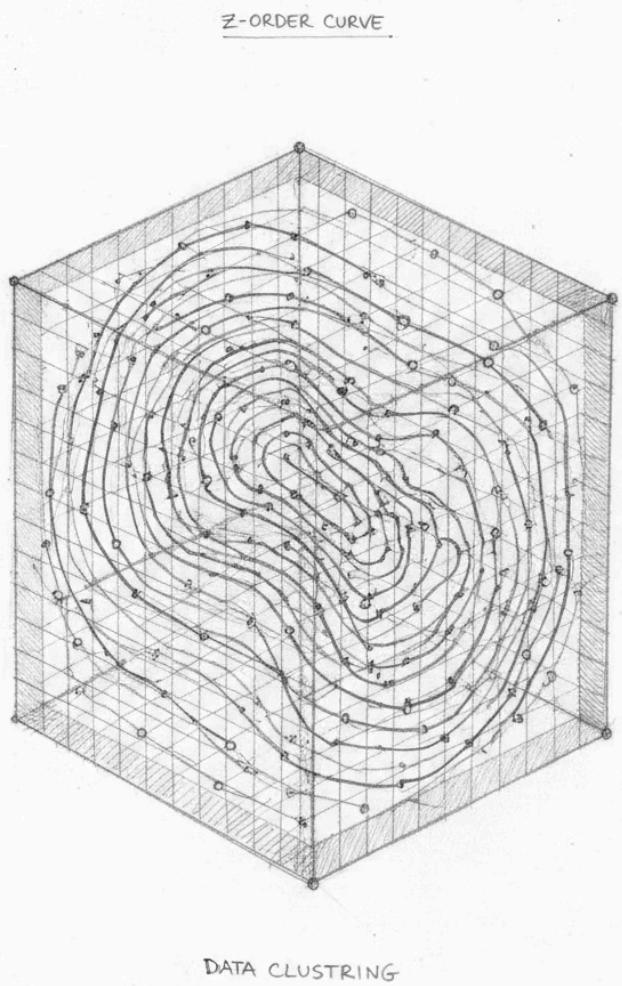
## ZORDER COMPACTON



## AUTO-COMPACTON



```
ALTER TABLE table-name SET TBLPROPERTIES  
(delta.autoOptimize.autoCompact' = 'true:')
```



# ZORDER BY – Klasteryzacja Wielowymiarowa

ZORDER BY to zaawansowana technika porządkowania danych w plikach Parquet, która optymalizuje pomijanie danych dla wielu kolumn jednocześnie. W przeciwieństwie do tradycyjnego partycjonowania (które działa tylko dla 1 kolumny), ZORDER umożliwia efektywne filtrowanie na 2-4 kolumnach.

```
-- ZORDER dla najczęściej filtrowanych kolumn  
OPTIMIZE customer_events  
ZORDER BY (event_date, customer_id, event_type);
```

```
-- Delta Lake automatycznie generuje statystyki (min/max) dla kolumn posortowanych przez  
ZORDER  
-- Umożliwia pomijanie danych: "Ten plik nie zawiera danych dla tego filtra"
```

## Kiedy używać ZORDER

- Kolumny często pojawiające się w klauzulach WHERE
- Kolumny o wysokiej kardynalności (daty, ID)
- Wiele wymiarów filtrowania w typowych zapytaniach

## Ograniczenia

- Maksymalnie 3-4 kolumny (skuteczność spada)
- Wymaga ponownego uruchomienia po każdym dużym zapisie
- Nie działa dla bardzo wysokiej kardynalności (UUID)

# Liquid Clustering – Nowoczesna Alternatywa

```
-- Tworzenie tabeli z Liquid Clustering
```

```
CREATE TABLE customer_events (
```

```
    event_id STRING,
```

```
    customer_id STRING,
```

```
    event_date DATE,
```

```
    event_type STRING,
```

```
    payload STRING
```

```
) USING DELTA
```

```
CLUSTER BY (event_date, customer_id);
```

```
-- Automatyczne przyrostowe klasteryzowanie przy każdym zapisie
```

```
INSERT INTO customer_events VALUES (...);
```

```
-- Dane automatycznie porządkowane zgodnie z CLUSTER BY
```

```
-- Zmiana kolumn klasteryzujących (niedostępne w ZORDER!)
```

```
ALTER TABLE customer_events CLUSTER BY (event_type, event_date);
```

Liquid Clustering to alternatywa nowej generacji dla partycjonowania i ZORDER. Automatycznie zarządza układem danych bez potrzeby ręcznego OPTIMIZE. Pozwala na zmianę strategii klasteryzowania bez przepisywania całej tabeli.

# Liquid Clustering - Zalety

## Automatyczne Zarządzanie

Przyrostowe klasteryzowanie z każdą operacją zapisu. Brak potrzeby planowania zadań OPTIMIZE.

## Elastyczność

Możliwość zmiany kolumn CLUSTER BY bez pełnej migracji tabeli. Dostosowanie do zmieniających się wzorców zapytań.

## Lepsza Wydajność

Bardziej efektywne pomijanie danych niż tradycyjne partycjonowanie. Eliminacja problemu eksplozji partycji.

## Prostota

Jedna technika zamiast kombinacji partycjonowania + ZORDER. Mniej punktów decyzyjnych dla inżynierów danych.

**Rekomendacja:** Użyj Liquid Clustering dla nowych tabel produkcyjnych. Dla starszych tabel zalecana jest stopniowa migracja z partycjonowania/ZORDER.

# Broadcast Hash Join – Optymalizacja Złączeń

## Jak działa Broadcast Join

Mała tabela jest kopiowana (broadcast) do każdego węzła wykonawczego. Eliminuje to kosztowne tasowanie (SHUFFLE) dużej tabeli. Działa tylko dla małych tabel <10GB (konfigurowalne).

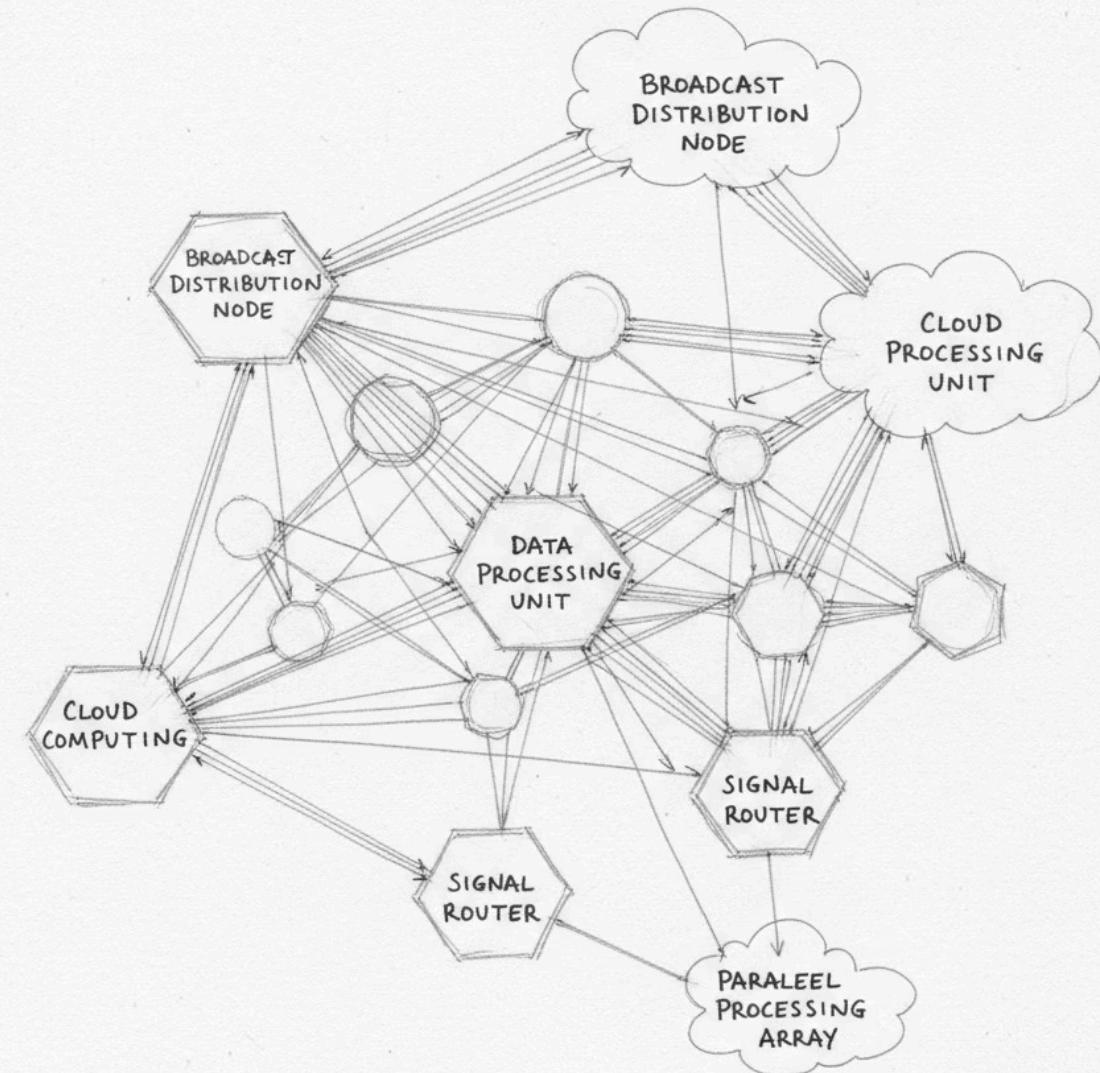
```
-- Wymuszone Broadcast Join
from pyspark.sql.functions import broadcast

result = large_df.join(
    broadcast(small_df),
    "product_id"
)

result.explain()
# => BroadcastHashJoin (BEZ SHUFFLE!)
```

## Kiedy używać

- Łączenie tabel faktów (dużej) z tabelą wymiarów (małej)
- Wymiar <10GB (domyślny próg)
- Wysokie koszty tasowania (SHUFFLE) w planie wykonania



# DEMO

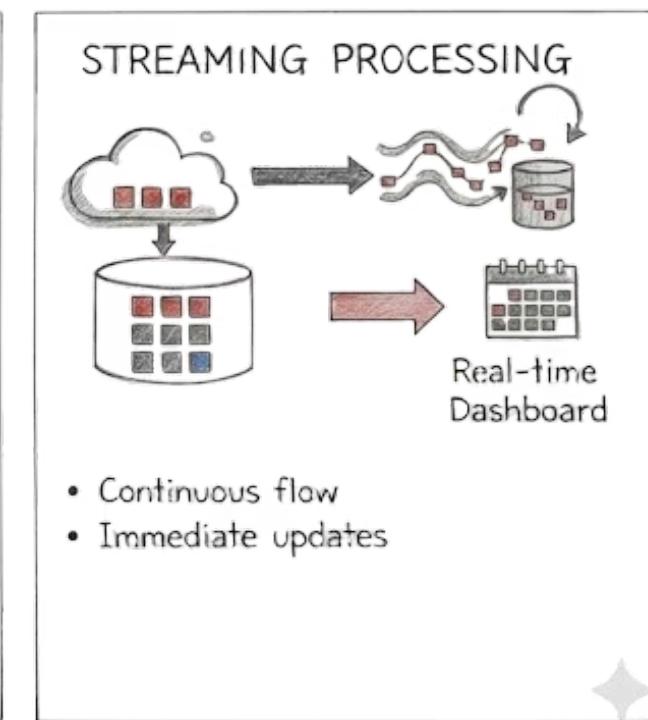
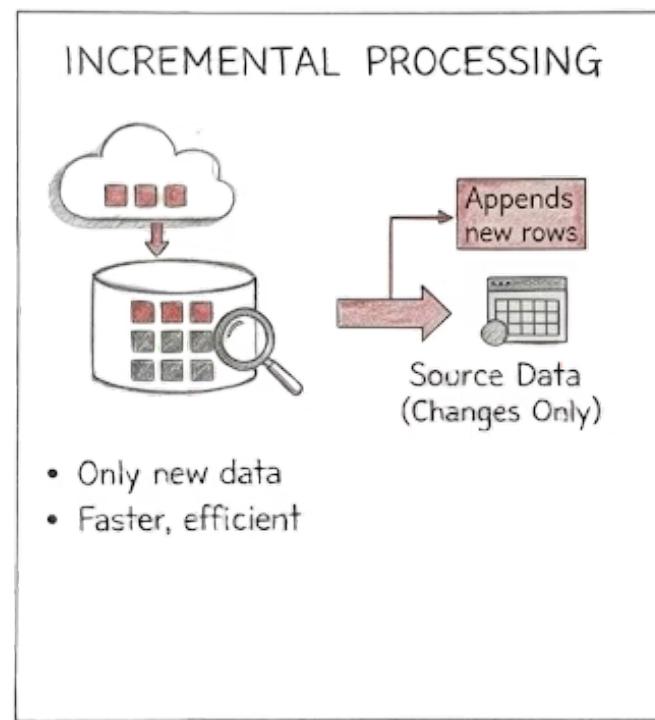
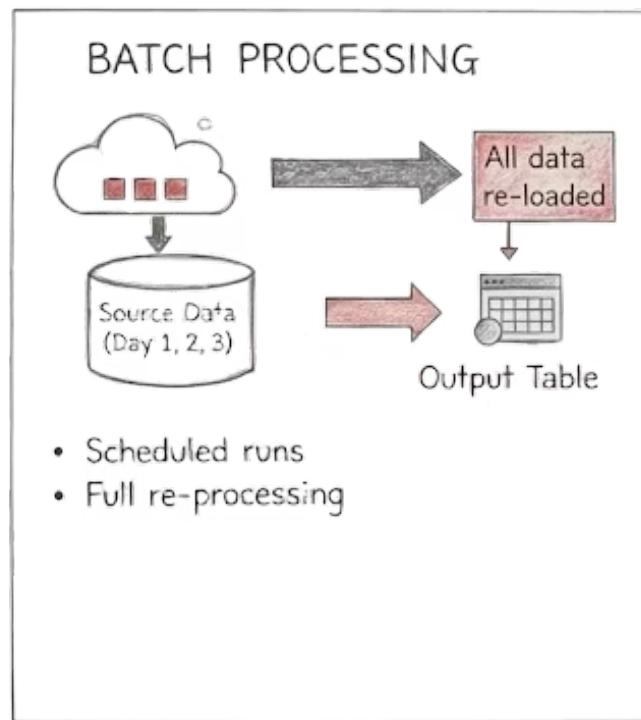
Zobaczmy Databricks w akcji!

# Ingestia Danych

Ladowanie danych do Lakehouse

# Wzorce Przetwarzania Danych

## DATA PROCESSING PATTERNS



# Porównanie Batch vs Streaming

Cecha	Batch (COPY INTO)	Streaming (Auto Loader)
Opóźnienie	Minuty – Godziny	Sekundy – Minuty
Rozmiar pliku	Duże pliki >1GB	Małe pliki <100MB
Częstotliwość	Planowane (godzinowo/dziennie)	Ciągłe / Bliskie czasu rzeczywistego
Koszt obliczeń	Niższy (na żądanie)	Wyższy (zawsze aktywne klastry)
Kluczowa funkcja	Idempotencja, Śledzenie plików	Ewolucja schematu, Punkty kontrolne
Przypadek użycia	Dzienne eksporty, Historyczne ładowanie	IoT sensors, Agregacja logów, CDC
Zarządzanie stanem	Proste (śledzenie plików)	Złożone (zarządzanie punktami kontrolnymi)

# Metody Ładowania Danych w Databricks

Databricks oferuje trzy główne metody ładowania danych do środowiska Delta Lake, każda z nich zoptymalizowana pod kątem różnych scenariuszy:

## CTAS (CREATE TABLE AS SELECT)

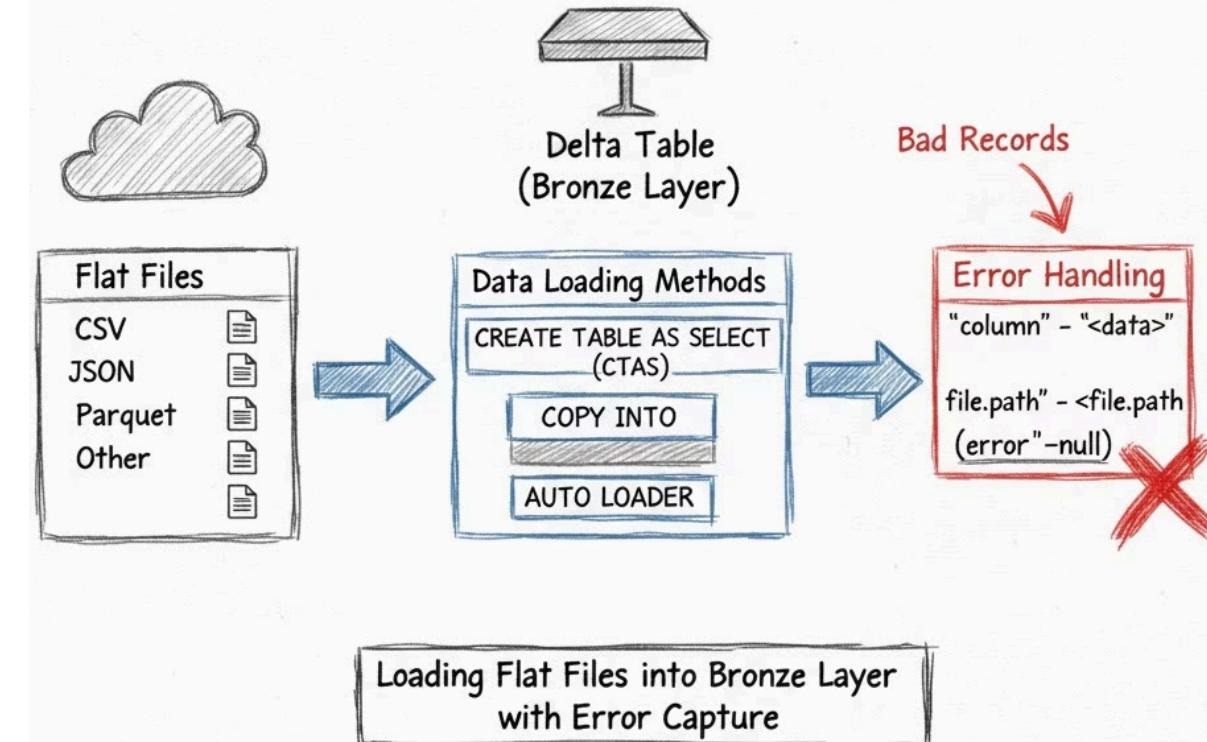
Ta metoda tworzy nową tabelę Delta Lake na podstawie wyników zapytania SQL. Jest idealna do jednorazowego tworzenia tabel, replikacji danych lub transformacji istniejących danych, gdzie cały zestaw danych jest już dostępny.

