Przetwarzanie danych w skali

Batch · Streaming · Kafka jako szkielet · CDC · Lambda/Kappa · Real-time OLAP

mgr inż. Jakub Woźniak

Politechnika Poznańska · Instytut Informatyki · Semestr letni 2025/26

Plan wykładu

Agenda

- Wzorce przetwarzania wsadowego (batch)
- Fundamentalne problemy przetwarzania strumieniowego (streaming)
- Kafka jako centralny szkielet danych (rozwinięcie z wykładu 3)
- Lambda vs Kappa Architecture
- Change Data Capture w praktyce (rozwinięcie z wykładu 5)
- Wzorce analityki czasu rzeczywistego (real-time OLAP)
- ETL vs ELT
- ☁ Managed streaming i OLAP w chmurze

Przetwarzanie wsadowe (batch)

Fundamentalny wzorzec: Map → Shuffle → Reduce

MapReduce jako wzorzec (nie narzędzie)

Schemat przetwarzania dużych zbiorów danych równolegle.

1. **Map** — niezależne przetwarzanie fragmentów danych → pary `(klucz, wartość)`
2. **Shuffle** — grupowanie par po kluczu, dystrybucja do procesorów odpowiedzialnych za redukcję
3. **Reduce** — agregacja wartości per klucz → wynik końcowy

Dlaczego ten wzorzec działa?

- **Partycjonowanie pracy**: Map jest trywialnie równoległy
- **Determinizm**: te same dane → ten sam wynik
- **Lokalizacja danych**: przetwarzaj tam, gdzie leżą

Uniwersalne wzorce batch processing

1. Partycjonowanie pracy

Duży zbiór dzielimy na niezależne fragmenty. Każdy fragment przetwarzany osobno. Brak komunikacji między węzłami roboczymi podczas przetwarzania.

2. Odporność przez lineage (nie replikację)

Nie kopiujemy danych pośrednich na inne węzły. Zamiast tego pamiętamy **przepis** (lineage) — ciąg transformacji od inputu. Awaria? Powtórz obliczenie z przepisu.

3. Lazy evaluation + DAG

Transformacje nie wykonują się natychmiast — budują graf (DAG). Optymalizator analizuje cały graf *przed* uruchomieniem. Eliminuje zbędne kroki, scala operacje.

4. Bariera shuffle

Jedyny punkt synchronizacji — wymiana danych między fazami. Zapis na dysk → odczyt przez następną fazę. **Wąskie gardło** każdego systemu batch.

Ewolucja: od MapReduce do Spark

Problem MapReduce (Hadoop, 2006)

Każdy krok Map→Reduce zapisuje wyniki na dysk (HDFS). Algorytmy iteracyjne (ML, PageRank) wymagają wielu przebiegów → wielokrotne I/O.

Rozwiązanie: przetwarzanie