## COPY INTO

COPY INTO to efektywne narzędzie do ładowania wsadowego plików z zewnętrznych lokalizacji (np. S3, ADLS) do tabel Delta Lake. Obsługuje idempotencję, co zapobiega duplikacji danych, i jest przeznaczone do regularnego ładowania dużych partii danych z plików.

## Auto Loader

Auto Loader to mechanizm strumieniowego pobierania danych, który automatycznie i przyrostowo przetwarza nowe pliki danych, gdy tylko pojawią się w źródle. Jest to optymalne rozwiązanie do ciągłego przetwarzania danych, scenariuszy strumieniowych i zapewniania aktualności danych w czasie rzeczywistym.



# CTAS – CREATE TABLE AS SELECT

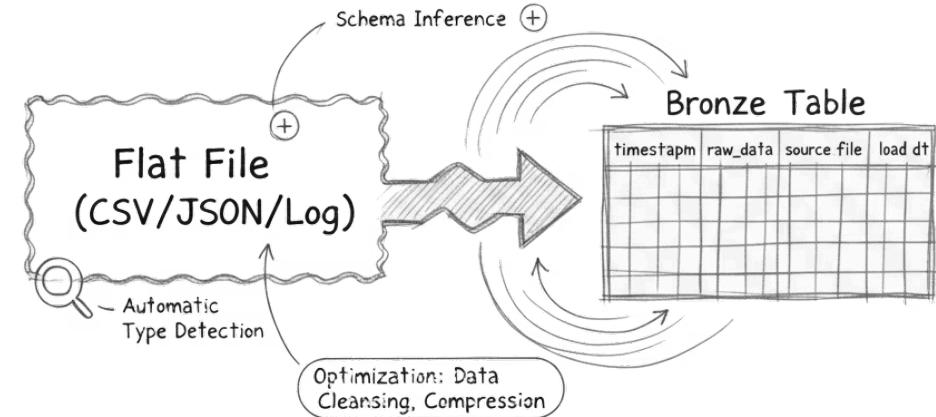
```
-- CTAS z automatycznym wnioskowaniem schematu
CREATE OR REPLACE TABLE raw_products
AS
SELECT * FROM read_files(
  '/Volumes/landing/products.csv',
  format => 'csv',
  header => true,
  inferSchema => true
);

-- - Struktura tabeli
DESCRIBE TABLE raw_products;
```

CTAS (CREATE TABLE AS SELECT) to wszechstronna metoda szybkiego tworzenia nowych tabel, idealna do początkowego ładowania danych ze źródeł zewnętrznych, takich jak pliki CSV. Umożliwia automatyczne wykrywanie schematu pliku źródłowego (**schema inference**) oraz stosowanie podstawowych transformacji danych (np. zmiana nazwy kolumn, konwersja typów) bezpośrednio podczas tworzenia tabeli.

## Zalety Schema Inference w CTAS:

- Automatyczne wykrywanie typów danych
- Brak konieczności wstępnego definiowania schematu
- Szybkie prototypowanie
- Wsparcie dla różnych formatów (CSV, JSON, Parquet)



**CREATE TABLE bronze\_table AS  
SELECT \* FROM flat-file\_source WHERE...**

# COPY INTO – Ładowanie Wsadowe

```
COPY INTO bronze_sales  
FROM 's3://bucket/sales/year=2024/'  
FILEFORMAT = PARQUET  
FORMAT_OPTIONS ('mergeSchema' = 'true')  
COPY_OPTIONS ('mergeSchema' = 'true');  
  
-- Automatyczne śledzenie plików  
SELECT * FROM bronze_sales;
```

Polecenie COPY INTO to najprostszy i najskuteczniejszy sposób ładowania danych wsadowych do Delta Lake. Kluczową cechą jest **idempotentność** – uruchomienie go wielokrotnie na tym samym zestawie plików nie spowoduje duplikatów.

## Kiedy używać COPY INTO

- Pliki >1GB (optymalny rozmiar dla Parquet)
- Harmonogramowane co kilka godzin lub rzadziej
- Początkowe ładowanie dużych ilości danych historycznych
- Eksporty z systemów zewnętrznych (codzienne zrzuty)
- Niskie wymagania dotyczące opóźnień (SLA >15 min)



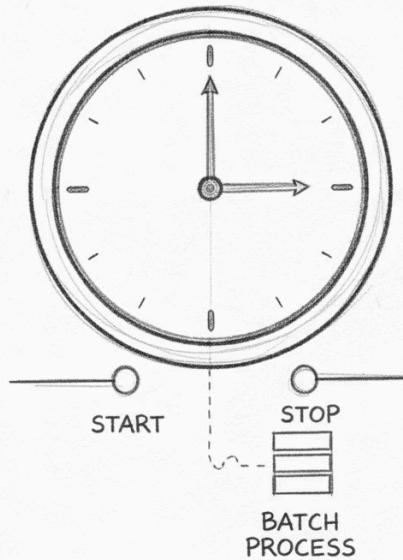
COPY INTO śledzi załadowane pliki w metastore – plik załadowany raz nie zostanie załadowany ponownie.

# Tryby Wyzwalania – Kontrola Opóźnień i Kosztów

Tryb wyzwalania określa częstotliwość wykonywania mikro-paczek w potoku strumieniowym. Jest to kluczowa decyzja, która wpływa na kompromis między opóźnieniami a kosztami obliczeniowymi.

Tryb Wyzwalania	Opis	Kiedy Używać	Koszt
availableNow=True	Przetwarza wszystkie dostępne dane i zatrzymuje się	Zaplanowane zadania, optymalizacja kosztów, strumieniowanie przypominające wsadowe	Niski
processingTime="10second"	Mikro-paczka co 10 sekund (ciągłe)	Pulpity nawigacyjne w czasie rzeczywistym, analityka operacyjna	Wysoki
once=True	Pojedyncza mikro-paczka i zakończenie	Testowanie, jednorazowe przetwarzanie	Bardzo Niski
Domyślny (brak wyzwalacza)	Ciągłe przetwarzanie ASAP	Ultra-niskie opóźnienia (<1s), wykrywanie oszustw	Bardzo Wysoki

# availableNow – Rekomendowany Tryb Wyzwalania



```
(df.writeStream  
    .format("delta")  
    .trigger(availableNow=True)  
    .option("checkpointLocation", checkpoint_path)  
    .table("bronze_table"))
```

`availableNow=True` to rozwiązanie łączące najlepsze cechy obu światów: semantykę strumieniową (dokładnie raz, zarządzanie checkpoint management, schema evolution) z ekonomicznością przetwarzania wsadowego.

## Zalety availableNow

- Klaster uruchamia się tylko, gdy dostępne są nowe dane
- Pełna kontrola nad harmonogramowaniem za pomocą Workflows
- Identyczna logika dla przetwarzania wsadowego i strumieniowego
- Redukcja kosztów obliczeniowych o 70–90% w porównaniu do trybu ciągłego

# processingTime – Wyzwalacz Czasu Rzeczywistego

```
(df.writeStream  
    .format("delta")  
    .trigger(processingTime="10 seconds")  
    .option("checkpointLocation", checkpoint_path)  
    .table("bronze_table"))  
  
# Klaster pozostaje aktywny i uruchamia mikropaczę co 10 sekund
```

Wyzwalacz processingTime uruchamia mikropaczę w stałych odstępach czasu, niezależnie od dostępności danych. Klaster musi pozostać aktywny non-stop, co generuje wysokie koszty obliczeniowe.

## ✓ Kiedy używać

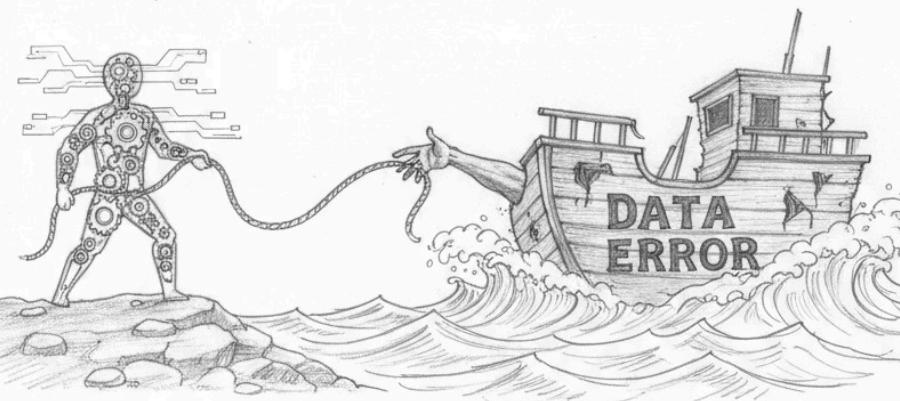
- Operacyjne pulpty nawigacyjne w czasie rzeczywistym
- SLA dla opóźnienia < 1 minuty
- Wykrywanie oszustw / wykrywanie anomalii
- Monitorowanie systemów IoT na żywo

## ⚠ Ograniczenia

- Klaster zawsze aktywny = wysokie koszty DBU
- Pusta mikropaczka, jeśli brak danych
- Trudniejsze zarządzanie cyklem życia
- Wymaga dedykowanych pul klastrów

# Kolumna `\_rescued\_data` – Obsługa Błędów

Auto Loader automatycznie dodaje kolumnę `\_rescued\_data` do każdej tabeli Bronze. Zawiera ona surowy JSON z rekordów, które nie pasują do oczekiwanej schematu – zamiast odrzucać błędne dane, Delta Lake zachowuje je do późniejszej analizy.



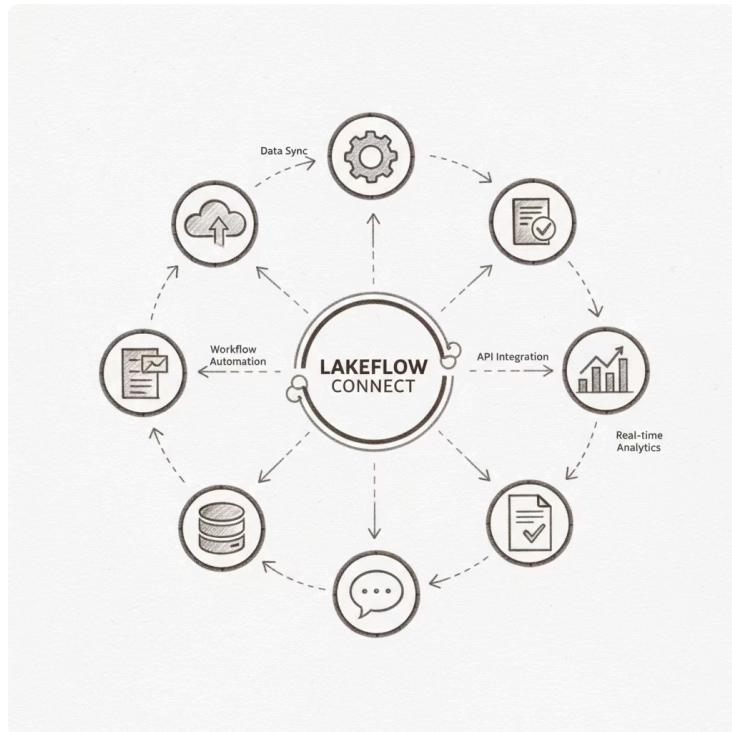
RECOVERY  
PROCESS

```
-- Sprawdź problematyczne rekordy
SELECT _rescued_data, count(*) as error_count
FROM bronze_events
WHERE _rescued_data IS NOT NULL
GROUP BY _rescued_data
ORDER BY error_count DESC;
```

```
-- Typowy rekord rescued
>{"unexpected_field": "value", "malformed_json": "..."}
```

Jest to kluczowy mechanizm jakości danych – zamiast cichej awarii, zyskujemy wgląd w problemy źródła danych oraz możliwość ich retrospektywnego naprawienia.

# Lakeflow Connect – Integracja SaaS



Konektor	Przypadek Użycia
Salesforce	Dane CRM, Szanse sprzedażowe, Konta
SAP	Dane ERP, Transakcje finansowe
Workday	Dane HR, Rejestry pracowników
ServiceNow	Zarządzanie Usługami IT, Incydenty
Google Analytics	Analityka internetowa, Zachowania użytkowników
HubSpot	Automatyzacja marketingu, Leady

Lakeflow Connect to rozwiązanie zero-code do integracji z popularnymi systemami SaaS. Automatyczna konfiguracja Auth, Schema Discovery, Incremental Load i Error Handling.

Lakeflow Connect eliminuje potrzebę budowania niestandardowych Konektorów i zarządzania limitami API, paginacją, odświeżaniem uwierzytelniania itp. Wszystko jest zarządzane przez Databricks.

# DEMO

Zobaczmy Databricks w akcji!

# Zaawansowane Transformacje

Zaawansowane Transformacje

PySpark & SQL

# Dlaczego Zaawansowane Transformacje?

## Proste SELECT to za mało

Współczesne problemy analityczne wymagają znacznie więcej niż podstawowych zapytań SQL. Potrzebujemy narzędzi do analizy trendów, rankingów, danych hierarchicznych i szeregów czasowych.

- Analiza konkurencyjności i benchmarking
- Wykrywanie anomalii w danych
- Agregacje szeregów czasowych

## Złożoność danych rośnie

Współczesne źródła danych dostarczają JSON, arrays i zagnieżdżone struktury. Musimy być w stanie przetwarzać je efektywnie, nie tracąc na wydajności.

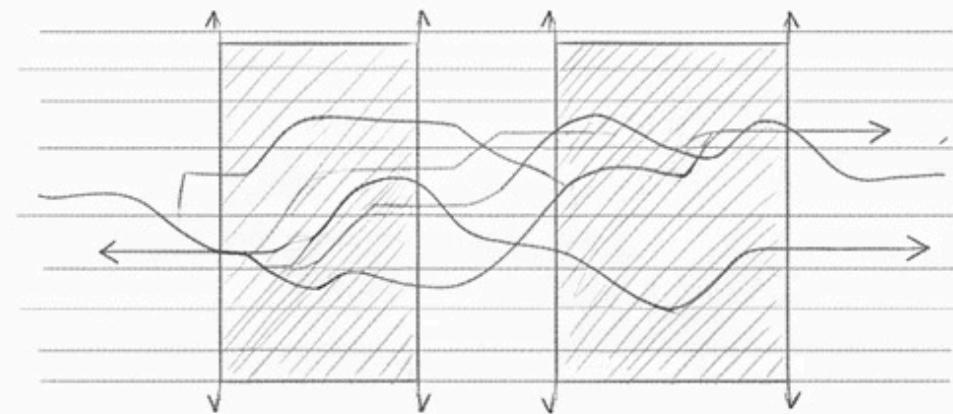
- APIs zwracające dane półstrukturalne
- Strumienie zdarzeń z metadanymi
- Logi aplikacji w formacie JSON

# Window Functions – Analiza w oknach

Window functions pozwalają na wykonywanie obliczeń w kontekście „okna” wierszy, bez grupowania całego zbioru danych. Jest to potężne narzędzie do rankingów, porównań międzyokresowych i analizy trendów.

## DATA ANALYTICS DASHBOARD

### WINDOW FUNCTIONS



RANK()

43.16
43.16

ROW-NUMBER

= 1.2.60
= 1.2.60

SUM) OVER...

~ 4.5.30
~ 4.5.30

# Kluczowe funkcje okna (Window Functions)

## **rank() / dense\_rank()**

Tworzy rankingi w obrębie grup. rank() pozostawia luki po równych wartościach, dense\_rank() numeruje w sposób ciągły.

```
rank() OVER (PARTITION BY category ORDER BY sales DESC)
```

## **row\_number()**

Unikalny numer wiersza w oknie, przydatny do wybierania top N rekordów z każdej grupy lub do usuwania duplikatów (deduplication).

```
row_number() OVER (PARTITION BY user_id ORDER BY timestamp DESC)
```

## **lag() / lead()**

Dostęp do wartości z poprzedniego lub następnego wiersza. Idealne do obliczeń różnic (delta calculations), zmian procentowych i porównań okres do okresu (period-over-period).

```
lag(revenue, 1) OVER (ORDER BY date)
```

## **partitionBy**

Klauzula definiująca logiczne grupy, w których obliczenia są wykonywane niezależnie. Kluczowa dla analizy wielowymiarowej (multi-dimensional analysis).

```
PARTITION BY region, product_category
```

# Przykład: Ranking produktów

## Przypadek użycia

Znajdź 3 najlepsze produkty w każdej kategorii na podstawie sprzedaży w ostatnim kwartale.

## PySpark Implementation

```
from pyspark.sql import Window
from pyspark.sql.functions import rank

window_spec = Window.partitionBy("category") \
    .orderBy(col("sales").desc())

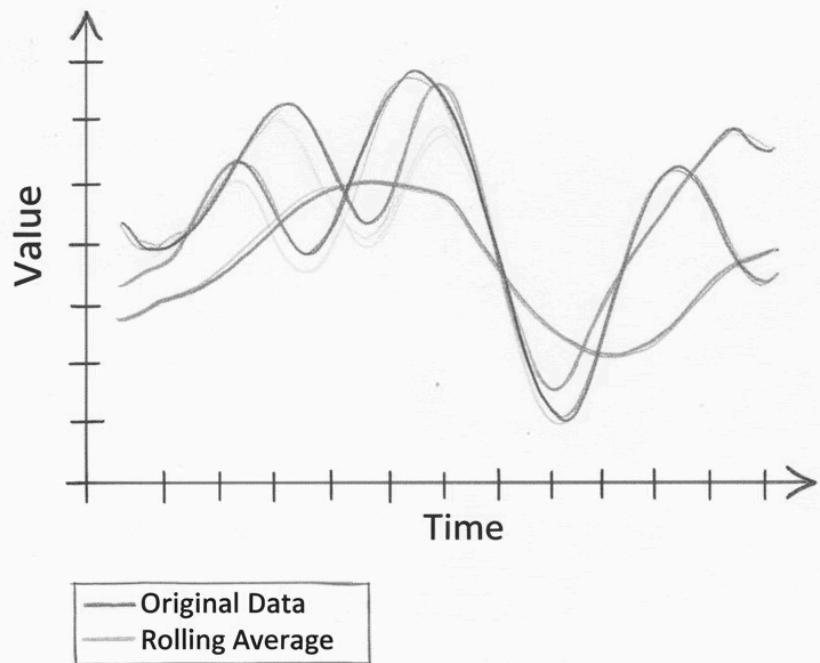
df_ranked = df.withColumn(
    "rank", rank().over(window_spec)
).filter(col("rank") <= 3)
```

## SQL Implementation

```
WITH ranked_products AS (
    SELECT
        product_name,
        category,
        sales,
        RANK() OVER (
            PARTITION BY category
            ORDER BY sales DESC
        ) as product_rank
    FROM sales_data
    WHERE quarter = 'Q4'
)
SELECT *
FROM ranked_products
WHERE product_rank <= 3
```

**Notatnik:** Zobacz pełną implementację w Notebook 01a, sekcji "Window Functions Basics"

## Time Series Rolling Average



## Rolling Windows – Moving Analysis

Rolling windows umożliwiają wykonywanie obliczeń w ruchomym oknie czasowym – co jest kluczowe dla data smoothing, trend detection i predictive analysis. Są szczególnie przydatne w time series, gdzie chcemy reduce noise i dostrzec long-term patterns.

# rowsBetween – Kontrolowanie Zakresu Okna

## Składnia i Parametry

Metoda `rowsBetween(start, end)` definiuje fizyczny zakres wierszy w oknie:

- **Wartości ujemne:** wiersze poprzedzające bieżący (-2 = dwa wiersze wstecz)
- **0:** bieżący wiersz
- **Wartości dodatnie:** wiersze następujące po bieżącym
- **Window.unboundedPreceding:** od początku partycji
- **Window.unboundedFollowing:** do końca partycji

## Typowe Scenariusze Użycia

### Średnia z 3 dni:

```
rowsBetween(-2, 0)
```

### Średnia centralna z 5 dni:

```
rowsBetween(-2, 2)
```

### Suma skumulowana:

```
rowsBetween(Window.unboundedPreceding, 0)
```

## Przykład: 7-dniowa średnia krocząca

```
from pyspark.sql import Window
from pyspark.sql.functions import avg, col

# Define the window: 7 days back (including the current day)
window_7d = Window.partitionBy("product_id") \
    .orderBy("date") \
    .rowsBetween(-6, 0)

# Calculate the moving average of sales
df_with_ma = df.withColumn(
    "sales_7d_avg",
    avg("daily_sales").over(window_7d)
)

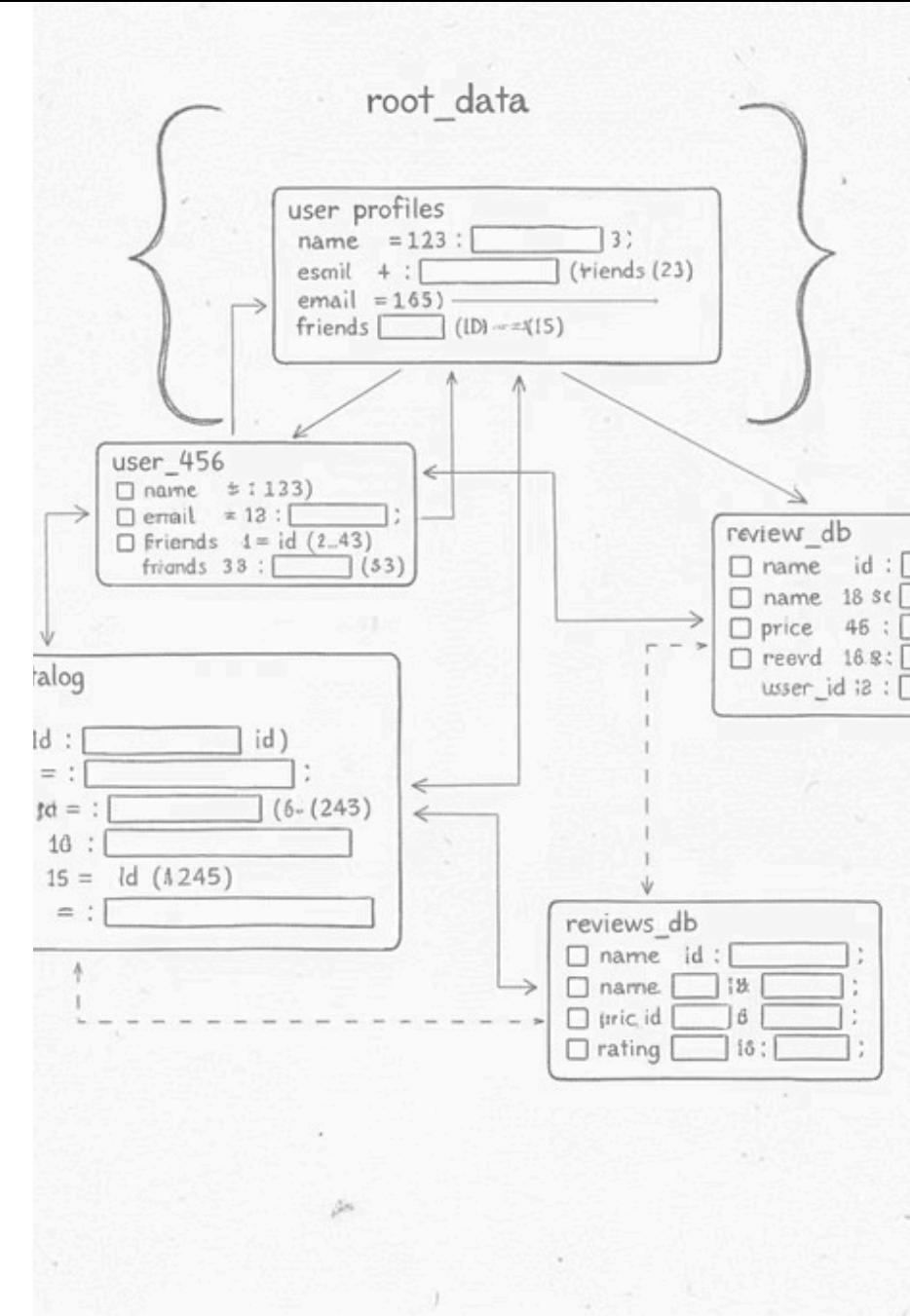
# In SQL, this can be written as:
# AVG(daily_sales) OVER (
#   PARTITION BY product_id
#   ORDER BY date
#   ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
# ) as sales_7d_avg
```

**Praktyczna uwaga:** Uporządkowanie (orderBy) jest kluczowe – bez niego wyniki będą nieprzewidywalne. Dla danych szeregów czasowych, zawsze sortuj według sygnatury czasowej/daty.

**Notebook:** Notebook 01a, sekcja "Rolling Windows for Time Series"

# Złożone Dane - Semi-structured Data

Nowoczesne aplikacje generują dane w formatach takich jak JSON, XML, Avro – zagnieżdżone struktury, tablice, mapy. PySpark udostępnia dedykowane funkcje do rozpakowywania i normalizowania takich danych, umożliwiając skuteczną analizę bez zewnętrznego przetwarzania.



# Kluczowe Funkcje dla Złożonych Danych

## **explode()**

Rozwija tablicę na wiele wierszy – każdy element tablicy staje się osobnym wierszem. Pozostałe kolumny są duplikowane.

```
df.select("order_id", explode("items").alias("item"))
```

## **from\_json()**

Analizuje ciąg znaków JSON do struktury StructType zgodnie z podanym schematem. Umożliwia dostęp do zagnieżdżonych pól za pomocą notacji kropkowej.

```
from_json(col("json_string"), schema)
```

## **get\_json\_object()**

Wyodrębnia pojedyncze pole z JSON bez definiowania pełnego schematu. Szybsze w prostych przypadkach, ale zwraca ciąg znaków.

```
get_json_object(col("json_col"), "$.user.name")
```

## **struct() i map()**

Tworzenie niestandardowych zagnieżdżonych struktur – przydatne podczas przygotowywania danych do zapisu w formacie Delta lub Parquet z zagnieżdżonymi kolumnami.

```
struct(col("id"), col("name")).alias("user_info")
```

# Przykład: Parsowanie JSON z Logów

## Dane Wejściowe

Tabela zawiera kolumnę event\_json z logami aplikacji:

```
{  
  "user_id": "u_12345",  
  "event_type": "purchase",  
  "items": [  
    {"product_id": "p_001", "qty": 2},  
    {"product_id": "p_042", "qty": 1}  
  ],  
  "metadata": {  
    "timestamp": "2024-01-15T10:30:00Z",  
    "source": "mobile_app"  
  }  
}
```

## Kod PySpark

```
from pyspark.sql.types import *  
  
# Define schema  
schema = StructType([  
  StructField("user_id", StringType()),  
  StructField("event_type", StringType()),  
  StructField("items", ArrayType(StructType([  
    StructField("product_id", StringType()),  
    StructField("qty", IntegerType())  
  ]))),  
  StructField("metadata", StructType([  
    StructField("timestamp", StringType()),  
    StructField("source", StringType())  
  ]))  
])  
  
# Parse and explode  
df_parsed = df.withColumn(  
  "event", from_json("event_json", schema))  

```

**Notebook:** Notebook 01b, sekcja "Working with JSON and Arrays"

# DEMO

Zobaczmy Databricks w akcji!

# Medallion Architecture

Warstwowa architektura danych

# Medallion Architecture

Medallion Architecture to warstwowe podejście do organizacji danych w data lake, zapewniające ustrukturyzowane i stopniowe czyszczenie i transformację danych.

Jest kluczowe dla utrzymania jakości danych, zarządzania nimi i dostępności w złożonych środowiskach analitycznych.

## Bronze

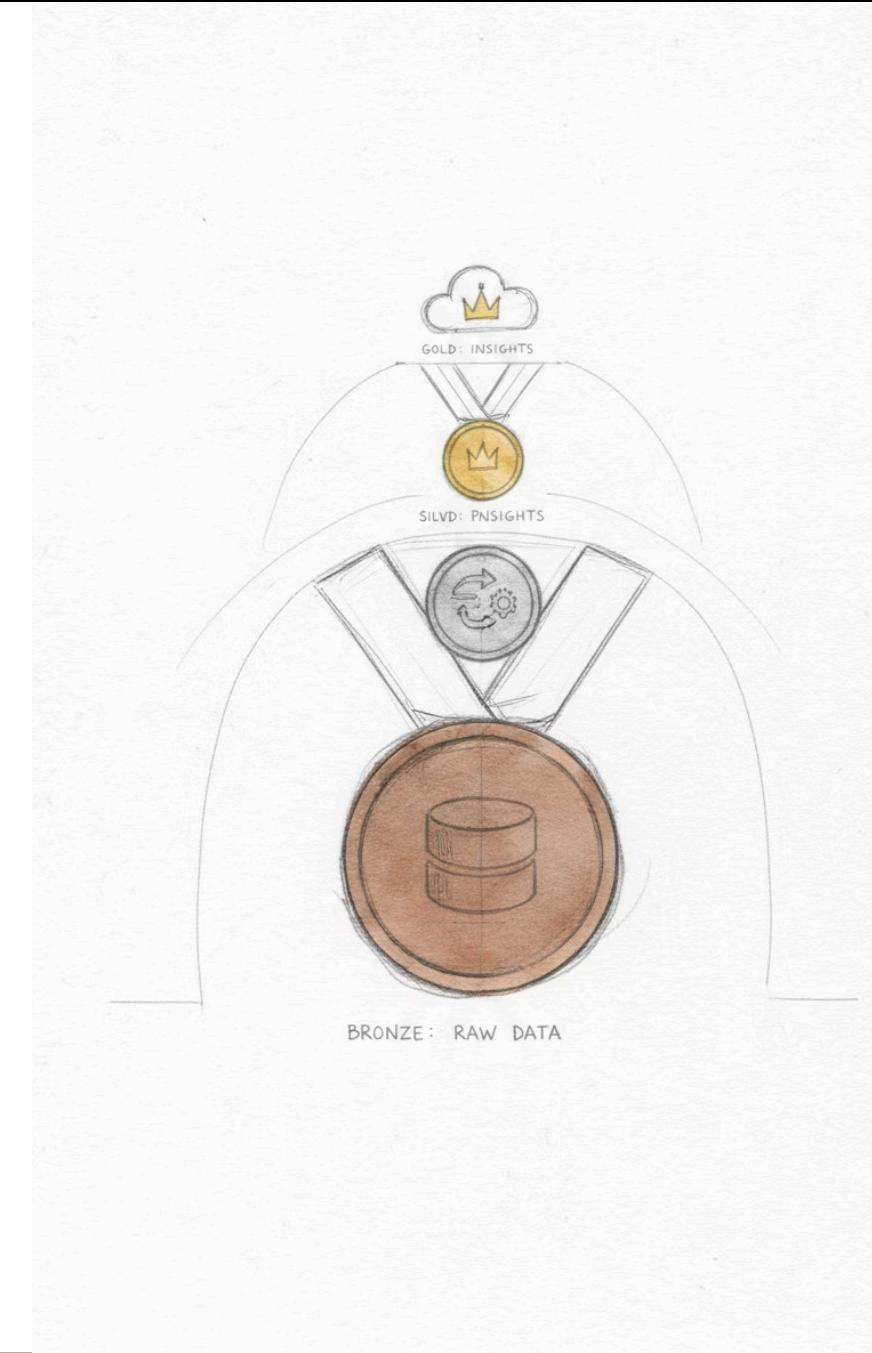
Warstwa surowych danych, niezmieniona od źródła.

## Silver

Warstwa danych oczyszczonych i zweryfikowanych.

## Gold

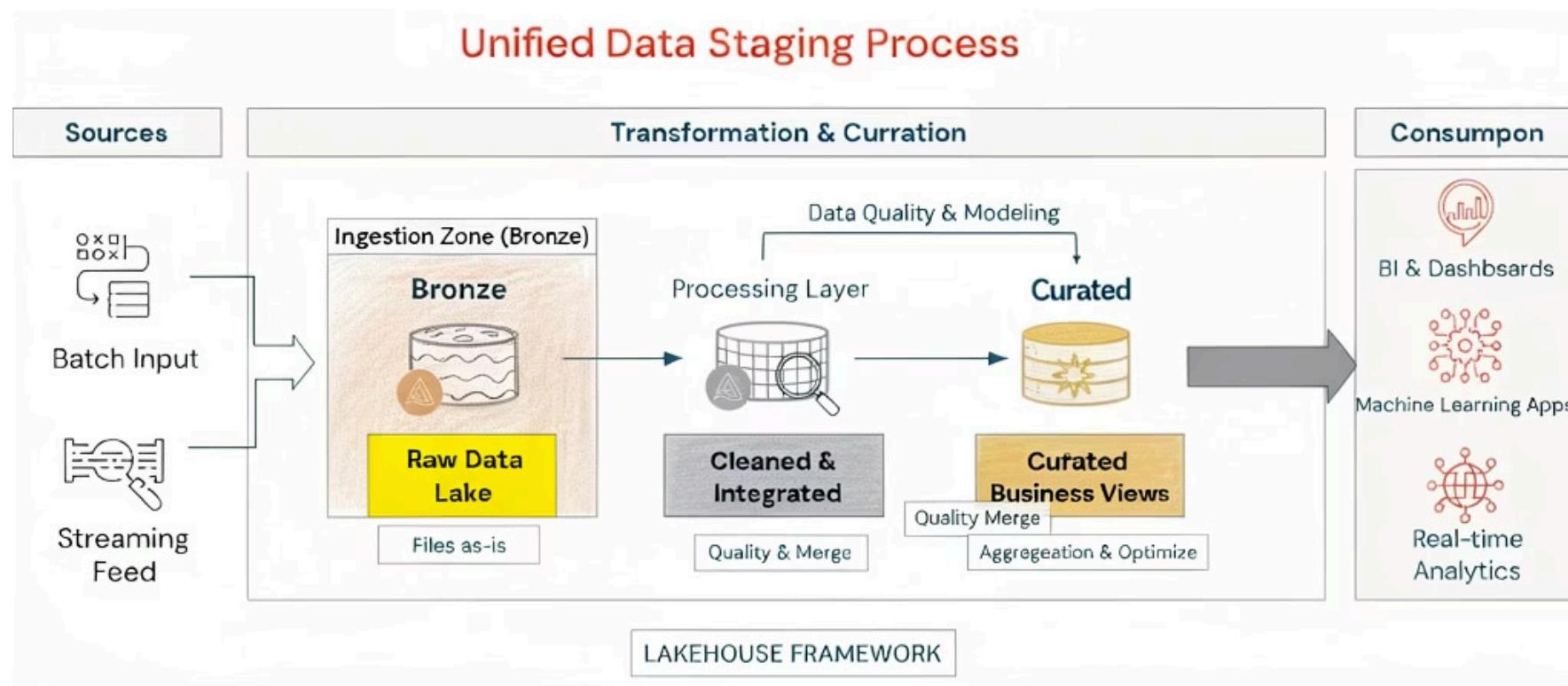
Warstwa danych gotowych do analizy biznesowej. Dane są zagregowane i w łatwym do wykorzystania formacie.

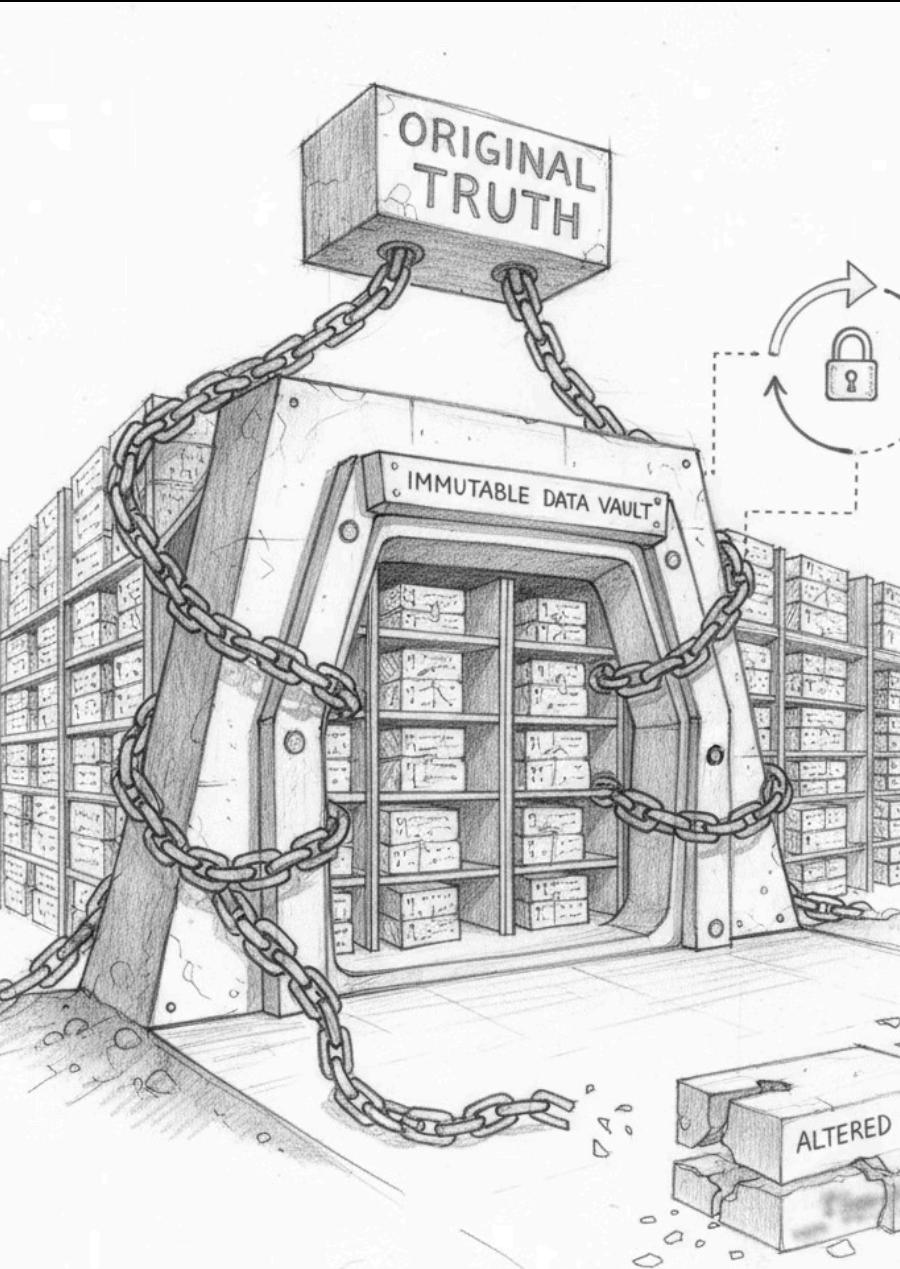


# Medallion Architecture

Medallion Architecture to standardowy wzorzec organizacji danych w Lakehouse, który stopniowo oczyszcza i wzbogaca surowe dane do postaci gotowej do konsumpcji przez użytkowników biznesowych i ML models.

Każda warstwa ma jasno określone obowiązki i data quality SLAs. Separacja warstw umożliwia izolację problemów, przetwarzanie przyrostowe oraz optymalizację storage/compute costs na każdym etapie.





# Warstwa Bronze – Zachowanie Surowych Danych

Warstwa Bronze to niezmienny archiwum surowych danych z systemów źródłowych. Filozofia: **zachowaj wszystko**, nie transformuj, dodaj metadane. Bronze jest fundamentem dla audytowania, odzyskiwania po awarii i ponownego przetwarzania.

## Kluczowe praktyki

- Wzorzec Append-only (nigdy DELETE/UPDATE)
- Zachowanie oryginalnych typów i formatów
- Dodanie technical columns (load\_timestamp, source\_file)
- Wykorzystanie *rescued* data dla błędnych rekordów
- Retention policy >= 90 dni (compliance)

```
ALTER TABLE bronze_events
SET TBLPROPERTIES (
    delta.logRetentionDuration = '180 days',
    delta.deletedFileRetentionDuration = '90 days'
);
```

### ❑ Wzbogacanie metadanymi

Każdy rekord w Warstwie Bronze powinien zawierać:

- `_load_timestamp` – czas załadowania
- `_source_file` – ścieżka pliku źródłowego
- `_source_system` – identyfikator systemu
- `_ingestion_id` – UUID batch/stream job

# Warstwa Silver – Zweryfikowana i Wzbogacona

## Oczyszczanie

Usuwanie wartości null, normalizacja formatu, usuwanie białych znaków (trim whitespace), standaryzacja kodowania (encoding standardization)

## Deduplikacja

Usuwanie duplikatów za pomocą MERGE INTO lub funkcji okienkowych (window functions, ROW\_NUMBER)

## Walidacja

Reguły biznesowe (Business rules), sprawdzanie integralności referencyjnej (referential integrity), walidacja zakresu (range validation), flagi jakości (quality flags)

## Wzbogacanie

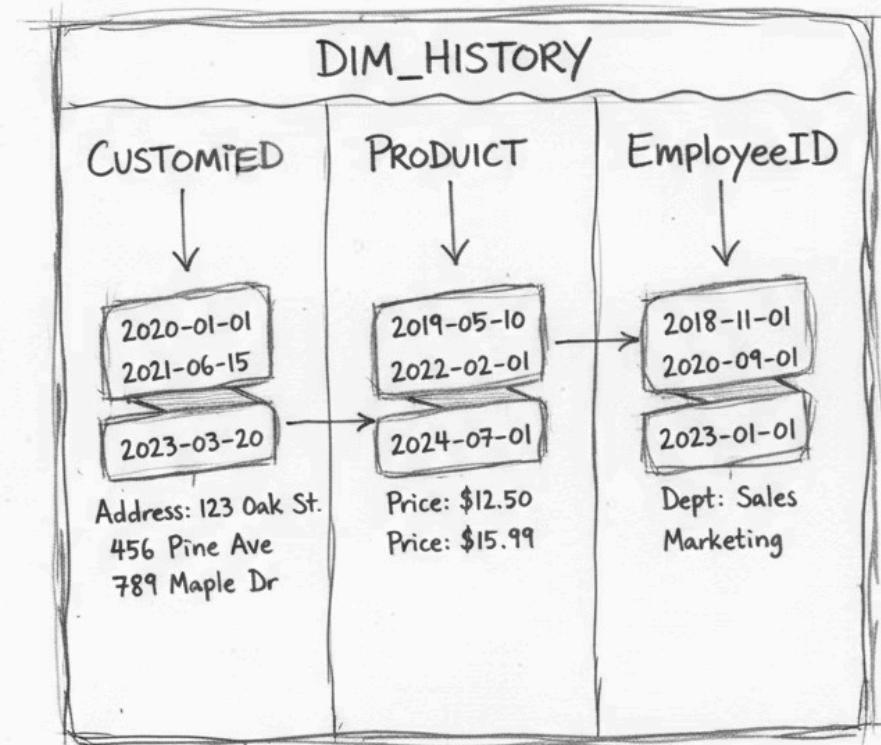
Łączenie z tabelami wymiarów (dimension tables), pola obliczeniowe (calculated fields), przeliczanie walut, geokodowanie (geocoding)

Warstwa Silver jest \*\*jedynym źródłem prawdy\*\* (single source of truth) dla dalszych konsumentów (downstream consumers). Dane w Silver gwarantują wysoką jakość i spójność – jeśli rekord istnieje w Silver, jest on poprawny biznesowo.

# Slowly Changing Dimensions

Slowly Changing Dimensions (SCD) to strategia zarządzania zmianami w tabelach wymiarów Data Warehouse. Adresy klientów zmieniają się, produkty zmieniają ceny, pracownicy przenoszą się do innych działów – jak zachować historię tych zmian?

Różne typy SCD reprezentują różne kompromisy między kosztem przechowywania, złożonością zapytań i wymaganiami biznesowymi dotyczącymi historii. Najpopularniejsze w praktyce to Type 1 (nadpisywanie) i Type 2 (pełna historia).



# SCD Type 1 – Nadpisz

## Strategia: Nadpisz starą wartość

SCD Type 1 to najprostsze podejście – gdy atrybut się zmienia, nadpisujemy starą wartość nową. **Historia zostaje utracona**. Używane dla atrybutów, które nie wymagają audytu (np. korekta błędów, nieistotne zmiany).

```
MERGE INTO dim_customer AS target
USING customer_updates AS source
ON target.customer_id = source.customer_id
WHEN MATCHED THEN
    UPDATE SET
        target.address = source.address,
        target.phone = source.phone,
        target.updated_at = current_timestamp()
WHEN NOT MATCHED THEN
    INSERT *;
```

## Przykład transformacji

```
-- State before update
customer_id | address      | phone
1001       | Warsaw       | 123-456
```

```
-- New data
customer_id | address      | phone
1001       | Krakow       | 789-012
```

```
-- State after MERGE (Type 1)
customer_id | address      | phone
1001       | Krakow       | 789-012
```

- ✓ **Zalety:** Prostota, niskie koszty przechowywania
- ✗ **Wady:** Brak historii, brak audytu

## SCD Type 2 – Pełna historia

SCD Type 2 zachowuje pełną historię zmian poprzez tworzenie nowego wiersza dla każdej wersji rekordu. Kluczowe kolumny: `is_current`, `effective_start_date`, `effective_end_date`. Umożliwia wykonywanie point-in-time queries.

```
MERGE INTO dim_product_scd2 AS target
USING (
    SELECT *, current_date() as effective_date FROM product_updates
) AS source
ON target.product_id = source.product_id
AND target.is_current = true

-- Close out current record
WHEN MATCHED AND (
    target.price != source.price OR
    target.category != source.category
) THEN UPDATE SET
    target.is_current = false,
    target.effective_end_date = source.effective_date

-- Insert new version
WHEN NOT MATCHED THEN INSERT (
    product_id, name, price, category,
    is_current, effective_start_date, effective_end_date
) VALUES (
    source.product_id, source.name, source.price, source.category,
    true, source.effective_date, '9999-12-31'
);

-- Insert new current version for updated records
INSERT INTO dim_product_scd2
SELECT * FROM source WHERE exists_in_target_as_outdated;
```

## SLOWLY CHANGING DIMENSION (SCD) TYPE 2

CustMoried	Firstname	Address	Straate	Ende	IsCurrent
101	John	John	2020-01-01	789 Elm Blvd	0
101	John	456 Pine Ave	2022-03-15	9999-12-31	1
101	Doe		2021-06-20	9999-12-31	1
102				1	1
102					1

### 1. UPDATE OLD RECORD:

```
UPDATE Customers SET IsCurrent = 0 WHERE CustomerID = '123' AND IsCustomerID = 1;
```

### 2. SET ENDDATE: SET EndDate = CURRENT\_DATE

```
WHERE Customer = CURRENT_123 AND Enddate = '9999-31';
```

### 3. INSERT NEW RECORD: (CustomerID, (IsCurime, Lastrame, Address,

```
Address, StartSate, EnartSate, Endats = LINES '789 Maple Ave' '789 Maple at) - 1;
VALUES "John, Doe, 'CURENT_939-31', 1;
```

```
)
```

# SCD Type 2 – Przykład Historii

product_id	name	price	is_current	effective_start	effective_end
P001	Laptop X	4500	false	2023-01-01	2023-06-30
P001	Laptop X	4200	false	2023-07-01	2024-01-15
P001	Laptop X Pro	4200	true	2024-01-16	9999-12-31

-- Zapytanie punktowe w czasie: jaka była cena w Q3 2023?

```
SELECT price  
FROM dim_product_scd2  
WHERE product_id = 'P001'  
AND '2023-09-15' BETWEEN effective_start_date AND effective_end_date;
```

-- Wynik: 4200

-- Aktualna wersja

```
SELECT * FROM dim_product_scd2 WHERE product_id = 'P001' AND is_current = true;
```

# SCD Type 3 & 4 – Alternatywne Strategie

## SCD Type 3 – Poprzednia Wartość

Przechowywanie tylko jednej poprzedniej wartości atrybutu w dedykowanej kolumnie (np. previous\_address). Ograniczona historia – tylko *bieżąca i poprzednia*.

<b>id</b>	<b>address</b>	<b>prev_address</b>
1001	Krakow	Warsaw

Zastosowanie: analiza "przed/po", śledzenie ostatnich zmian, niski narzut pamięci masowej.

Typy 3 i 4 są rzadziej używane w praktyce – większość przypadków dotyczy wyboru między Type 1 (proste nadpisanie) a Type 2 (pełna historia).

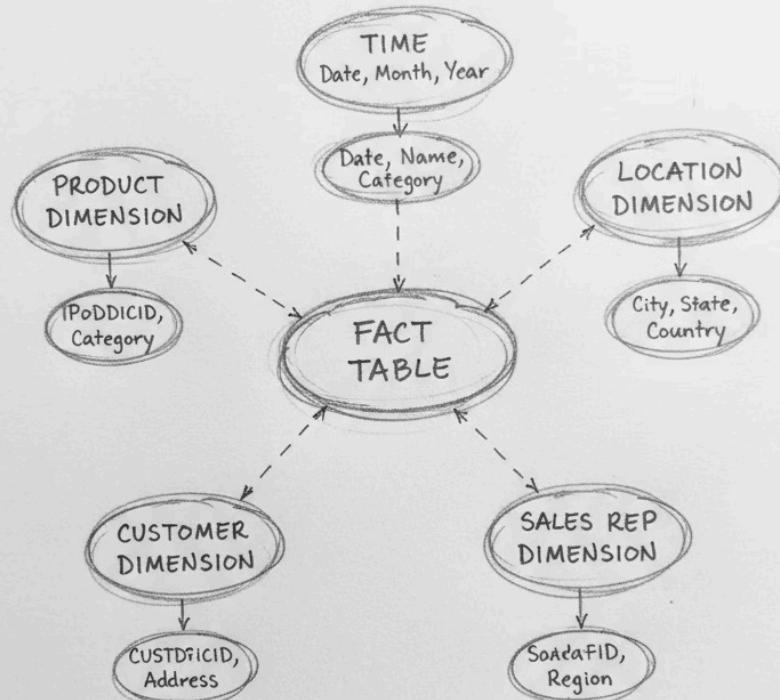
## SCD Type 4 – Historia w Oddzielnej Tabeli

Podział na dwie tabele: dim\_customer (stan bieżący) + dim\_customer\_history (pełny ślad audytowy). Separacja danych gorących od danych zimnych.

- dim\_customer: tylko bieżące rekordy
- dim\_customer\_history: wszystkie historyczne wersje

Zastosowanie: Optymalizacja wydajności zapytań dla bieżącego stanu, wymagania zgodności dla długoterminowego audytu.

# Warstwa Gold – Dane gotowe do użytku biznesowego



Warstwa Gold to **warstwa konsumpcji biznesowej** – dane są agregowane, zdenormalizowane i zoptymalizowane pod kątem konkretnych zastosowań (BI dashboards, ML features, API endpoints). Warstwa Gold implementuje klasyczne wzorce Data Warehouse: Star Schema, Fact & Dimension tables.

## Fact Tables

Transakcje, zdarzenia, pomiary. High cardinality, częste wstawienia.  
Zawierają foreign keys do wymiarów i metrics (amount, quantity).

## Dimension Tables

Jednostki biznesowe (klienci, produkty, lokalizacje). Low cardinality, rzadkie aktualizacje. Zawierają atrybuty opisowe do slice & dice.

## Aggregates

Wstępnie obliczone podsumowania (daily\_sales, monthly\_kpis). Drastyczne zmniejszenie kosztów skanowania dla typowych zapytań. Materialized views.

# Lakeflow Pipelines

Nowoczesne ETL

# Czym są Lakeflow Pipelines?

Lakeflow (wcześniej Delta Live Tables) to framework do **deklaratywnego budowania data pipelines** w Databricks. Zamiast pisać kod orkiestracji, definiujemy **końcowy rezultat** – framework zarządza wykonaniem, monitorowaniem i jakością danych.

## Tradycyjne Podejście

- Ręczne zarządzanie stanem
- Niestandardowa logika ponownych prób
- Niestandardowe punkty kontrolne
- Ręczne śledzenie zależności
- Oddzielne narzędzia kontroli jakości

## Lakeflow Pipelines

- Deklaratywne definicje tabel
- Automatyczne ponowne próby i obsługa błędów
- Wbudowane checkpointing
- Automatyczny wykres pochodzenia danych
- Wbudowane Expectations (zasady jakości)

# Co stało się z Delta Live Tables (DLT)?

Delta Live Tables (DLT) zostało zaktualizowane do **Lakeflow Spark Declarative Pipelines (SDP)**. Istniejący kod DLT jest w pełni kompatybilny, ale zaleca się aktualizację do nowych nazw API dla przyszłych funkcji i kompatybilności z Apache Spark™ Declarative Pipelines (od Spark 4.1).

Kluczowe zmiany w Python API:

`@dlt` → `@dp`

Zastępuje referencję do biblioteki.

`@table (general)` → `@table (streaming)` /  
`@materialized_view`

Wprowadzono dedykowany dekorator dla  
materialized views, zwiększając czytelność.

`@view` → `@temporary_view`

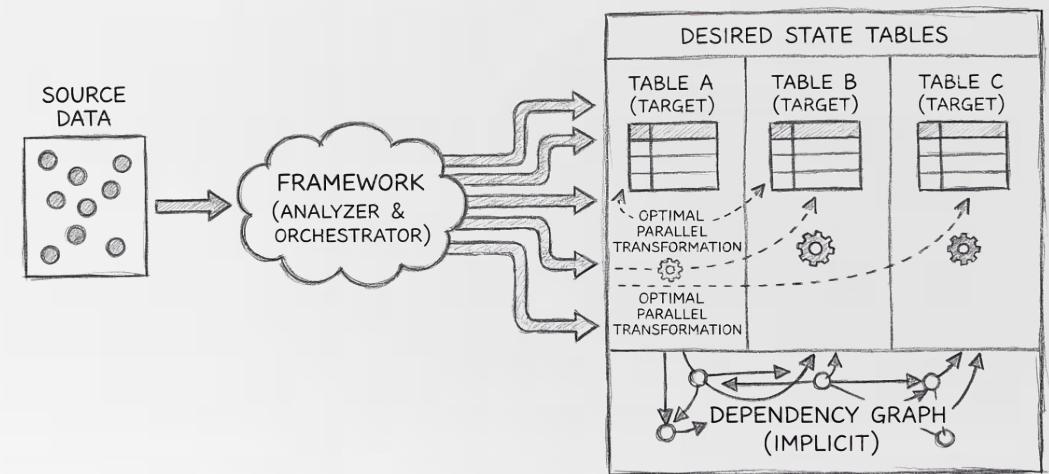
Wyraźne rozróżnienie dla temporary views.

- ❑ **Uwaga:** Nadal możesz napotkać odniesienia do nazwy DLT w niektórych miejscach w Databricks (np. classic SKUs, event log schemas). Istniejący Python API z nazwami DLT jest nadal obsługiwany, ale zaleca się przejście na nową konwencję nazewnictwa.

## Podejście deklaratywne – "CO" zamiast "JAK"

W podejściu deklaratywnym opisujemy pożądany stan danych (co chcemy osiągnąć), zamiast kroków imperatywnych (jak to zrobić). Lakeflow niezależnie analizuje zależności i optymalizuje wykonanie transformacji.

DECLARATIVE DATA PIPELINE FRAMEWORK



# Co Składa się na Lakeflow Pipelines?

Lakeflow Pipelines to framework integrujący kluczowe aspekty inżynierii danych, umożliwiający budowanie niezawodnych i wydajnych pipelines w oparciu o paradygmat Lakehouse.



## Definicje Deklaratywne

Skup się na "**WHAT**" chcesz osiągnąć z danymi, a nie "**HOW**" je zaimplementować. Lakeflow automatycznie zarządza logiką wykonania.



## Automatyczna Orchestracja

Framework samodzielnie zarządza kolejnością zadań, zależnościami, ponownymi próbami w przypadku błędów oraz monitorowaniem postępu.



## Wbudowana Jakość Danych

Zdefiniuj zasady jakości danych bezpośrednio w kodzie pipeline, zapewniając automatyczną validację i kontrolę nad danymi.



## Streaming i Materialized Views

Ujednolicone podejście do przetwarzania danych streamingowych i wsadowych w ramach jednego pipeline.



## Uproszczone Obliczenia

Opcje Serverless minimalizują potrzebę zarządzania infrastrukturą, automatycznie skalując zasoby w zależności od obciążenia.

# Składnia CREATE OR REFRESH

## Tabela strumieniowa

Dla źródeł typu "append-only" (logi, zdarzenia, transakcje):

```
CREATE OR REFRESH STREAMING TABLE bronze_orders
AS SELECT *, current_timestamp() as ingestion_time
FROM read_files(
    "/Volumes/raw/orders/", format=>"csv", header=>true,
inferSchema=>true
)
```

Tabela strumieniowa automatycznie przetwarza nowe pliki.

## Zmaterializowany widok

Dla przekształceń z pełnym odświeżaniem lub aktualizacją inkrementalną:

```
CREATE OR REFRESH MATERIALIZED VIEW silver_orders
AS SELECT
    order_id, customer_id, order_date, total_amount, items
FROM STREAM(bronze_orders)
WHERE order_status != 'cancelled'
```

STREAM() wskazuje na przetwarzanie inkrementalne.

# Kiedy używać Streaming vs. Materialized?

## Streaming Table

- Dane są typu append-only (zdarzenia, logi)
- Wymagana jest semantyka Exactly-once
- Dane docierają w czasie rzeczywistym
- Bronze/Raw Layer – zachowanie wszystkich danych
- CDC z baz danych transakcyjnych

**Przykłady:** logi aplikacji, Kafka streams, CDC z PostgreSQL

## Materialized View

- Wymagane operacje UPDATE/DELETE (SCD2, deduplikacja)
- Agregacje i złożonełączenia
- Gold Layer – metryki biznesowe
- Snapshot Tables – tylko bieżący stan
- Feature Tables dla ML

**Przykłady:** Obliczanie CLV, dziennych aktywnych użytkowników, funkcji rekomendacji produktów

# Przepływy Danych w Lakeflow Pipelines

Przepływ (flow) to kombinacja zapytania (query) i tabeli docelowej (target table), przetwarzająca dane w trybie batch lub streaming mode, przyrostowo w potoku Lakeflow Spark Declarative Pipelines.

## Definicja i Rodzaje Przepływów

- Przepływy (Flows) są tworzone **automatycznie** podczas definiowania tabeli lub widoku (np. CREATE OR REFRESH STREAMING TABLE).
- Mogą być definiowane **jawnie** (CREATE FLOW / @dp.append\_flow) dla złożonych scenariuszy.
- Standardowe przepływy (append flows) **dodają nowe wiersze**, idealne do ingestii i transformacji danych.
- Przepływy **Auto CDC** służą do ingestii danych ze zmianami (Change Data Capture).

## Zastosowania i Korzyści

- Każda aktualizacja potoku wyzwala przepływ (flow), który odświeża tabele.
- Może to być **odświeżanie przyrostowe** (tylko nowe rekordy) lub **pełne odświeżanie**.
- Umożliwia **łączenie danych z wielu źródeł** w jedną tabelę docelową (np. dodawanie nowych regionów) bez pełnego odświeżania.
- Umożliwia **uzupełnianie danych historycznych** za pomocą składni INSERT INTO ONCE.

# Wiele przepływów do jednego celu

## Tworzenie tabeli i przypisywanie przepływu

Możesz najpierw utworzyć tabelę strumieniową (streaming table), a następnie zdefiniować do niej przepływ (flow). Poniższy przykład pokazuje, jak niezależnie utworzyć tabelę i przypisać jeden przepływ, uzyskując te same rezultaty, co przy domyślnym tworzeniu przepływu.

```
-- utwórz tabelę strumieniową  
CREATE OR REFRESH STREAMING TABLE customers_silver;  
  
-- dodaj przepływ do tabeli customers_silver  
CREATE FLOW customers_silver_flow  
AS INSERT INTO customers_silver BY NAME  
SELECT * FROM STREAM(customers_bronze);
```

## Łączenie danych z wielu źródeł

Użycie wielu przepływów dopisujących (append flows) (`append_flow` w Python lub `CREATE FLOW...INSERT INTO` w SQL) jest idealne do łączenia danych z różnych źródeł w jedną tabelę strumieniową (streaming table). Zapewnia to inkrementalne aktualizacje, które są bardziej wydajne niż użycie klauzuli `UNION`.

```
-- utwórz tabelę strumieniową  
CREATE OR REFRESH STREAMING TABLE customers_us;  
  
-- dodaj pierwszy przepływ dopisujący z danych z zachodniego USA  
CREATE FLOW append_us_west  
AS INSERT INTO customers_us BY NAME  
SELECT * FROM STREAM(customers_us_west);  
  
-- dodaj drugi przepływ dopisujący z danych ze wschodniego USA  
CREATE FLOW append_us_east  
AS INSERT INTO customers_us BY NAME  
SELECT * FROM STREAM(customers_us_east);
```

 **Ważne:** Oczekiwania dotyczące jakości danych (data quality expectations) powinny być zdefiniowane na **tabeli docelowej** (np. `customers_us`), a nie w definicji przepływu `@dp.append_flow`.

Przepływy są identyfikowane po nazwie, która służy do zarządzania punktami kontrolnymi strumieniowania (streaming checkpoints). Zmiana nazwy istniejącego przepływu jest traktowana jako utworzenie nowego, co resetuje jego punkty kontrolne. Dodatkowo, ta sama nazwa przepływu nie może być ponownie użyta w ramach jednego pipeline'u.

# Wypełnianie danych historycznych

Lakeflow Pipelines ułatwia backfilling danych historycznych w istniejących potokach. Jest to kluczowe, gdy trzeba przetwarzać dane z okresów nieuwzględnionych w początkowej konfiguracji potoku.

## Definiowanie zakresu

Utwórz nowy flow, na przykład usuwając lub zmieniając `modifiedAfter`.

## Jednorazowe uruchomienie

Użyj składni `INSERT INTO ONCE` do jednorazowego wykonania i uniknięcia duplikatów.

## Integracja

Nowe dane historyczne zostaną płynnie zintegrowane z tabelą docelową.

```
-- create the streaming table  
CREATE OR REFRESH STREAMING TABLE registration_events_raw;
```

```
-- original incremental flow (e.g., from 2025 onwards)  
CREATE FLOW registration_events_raw_incremental  
AS INSERT INTO registration_events_raw BY NAME  
SELECT * FROM STREAM read_files(  
    "/Volumes/gc/demo/apps_raw/event_registration/*",  
    format => "json",  
    modifiedAfter => "2024-12-31T23:59:59.999+00:00"  
);
```

```
-- one-time backfill for 2024  
CREATE FLOW registration_events_raw_backfill_2024  
AS INSERT INTO ONCE registration_events_raw BY NAME  
SELECT * FROM read_files(  
    "/Volumes/gc/demo/apps_raw/event_registration/year=2024/*",  
    format => "json"  
);
```

# CDC APPLY w Lakeflow Pipelines (SCD Type 1 and 2)

## Co to jest AUTO CDC INTO?

AUTO CDC INTO automatyzuje zarządzanie zmianami danych, zastępując manualną logikę MERGE INTO. Upraszcza integrację danych z systemów transakcyjnych, gdzie śledzenie ewolucji rekordów jest kluczowe.

### KEYS (column)

Określa jedną lub więcej kolumn, które tworzą unikalny klucz rekordu.

### APPLY AS DELETE WHEN ...

Definiuje warunek logiczny, kiedy rekord źródłowy powinien być traktowany jako usunięcie.

### APPLY AS TRUNCATE WHEN ...

Definiuje warunek, który powoduje usunięcie wszystkich rekordów w tabeli docelowej przed zastosowaniem zmian.

### SEQUENCE BY column

Używane do obsługi zdarzeń poza kolejnością poprzez określenie najnowszej zmiany.

### COLUMNS \* EXCEPT (columns)

Wskazuje wszystkie kolumny źródłowe do uwzględnienia, z wyjątkiem tych wymienionych.

### STORED AS SCD TYPE 1

Zmiany aktualizują rekordy, nadpisując stare wartości (tylko najnowsza wersja).

### STORED AS SCD TYPE 2

Zmiany są przechowywane jako nowe rekordy z kolumnami `_START_AT` i `_END_AT` do śledzenia historii.

## SCD Type 1: Zachowanie Najnowszych Danych

W SCD Type 1 aktualizacje danych nadpisują istniejące rekordy w tabeli docelowej, zawsze zapewniając najnowszą wersję danych. Lakeflow Pipelines automatycznie zarządza tą logiką.

```
CREATE OR REFRESH STREAMING TABLE target;
```

```
CREATE FLOW flowname AS AUTO CDC INTO target
  FROM stream(cdc_data.users)
  KEYS (userId)
  APPLY AS DELETE WHEN operation = "DELETE"
  APPLY AS TRUNCATE WHEN operation = "TRUNCATE"
  SEQUENCE BY sequenceNum
  COLUMNS * EXCEPT (operation, sequenceNum)
  STORED AS SCD TYPE 1;
```

Po zastosowaniu aktualizacji SCD Type 1, tabela docelowa zawiera najnowsze rekordy:

userId	nazwa	miasto
124	Raul	Oaxaca
125	Mercedes	Guadalajara
126	Lily	Cancun

## SCD Type 2: Śledzenie Zmian Historycznych

SCD Type 2 śledzi wszystkie zmiany, zachowując historyczne wersje rekordów. Każda zmiana tworzy nowy rekord z nowym okresem ważności (`_START_AT` i `_END_AT`), co jest kluczowe dla analizy zmian w czasie.

```
CREATE OR REFRESH STREAMING TABLE target;
```

```
CREATE FLOW target_flow AS AUTO CDC INTO target
  FROM stream(cdc_data.users)
  KEYS (userId)
  APPLY AS DELETE WHEN operation = "DELETE"
  SEQUENCE BY sequenceNum
  COLUMNS * EXCEPT (operation, sequenceNum)
  STORED AS SCD TYPE 2;
```

Po zastosowaniu aktualizacji SCD Type 2, tabela docelowa zawiera rekordy historyczne:

userId	nazwa	miasto	_START_AT	_END_AT
123	Isabel	Monterrey	1	5
123	Isabel	Chihuahua	5	6

# Automatyczne zadania konserwacyjne

Lakeflow Pipelines autonomicznie wykonuje kluczowe zadania konserwacyjne na zarządzanych tabelach, wykorzystując zaawansowaną predictive optimization. Zapewnia to lepszą query performance i redukcję kosztów poprzez usuwanie nieaktualnych danych i ich organizowanie.

## Kluczowe operacje

Potoki wykonują operacje **OPTIMIZE** (kompresja plików) i **VACUUM** (usuwanie starych danych), zapewniając optymalne wykorzystanie pamięci masowej.

## Inteligentne planowanie

Zadania konserwacyjne są planowane przez mechanizm predictive optimization i uruchamiane tylko po aktualizacji potoku.

## Korzyści

Automatyczna konserwacja utrzymuje tabele w optymalnym stanie, co skutkuje **zwiększoną query performance**, **zredukowanym resource consumption** i **niższymi storage costs**.

- ❑ Aby dowiedzieć się więcej o częstotliwości predictive optimization i związanych z nią kosztach, zapoznaj się z **predictive optimization system table**.

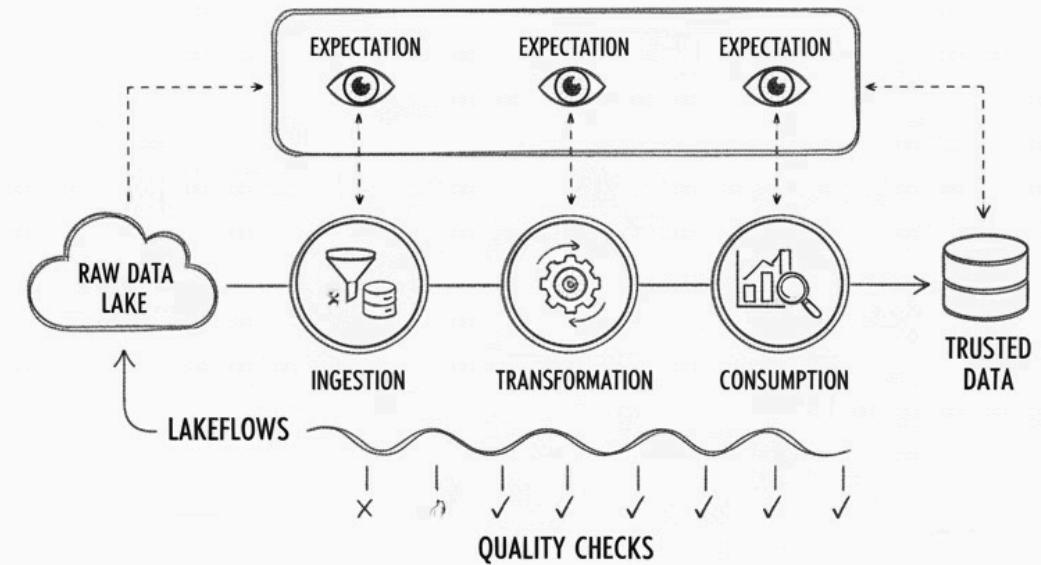
# Expectations

W kontekście Lakeflow Pipelines, „Expectations” definiują zasady jakości danych i testy, które są automatycznie weryfikowane podczas przetwarzania. Zapewniają one niezawodność danych i wcześnie wykrywanie anomalii.

## *Expectations in Lakeflows*

# Data Quality - Expectation

Expectations to deklaratywne reguły jakości danych, wbudowane bezpośrednio w pipeline. Działają jak unit tests dla danych, automatycznie weryfikując warunki przy każdym przetwarzaniu. Blokują one propagację błędnych danych do downstream tables.



# Rodzaje reakcji na naruszenie

## DELETE ROW

Wiersze, które nie spełniają warunku, są automatycznie pomijane. Stosuj, gdy błędne dane mogą być bezpiecznie ignorowane.

```
CONSTRAINT valid_email  
EXPECT (email LIKE '%@%')  
ON VIOLATION DROP ROW
```

## CANCEL UPDATE

Cała partia jest odrzucana, jeśli jakikolwiek wiersz narusza warunek. Stosuj w przypadku krytycznych zasad biznesowych.

```
CONSTRAINT positive_amount  
EXPECT (amount > 0)  
ON VIOLATION FAIL UPDATE
```

## TRACK ONLY (DEFAULT)

Naruszenie jest rejestrowane w metrics, ale dane nadal przepływają. Idealne do monitorowania jakości bez blokowania pipeline.

```
CONSTRAINT recent_data  
EXPECT (  
    event_date >= current_date() - 7  
)
```

**Metrics:** Wszystkie naruszenia są automatycznie rejestrowane w pipeline event log i są dostępne w user interface jako metrics dla każdej tabeli.

# Najlepsze Praktyki dla Expectations

## Dlaczego jest to ważne?

Dobre praktyki ułatwiają stosowanie, aktualizowanie i audytowanie expectations rules w wielu datasets i pipelines, bez modyfikowania code. Zwiększą validation consistency, redukują maintenance costs i poprawią solution portability.

### Separacja Definicji

Przechowuj expectation definitions oddzielnie od pipeline logic

### Tagging

Dodaj custom tags, aby grupować powiązane expectations

### Możliwość Ponownego Użycia

Stosuj te same expectations w wielu pipelines

# Scentralizowane Repozytorium Oczekiwania: Delta Table

Tabela `rules` centralizuje reguły walidacji, przechowując nazwę, `SQL constraint` i `tag` do filtrowania.

```
CREATE OR REPLACE TABLE rules AS
SELECT col1 AS name, col2 AS constraint, col3 AS tag
FROM (
VALUES
("website_not_null",
 "Website IS NOT NULL",
 "validity"),
("fresh_data",
 "to_date(updateTime,'M/d/yyyy h:m:s a') > '2010-01-01'",
 "maintained"),
("social_media_access",
 "NOT(Facebook IS NULL AND Twitter IS NULL AND Youtube IS NULL)",
 "maintained")
)
```

Upraszcza zarządzanie regułami za pomocą `SQL`, umożliwiając łatwe dodawanie nowych ograniczeń i kategoryzację za pomocą `business tags`.

# Stosowanie reguł w potoku Python

Funkcja `get_rules()` pobiera reguły walidacji z tabeli, dopasowując je według tagu. Dekorator `@dp.expect_all_or_drop()` automatycznie usuwa rekordy, które nie spełniają tych kryteriów.

```
from pyspark import pipelines as dp
from pyspark.sql.functions import expr, col

def get_rules(tag):
    df = spark.read.table("rules").filter(col("tag") == tag).collect()
    return {row['name']: row['constraint'] for row in df}

@dp.table
@dp.expect_all_or_drop(get_rules('validity'))
def raw_farmers_market():
    return (
        spark.read.format('csv').option("header", "true")
        .load('/databricks-datasets/data.gov/farmers_markets_geographic_data/')
    )

@dp.table
@dp.expect_all_or_drop(get_rules('maintained'))
def organic_farmers_market():
    return spark.read.table("raw_farmers_market").filter(expr("Organic = 'Y'"))
```

# Ewolucja Schematu

Ten wzorzec umożliwia migrację źródeł danych i zarządzanie wieloma wersjami, zapewniając kompatybilność wsteczną i jakość danych.

```
CREATE OR REFRESH MATERIALIZED VIEW evolving_table(  
    CONSTRAINT valid_migrated_data EXPECT (  
        (col1 IS NOT NULL AND col2 IS NOT NULL) AND  
        (CASE WHEN col3 IS NOT NULL THEN col3 > 0 ELSE TRUE END)  
    ) ON VIOLATION FAIL UPDATE  
    ) AS  
    SELECT * FROM new_source  
    UNION  
    SELECT *, NULL as col3 FROM legacy_source;
```

UNION łączy nowe i stare źródła. CASE obsługuje opcjonalne kolumny. ON VIOLATION FAIL UPDATE zatrzymuje potok w przypadku krytycznych błędów.

# Weryfikacja Liczby Wierszy

Monitorowanie liczby wierszy w materialized views za pomocą COUNT(\*) pozwala na szybką weryfikację rekordów i identyfikację anomalii, pomagając w utrzymaniu jakości danych.

```
CREATE OR REFRESH MATERIALIZED VIEW  
    count_verification(  
        CONSTRAINT no_rows_dropped  
        EXPECT (a_count == b_count)  
    ) AS  
    SELECT * FROM  
        (SELECT COUNT(*) AS a_count FROM table_a),  
        (SELECT COUNT(*) AS b_count FROM table_b)
```

Ograniczenie EXPECT (a\_count == b\_count) automatycznie wykrywa rozbieżności w liczbie wierszy między tabelami źródłowymi.

# Wykrywanie brakujących rekordów i unikalności kluczy

## Wykrywanie brakujących rekordów

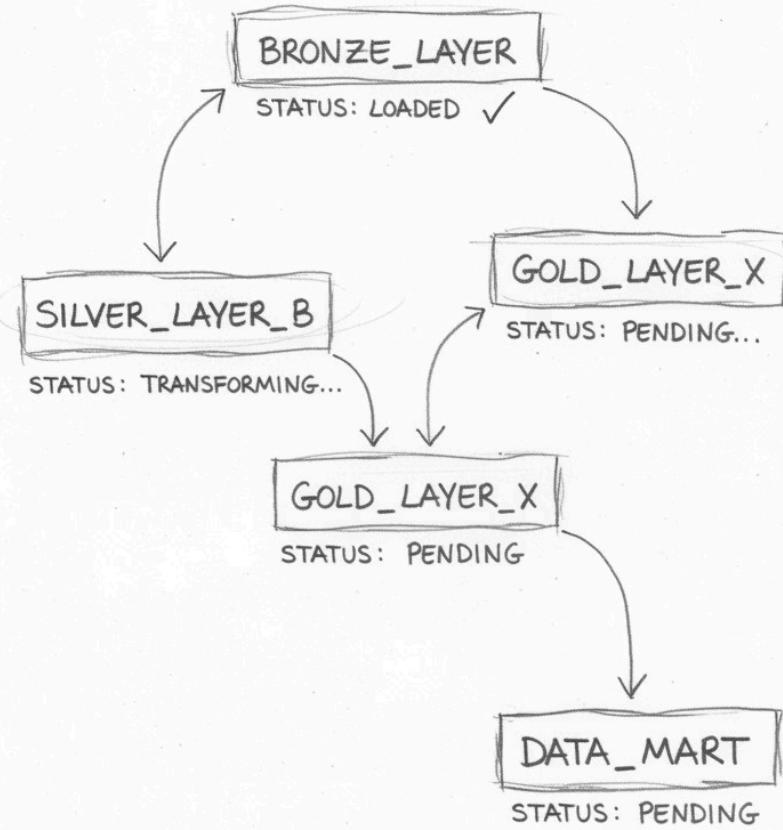
```
CREATE OR REFRESH MATERIALIZED VIEW  
report_compare_tests(  
    CONSTRAINT no_missing_records  
    EXPECT (r_key IS NOT NULL)  
) AS  
SELECT v.*, r.key as r_key  
FROM validation_copy v  
LEFT OUTER JOIN report r  
ON v.key = r.key
```

LEFT JOIN ujawnia rekordy obecne w validation, ale brakujące w report.

## Walidacja klucza głównego

```
CREATE OR REFRESH MATERIALIZED VIEW  
report_pk_tests(  
    CONSTRAINT unique_pk  
    EXPECT (num_entries = 1)  
) AS  
SELECT pk, count(*) as num_entries  
FROM report  
GROUP BY pk
```

GROUP BY wykrywa duplikaty primary key poprzez zliczanie wystąpień.



## DATABRICKS DATA PIPELINE TABLE DEPENDENCIES & STATUS

# Widok DAG w interfejsie Lakeflow

Interfejs Databricks UI automatycznie generuje wizualny DAG (Directed Acyclic Graph) wszystkich tabel w potoku. Każdy węzeł pokazuje status (running/success/failure), metryki przetwarzania i naruszenia jakości danych. Możesz kliknąć na węzeł, aby zobaczyć kod definicji, podgląd danych i szczegółowe logi.

## Co pokazuje DAG

- Zależności tabel (upstream → downstream)
- Status każdej tabeli w czasie rzeczywistym
- Liczba wierszy i objętość danych
- Czas trwania przetwarzania
- Podsumowanie naruszeń oczekiwania

## Interakcja

- Kliknięcie węzła → szczegóły tabeli
- Najechanie myszką → szybki podgląd metryk
- Widok osi czasu → historyczne uruchomienia
- Szczegółowa analiza błędów
- Śledzenie ewolucji schematu

# Standard Spark kontra Lakeflow Pipelines

Cecha	Standard Spark SQL/Python	Lakeflow Pipelines (DLT)
Model programowania	SQL/DataFrame API, więcej kodu szablonowego.	Rozszerzony SQL/Python z DLT (`CREATE OR REFRESH`, `EXPECT`).
Automatyzacja infrastruktury	Ręczna konfiguracja klastra, zarządzanie punktami kontrolnymi.	Automatyczne zarządzanie klastrem, zarządzanie punktami kontrolnymi, odświeżanie.
Jakość danych	Niestandardowa implementacja reguł jakości danych.	Wbudowane oczekiwania (`EXPECT`) z konfigurowalnymi reakcjami.
Monitorowanie i obserwonalność	Integracja z zewnętrznymi narzędziami lub niestandardowa implementacja.	Automatyczny DAG, metryki, logi i statusy w UI.
Zarządzanie zależnościami	Zewnętrzny orkiestrator (np. Airflow).	Automatyczne wykrywanie i zarządzanie zależnościami tabel.
Optymalizacja wydajności	Ręczna optymalizacja (np. partycjonowanie, indeksowanie).	Automatyczne optymalizacje Photon, buforowanie i skalowanie.

# Automatyzacja – co zyskujemy za darmo?

## Data Lineage

Automatyczny wykres zależności tabel wskazuje źródła i konsumentów, propagując zmiany w górę strumienia.

## State Management

Automatyczne zarządzanie punktami kontrolnymi i przetwarzanie stanu, eliminując ręczną konfigurację i śledzenie plików.

## Incremental Refresh

Potok przetwarza tylko zmienione dane (delta), oszczędzając zasoby i skracając czas wykonania 10–100 razy.

## Error Handling

Automatyczne ponawianie prób i stopniowa degradacja w przypadku błędów, zapobiegając zawieszaniu się potoku z powodu wadliwych rekordów.

## Observability

Wbudowane metryki (liczba wierszy, czas przetwarzania, jakość danych) dostępne w interfejsie użytkownika, bez dodatkowej instrumentacji kodu.

## Optimization

Automatyczna optymalizacja planu wykonania i buforowanie wyników, zapewniające wydajność bez ręcznej interwencji.

# Compute

Lakeflow Pipelines oferuje dwie opcje obliczeniowe, dostosowane do różnych obciążen danych.

## Serverless

Build streaming and batch pipelines on a fully managed serverless platform requiring minimal additional configuration. Available in two modes - **Performance Optimized** and **Standard**

\$0.45  
/ DBU

Includes underlying compute costs

# Serverless

Serverless dla Lakeflow Pipelines to w pełni zarządzana opcja. Eliminuje konfigurację klastra, a cena jest wliczona w koszt DBU, bez dodatkowych opłat za infrastrukturę chmurową. Dostępne są dwie warianty:

- **Standard:** do 70% taniej, czas uruchomienia 4–6 minut.
- **Performance Optimized:** szybsze uruchamianie (<1 minuta), z Enzyme (incremental refresh), Stream Pipelining i automatycznym skalowaniem zapobiegającym problemom OOM.

Obie warianty mają automatycznie włączony silnik Photon, zapewniają automatyczny wybór instancji i skalowanie. Obsługują również zaawansowane funkcje: CDC, SCD Type 2 i zasady jakości danych.

# Classic Clusters

Classic mode for Lakeflow Pipelines wykorzystuje zaawansowane clusters, które wymagają ręcznej konfiguracji. Wariant ten wiąże się z osobną płatnością za cloud instances oraz kosztem Lakeflow Pipelines DBU. Cluster startup time wynosi od 1 do 6 minut, a Photon engine jest opcjonalny. Należy pamiętać, że zaawansowane funkcje, takie jak CDC, SCD Type 2 i data quality rules, są niedostępne w Classic Core.

Classic Core	Classic Pro	Classic Advanced
Easily build scalable streaming or batch pipelines in SQL and Python	Easily build scalable streaming or batch pipelines in SQL and Python and handle change data capture (CDC) from any data source	Easily build scalable streaming or batch pipelines in SQL and Python, handle change data capture (CDC) and maximize your data credibility with quality expectations and monitoring
<b>\$0.30</b> / DBU <small>Plus underlying compute costs billed by cloud provider</small>	<b>\$0.38</b> / DBU <small>Plus underlying compute costs billed by cloud provider</small>	<b>\$0.54</b> / DBU <small>Plus underlying compute costs billed by cloud provider</small>

# Porównanie: Classic vs Serverless

	Classic	Serverless Standard	Serverless Performance Optimized
Typowy przypadek użycia	Tradycyjne obciążenia, pełna kontrola	Ogólne potoki danych, niższe koszty	Potoki danych o wysokiej wydajności, szybki start
Change Data Capture (CDC)	Niedostępne	Tak	Tak
Slowly Changing Dimensions (SCD) Type 2	Niedostępne	Tak	Tak
Reguły oczekiwania jakości danych	Niedostępne	Tak	Tak
Photon Engine	Opcjonalnie (2.5X DBU)	Automatycznie	Automatycznie
Automatyczny wybór instancji	Ręcznie	Automatyczny	Automatyczny
Automatyczne skalowanie (OOM)	Nie	Nie	Tak
Enzym (Incremental Refresh)	Nie	Nie	Tak
Stream Pipelines	Nie	Nie	Tak
Czas uruchamiania	4–6 min	4–6 min	<1 min
Całkowity koszt	Instancje płatne oddzielnie	Wliczone w cenę (do 70% taniej)	Wliczone w cenę

# Lakeflow Pipelines vs MLV (Fabric)

W kontekście nowoczesnych architektur lakehouse, Materialized Lake Views (MLV) i Lakeflow służą do automatyzacji potoków danych, ale różnią się zakresem i platformą.

Cecha	Lakeflow Pipelines (Databricks)	Microsoft Fabric MLV
<b>Ekosystem</b>	Databricks (Unified Platform)	Microsoft Fabric (SaaS)
<b>Zakres</b>	Cały cykl życia danych (ingest, ETL, orchestration)	Pojedynczy obiekt/transformacja
<b>Język</b>	SQL i Python	SQL (głównie)
<b>Główny cel</b>	Skalowalne, profesjonalne potoki danych	Szybkie dostarczanie danych, uproszczenie architektury Medallion
<b>Mechanizm</b>	Automatyczna orchestration, incremental processing, autoscaling, real-time	Deklaratywne podejście – system zarządza odświeżaniem (refresh)
<b>Typy obiektów</b>	Streaming Tables i Materialized Views	Materialized Lake Views (format Delta w OneLake)
<b>Jakość danych</b>	Wbudowany EXPECT (konfigurowalne reakcje)	Wbudowany monitoring (np. CONSTRAINT ON MISMATCH)
<b>CDC / SCD Type 2</b>	Wbudowana składnia (APPLY CHANGES INTO)	Wymaga manualnej implementacji
<b>Monitoring</b>	Automatyczny DAG, metryki, logi	Automatyczna wizualizacja pochodzenia danych (lineage), monitoring w Fabric

# DEMO

Let's see Databricks in action!

# Orkiestracja

Lakeflow Jobs

# Od Kodu do Produkcji

Lakeflow Jobs (Databricks Workflows) to zarządzana platforma orkiestracji, która przekształca notatniki i potoki w procesy gotowe do produkcji. Zero zewnętrznych zależności – wszystko w jednym miejscu: planowanie, monitorowanie, alerty, logika ponownych prób.

## Rozwój

Piszesz kod w notatkach lub definicjach potoków Lakeflow, testując lokalnie lub na klastrze deweloperskim

## Definicja Zadania

Definiujesz zadania, zależności, wymagania obliczeniowe w UI lub za pośrednictwem API

## Planowanie

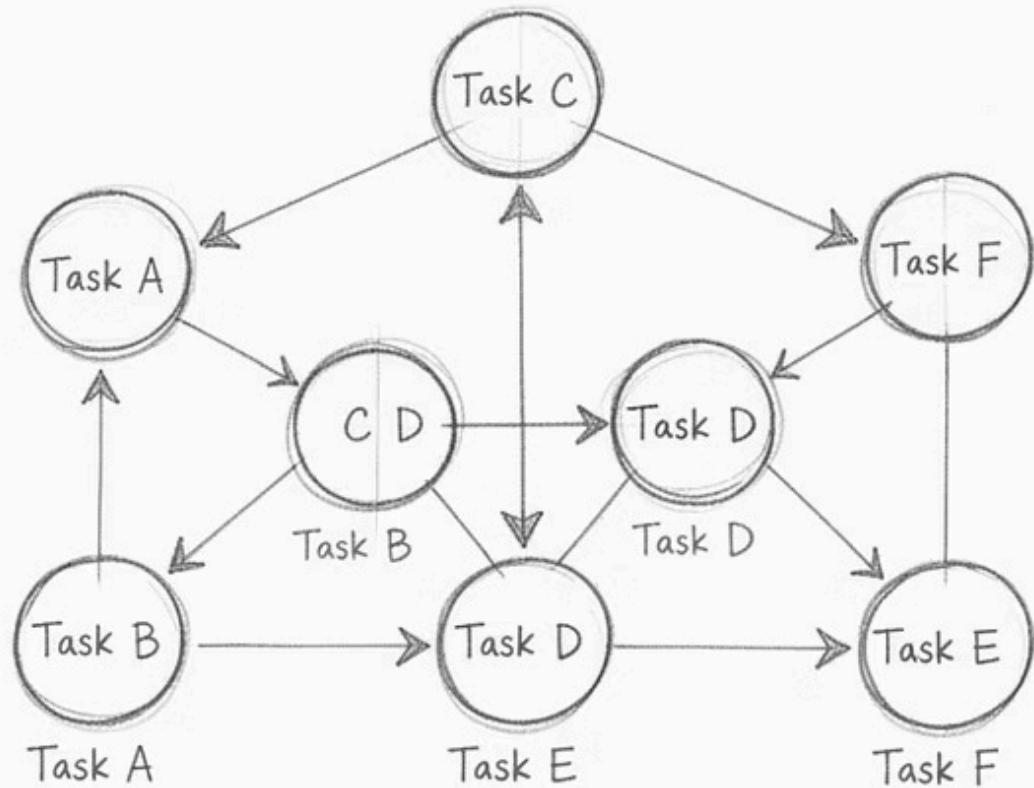
Konfigurujesz wyzwalacz: harmonogram CRON, nadejście pliku, ciągły lub ręczny

## Wykonanie

Zadanie uruchamia się automatycznie, zarządzając zasobami, ponownymi próbami i powiadomieniami

## Monitorowanie

Obserwujesz postępy w czasie rzeczywistym, analizujesz historyczne przebiegi i otrzymujesz alerty



## Struktura Zadania – Taski i Zależności

Zadanie składa się z jednego lub więcej tasków ułożonych w DAG (Directed Acyclic Graph). Każdy task jest niezależną jednostką wykonawczą – może to być notebook, pipeline Lakeflow, skrypt Python, zapytanie SQL, a nawet transformacja dbt.

# Typy Tasków

## Notebook Task

Uruchamia notebook Databricks z parametrami. Najpopularniejszy typ dla niestandardowej logiki i eksploracyjnych przepływów pracy.

## Lakeflow Pipeline Task

Wykonuje całą definicję pipeline Lakeflow. Automatycznie propaguje zmiany w całym DAG tabel.

## Python Task

Uruchamia samodzielny skrypt Python (plik wheel). Użyj do lekkich operacji bez potrzeby notebooka.

## SQL Task

Wykonuje zapytanie SQL w SQL Warehouse. Idealny do prostych kontroli jakości danych lub zapytań raportujących.

## JAR Task

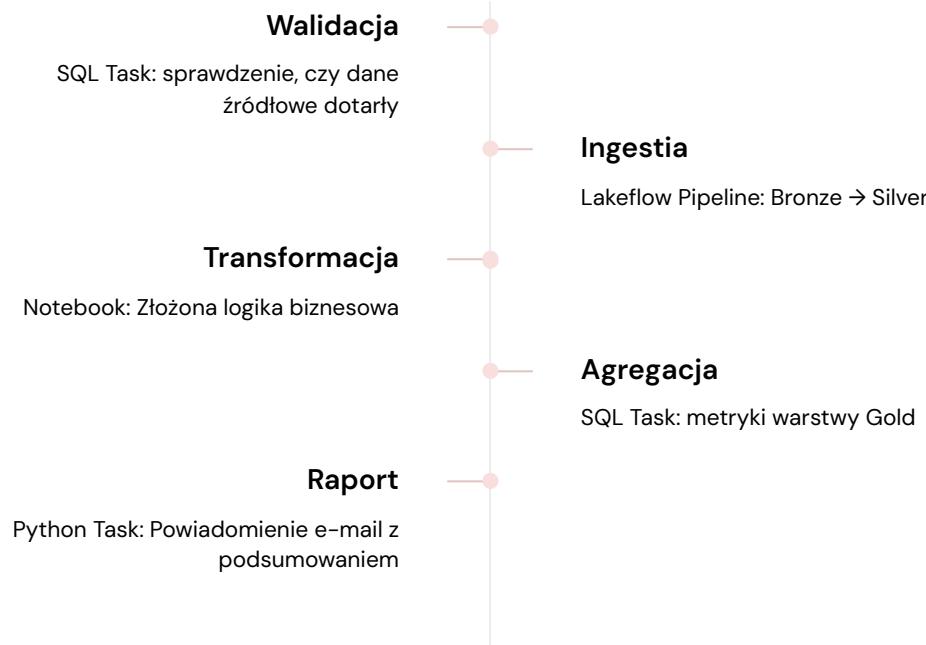
Uruchamia aplikację Scala/Java. Do starszego kodu lub wysoko wydajnych zadań Spark napisanych w Scala.

## Run Job Task

Wywołuje inne zadanie – umożliwia kompozycję i ponowne użycie. Modularyzacja złożonych przepływów pracy.

# Przykład: Wieloetapowe zadanie ETL

## Przepływ Danych



## Konfiguracja Zależności

```
# W definicji zadania JSON:  
{  
  "tasks": [  
    {"task_key": "validate", "sql_task": {...}},  
    {  
      "task_key": "ingest",  
      "pipeline_task": {...},  
      "depends_on": [{"task_key": "validate"}]  
    },  
    {  
      "task_key": "transform",  
      "notebook_task": {...},  
      "depends_on": [{"task_key": "ingest"}]  
    },  
    {  
      "task_key": "aggregate",  
      "sql_task": {...},  
      "depends_on": [{"task_key": "transform"}]  
    },  
    {  
      "task_key": "report",  
      "python_wheel_task": {...},  
      "depends_on": [{"task_key": "aggregate"}]  
    }  
  ]  
}
```

Zadania wykonują się w kolejności zdefiniowanej przez zależności. Jeśli walidacja się nie powiedzie, pozostałe zadania zostaną pominięte.

**Demo UI:** Zobacz na żywo w interfejsie użytkownika Databricks Workflows – interaktywny wykres DAG ze statusem każdego zadania w czasie rzeczywistym.

# Wyzwalacze – Kiedy uruchomić zadanie?

## Harmonogram (CRON)

Klasyczny wyzwalacz oparty na czasie. Definiujesz, kiedy zadanie ma zostać uruchomione, używając wyrażenia CRON lub prostego UI.

**Przypadek użycia:** Codzienny ETL o 2:00, godzinne odświeżanie metryk, raporty na koniec miesiąca

```
0 2 * * *
```

```
0 */4 * * *
```

## Ciągły

Zadanie uruchamia się ponownie natychmiast po zakończeniu poprzedniego przebiegu. Do przetwarzania bliskiego czasu rzeczywistego.

**Przypadek użycia:** Potoki streamingowe, przetwarzanie CDC, pulpty nawigacyjne na żywo

 Uwaga na koszty – klaster działa non-stop

## Pojawienie się Pliku

Wyzwalacz sterowany zdarzeniami, który reaguje na pojawienie się nowych plików w określonej lokalizacji (S3, ADLS, DBFS).

**Przypadek użycia:** Przetwarzanie plików dostawców, dostarczanie danych partnerów, automatyczny import

```
Reaguje w ciągu <5 minut od pojawienia się pliku
```

## Ręczny / API

Uruchamiany ręcznie poprzez UI, CLI lub REST API. Do uruchomień ad-hoc i integracji z zewnętrznymi orkiestratorami.

**Przypadek użycia:** Jednorazowe ładowanie danych, testowanie, integracja z Airflow/Azure Data Factory

```
databricks jobs run-now --job-id 123
```

## Aktualizacja Tabeli

Wyzwalacz, który uruchamia się po aktualizacji określonej tabeli Delta. Monitoruje zmiany w tabeli i automatycznie uruchamia zadanie.

**Przypadek użycia:** Przetwarzanie danych zależnych, wyzwalane przez aktualizację tabeli nadzędnej. Potoki sterowane zdarzeniami, które reagują na dostępność danych.

**Przykład:** Gdy warstwa bronze zostanie odświeżona, automatycznie wyzwól transformację warstwy silver.

# Typy Parametryzacji

## Parametry Zadania

Globalne dla całego zadania, dostępne we wszystkich taskach:

```
# W job config:  
"parameters": {  
    "environment": "prod",  
    "date": "2024-01-15"  
}  
  
# W notebook:  
env =  
dbutils.widgets.get("environment")
```

## Parametry Taska

Specyficzne dla pojedynczego taska, nadpisują parametry zadania:

```
"notebook_task": {  
    "notebook_path": "/ETL/transform",  
    "base_parameters": {  
        "input_table": "silver.orders",  
        "output_table": "gold.daily_metrics"  
    }  
}
```

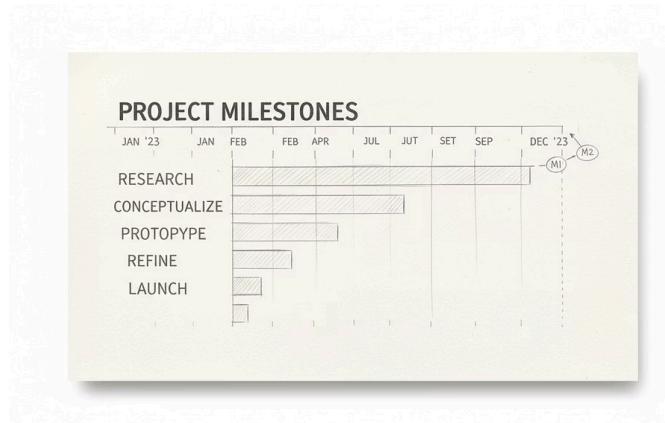
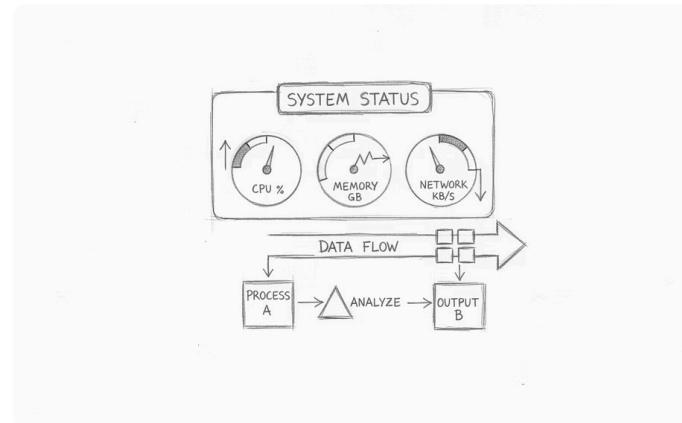
## Wartości Taska

Przekazywanie danych między taskami – wyjście jednego jest wejściem następnego:

```
# Task 1 - writes result:  
dbutils.jobs.taskValues.set(  
    key="row_count",  
    value=df.count()  
)  
  
# Task 2 - reads:  
count = dbutils.jobs.taskValues.get(  
    taskKey="task_1",  
    key="row_count")
```

# Obserwowałość – Monitorowanie i Alerty

Databricks zapewnia wielopoziomową obserwowałość: metryki na poziomie zadań, szczegóły na poziomie tasków, wykorzystanie klastra i wydajność na poziomie zapytań. Wszystko dostępne od ręki, bez potrzeby zewnętrznych narzędzi.



## Historia Uruchomień Zadania

Każde uruchomienie jest zapisywane z pełnym kontekstem: czas trwania, status, parametry, konfiguracja klastra, logi wyjściowe. Historyczne trendy i porównania między uruchomieniami.

## Oś Czasu Tasków

Wykres Gantt pokazujący wykonanie każdego tasku, zależności, wąskie gardła. Zidentyfikuj, które taski są wolne i odpowiednio je zoptymalizuj.

## Alerty

Powiadomienia Email, Slack, PagerDuty, webhook o awarii zadania, przekroczeniu czasu lub sukcesie. Konfigurowalne progi i logika ponawiania prób.

# Tabele Systemowe dla Zaawansowanej Analityki

Databricks zapisuje wszystkie metadane wykonania zadań w System Tables – zarządzanych tabelach Delta, które można odpytywać za pomocą SQL. Umożliwia to tworzenie niestandardowych raportów, analizę trendów, przypisywanie kosztów i monitorowanie SLA.

## Kluczowe Tabele

- **system.lakeflow.job\_run\_timeline:** Historia wszystkich uruchomień zadań z czasem trwania, statusem i kosztem
- **system.lakeflow.task\_run\_timeline:** Szczegółowa analiza każdego uruchomienia na poziomie tasków
- **system.compute.clusters:** Wykorzystanie klastra i czas pracy
- **system.billing.usage:** Zużycie DBU na zadanie/task

## Przykładowe zapytanie

```
-- 10 najdłużej trwających uruchomień zadań
SELECT
    job_name,
    run_id,
    start_time,
    ROUND(run_duration_sec/60, 2) as duration_min,
    result_state
FROM system.lakeflow.job_run_timeline
WHERE start_time >= current_date() - 7
ORDER BY run_duration_sec DESC
LIMIT 10
```

Możesz przekształcić te zapytania w dashboardy w Databricks SQL, zaplanowane raporty lub alerty (np. „powiadom, jeśli którekolwiek zadanie działa dłużej niż 2 godziny”)

**Notebook:** Notebook 03, sekcja „Observability with System Tables”

# DEMO

Let's see Databricks in action!

# Unity Catalog

Governance

# Czym jest Unity Catalog?

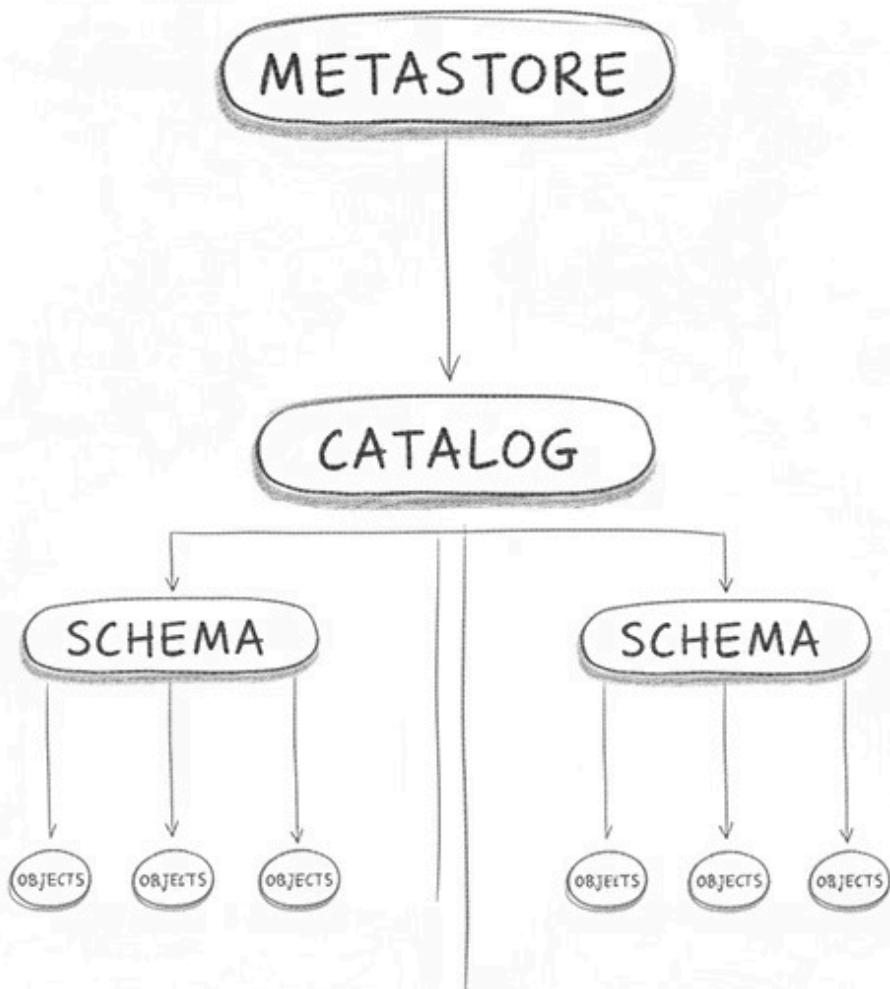
Unity Catalog to ujednolicona warstwa zarządzania dla całego lakehouse – pojedynczy metastore dla tables, volumes (files), models, functions i wielu innych. Centralizuje zarządzanie dostępem, audyty i lineage w jednym miejscu, niezależnie od cloud provider czy compute engine.

## Przed Unity Catalog

- Osobne Hive metastores dla każdego workspace
- Uprawnienia na poziomie plików (S3 policies, ACLs)
- Sfragmentowany ślad audytu
- Brak centralnego discovery
- Ręczne zarządzanie udostępnianiem danych

## Z Unity Catalog

- Pojedynczy metastore dla wszystkich workspaces
- Szczegółowa kontrola dostępu w SQL
- Automatyczne logowanie audytu
- Wbudowane data discovery & search
- Bezpieczne Delta Sharing od razu po wyjęciu z pudełka



## Architektura Unity Catalog

### Metastore

Kontener najwyższego poziomu, jeden na region. Przechowuje metadane dla wszystkich obiektów. Może być współdzielony między wieloma workspace'ami.

### Catalog

Pierwszy poziom organizacji – logiczne grupowanie danych. Zazwyczaj per domena biznesowa lub środowisko (prod\_catalog, staging\_catalog).

### Schema (Database)

Drugi poziom – organizacja w ramach Catalog. Często per aplikacja, zespół lub domena danych (sales, marketing, finance).

### Objects

Specyficzne zasoby: Tables, Views, Volumes (pliki), Models, Functions. Uprawnienia są nadawane na tym poziomie.

**Składnia trójpoziomowej przestrzeni nazw:**

`catalog_name.schema_name.object_name`

Przykład: `production.sales.daily_orders` lub `ml_prod.features.customer_ltv`

USER/ROLE	TABLES	VOLUMES	MODELS
admin	SELECT, INSERT, UPDATE, DELETE	READ EXECUTE	CREATE, READ UPDATE DELETE, TRAIN
admin	GRANT	MOUNT	
data_analyst	SELECT READ_ONLY	READ EXECUTE	READ PREDICT
developer	SELECT, READ_ONLY ALTER	READ WRITE	
guest	SELECT, INSERT, UPDATE, UPDATE, ALTER	CREATE, READ UPDATE, UPDATE, TRAIN	CREATE, READ UPDATE, TRAIN
guest	SELECT (limited)	NONE	PREDICT (limited)

## Kontrola Dostępu – Uprawnienia oparte na SQL

Unity Catalog używa standardowej składni SQL GRANT/REVOKE do zarządzania uprawnieniami. To samo podejście dotyczy tables, volumes, models – spójna składnia, łatwa do audytu i automatyzacji.

# Kluczowe Uprawnienia

## SELECT

Dostęp do odczytu – użytkownik może wykonywać zapytania do danych

```
GRANT SELECT ON TABLE  
prod.sales.orders  
TO `analysts@company.com`
```

## MODIFY

Dostęp do zapisu – INSERT, UPDATE, DELETE, MERGE

```
GRANT MODIFY ON TABLE  
prod.sales.orders  
TO `etl_service_principal`
```

## USAGE

Dostęp do katalogu/schematu – wymagany do przeglądania obiektów wewnętrz

```
GRANT USAGE ON CATALOG prod  
TO `data_consumers`
```

## CREATE

Możliwość tworzenia nowych obiektów w schemacie/katalogu

```
GRANT CREATE TABLE ON SCHEMA  
prod.analytics  
TO `data_engineers`
```

## WSZYSTKIE UPRAWNIENIA

Poły dostęp – wszystkie operacje (używać ostrożnie!)

```
GRANT ALL PRIVILEGES ON CATALOG  
dev  
TO `platform_admins`
```

## Dziedziczenie

Uprawnienia są dziedziczone w dół: KATALOG → SCHEMAT → TABELA

Nadanie uprawnień do katalogu = dostęp do wszystkich schematów i tabel wewnętrz

# Bezpieczeństwo Szczegółowe

Czasami uprawnienia na poziomie tabeli nie są wystarczająco szczegółowe. Unity Catalog wspiera maskowanie na poziomie kolumn (Column Masking) i bezpieczeństwo na poziomie wierszy (Row-Level Security) bez duplikowania danych ani tworzenia wielu wersji tabeli.

## Column Masking

Ukryj lub maskuj wrażliwe kolumny dla konkretnych użytkowników:

- PII (e-mail, telefon, numer ubezpieczenia społecznego)
- Dane finansowe (wynagrodzenia, numery kont)
- Informacje zdrowotne (diagnozy, leki)

**Podejście:** Widoki dynamiczne z logiką warunkową

## Row-Level Security (RLS)

Ogranicz, które wiersze użytkownik może zobaczyć:

- Aplikacje wielodostępne (każdy klient widzi tylko swoje dane)
- Ograniczenia regionalne (zgodność)
- Dostęp hierarchiczny (menedżer widzi tylko swój zespół)

**Podejście:** Predykaty oparte na kontekście użytkownika

# Implementacja Maskowania Kolumn

## Definiowanie zabezpieczonego widoku

```
-- Utwórz bezpieczny widok z warunkowym maskowaniem  
CREATE OR REPLACE VIEW production.sales.orders_secured AS  
SELECT  
    order_id,  
    order_date,  
    product_id,  
    quantity,  
  
-- Maskuj adres e-mail dla osób niebędących administratorami  
CASE  
    WHEN is_account_group_member('admins') THEN customer_email  
    ELSE CONCAT('***@', SPLIT(customer_email, '@')[1])  
END as customer_email,  
-- Całkowicie maskuj kartę kredytową dla osób niebędących administratorami  
CASE  
    WHEN is_account_group_member('admins') THEN credit_card_last4  
    ELSE '****'  
END as credit_card_last4,  
-- Całkowita kwota widoczna tylko dla finansistów i administratorów  
CASE  
    WHEN is_account_group_member('finance') OR is_account_group_member('admins') THEN  
        total_amount  
    ELSE NULL  
END as total_amount  
FROM production.sales.orders_raw;
```

## Jak działa maskowanie i przykładowe użycie

Powyższy kod tworzy zabezpieczony widok (orders\_secured), który dynamicznie maskuje wrażliwe dane w kolumnach na podstawie członkostwa użytkownika w grupie. Dzięki temu różne zespoły widzą tylko te dane, do których mają uprawnienia.

- **customer\_email:** Pełny adres e-mail jest widoczny tylko dla członków grupy 'admins'. Dla innych użytkowników adres jest częściowo maskowany (np. \*\*\*@example.com).
- **credit\_card\_last4:** Ostatnie 4 cyfry karty kredytowej są widoczne tylko dla 'admins'. Dla wszystkich innych użytkowników kolumna wyświetla cztery gwiazdki (\*\*\*\*).
- **total\_amount:** Kwota jest widoczna wyłącznie dla członków grup 'finance' i 'admins'. Dla innych użytkowników wartość w kolumnie wynosi NULL.

### Przykład zapytania:

```
-- Zapytanie z perspektywy zwykłego użytkownika  
SELECT  
    order_id,  
    customer_email,  
    credit_card_last4,  
    total_amount  
FROM production.sales.orders_secured  
WHERE order_id = 'ORD-001';
```

Ten mechanizm zapewnia bezpieczeństwo i zgodność danych z polityką prywatności, bez potrzeby duplikowania danych czy skomplikowanego zarządzania uprawnieniami na poziomie tabeli.

**Kluczowa funkcja:** `is_account_group_member('group_name')` – sprawdza, czy bieżący użytkownik należy do danej grupy

# Implementacja Bezpieczeństwa na Poziomie Wierszy

-- Przykład 1: Wielonajmowy SaaS - każdy klient widzi tylko swoje dane

```
CREATE OR REPLACE VIEW saas_app.customers_filtered AS
SELECT *
FROM saas_app.customers_raw
WHERE
    -- is_account_group_member zwraca true, jeśli użytkownik
    należy do grupy o nazwie customer_id
    is_account_group_member(CAST(customer_id AS STRING))
    OR is_account_group_member('platform_admins');
```

-- Przykład 2: Zgodność regionalna - użytkownicy z UE widzą tylko dane z UE

```
CREATE OR REPLACE VIEW global.transactions_compliant AS
SELECT *
FROM global.transactions_raw
WHERE
    CASE
        WHEN current_user() LIKE '%@company.eu' THEN region =
        'EU'
        WHEN current_user() LIKE '%@company.us' THEN region IN
        ('US', 'LATAM')
        WHEN is_account_group_member('global_admins') THEN
        TRUE
        ELSE FALSE
    END;
```

# Implementacja Maskowania Danych za pomocą Ograniczeń

Unity Catalog wprowadza natywne możliwości maskowania danych za pomocą ograniczeń. Pozwala to deklaratywnie definiować polityki maskowania bezpośrednio na kolumnach tabeli, bez potrzeby tworzenia dodatkowych widoków.

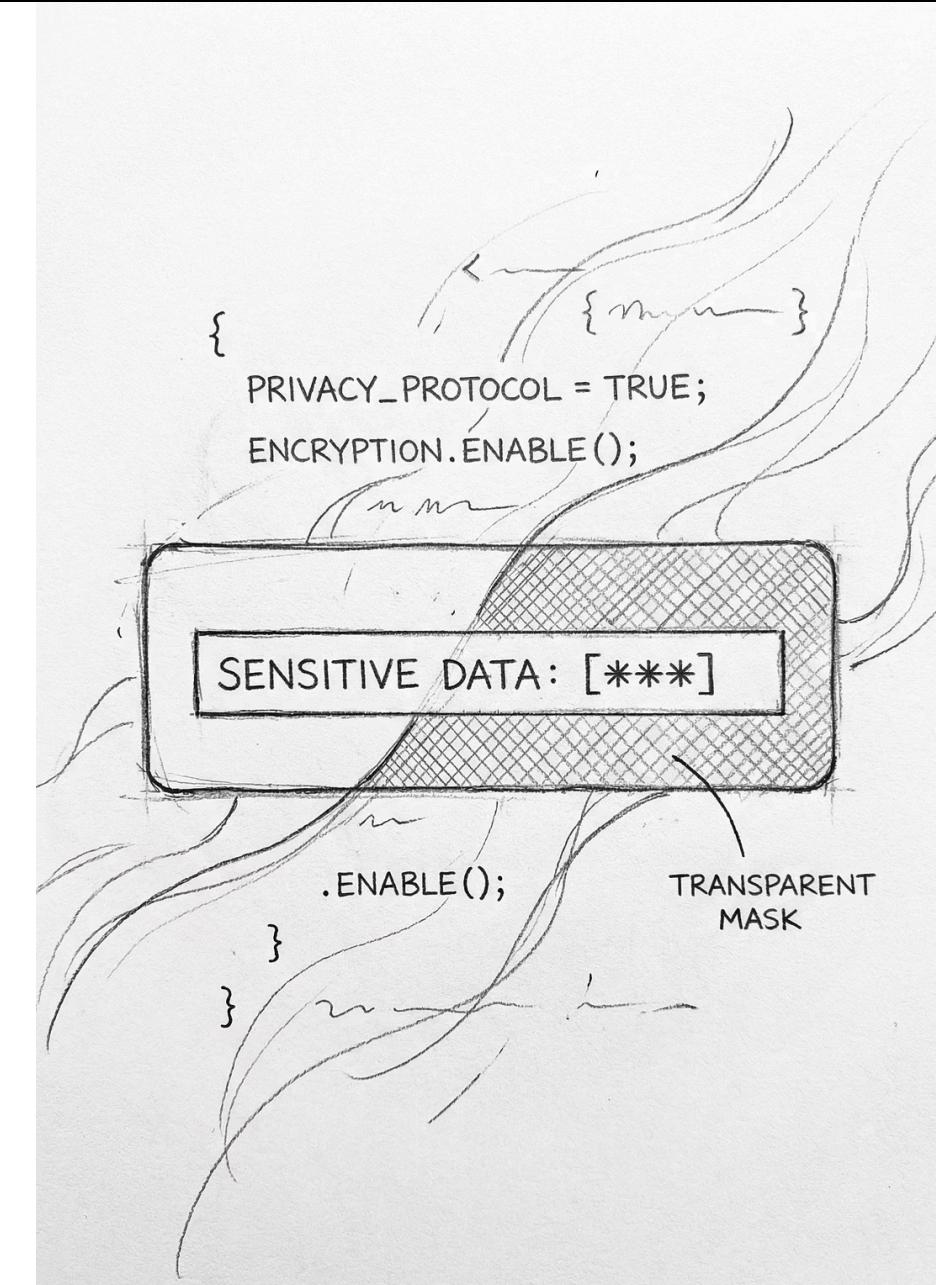
```
-- Create a masking function
```

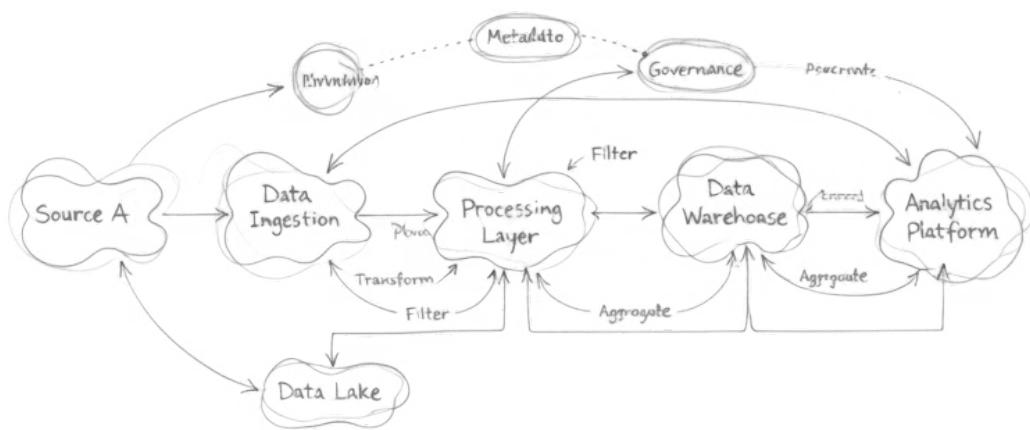
```
CREATE FUNCTION mask_email(email STRING)
RETURNS STRING
RETURN CASE WHEN is_account_group_member('data_stewards') THEN email
ELSE '***@***.com' END;
```

```
-- Apply the masking policy to the column
```

```
ALTER TABLE production.customers
ALTER COLUMN email SET MASK mask_email;
```

Powyższy przykład demonstruje tworzenie funkcji maskującej, która ukrywa adres e-mail dla użytkowników, którzy nie są "data\_stewards". Wszelkie zapytania do tej kolumny automatycznie zastosują zdefiniowaną politykę maskowania.





## Lineage i Audyt

Unity Catalog automatycznie śledzi Lineage danych (skąd pochodzą dane, kto ich używa) i logi Audit (kto wykonywał jakie operacje). Nie jest wymagana żadna konfiguracja – wszystko działa od razu.

# Data Lineage

## Automatyczne śledzenie

Unity Catalog rejestruje Lineage dla każdej operacji:

- Odczyty: które tabele były źródłem
- Zapisy: które tabele zostały utworzone
- Transformacje: kod, który przetwarzał dane
- Kontekst Notebook/Job: kto i kiedy został uruchomiony

Lineage działa w obrębie:

- Zapytania SQL
- PySpark/Scala DataFrames
- Lakeflow Pipelines
- trening ML Model

## Dostęp do danych Lineage

```
-- Query system table
SELECT
    source_table_full_name,
    target_table_full_name,
    source_type,
    created_at,
    created_by
FROM system.access.table_lineage
WHERE
    target_table_full_name =
        'production.sales.daily_metrics'
ORDER BY created_at DESC
LIMIT 10;
```

**UI:** W Catalog Explorer, kliknij na tabelę → zakładka Lineage. Interaktywny wykres pokazujący zależności upstream i konsumentów downstream.

# DEMO

Let's see Databricks in action!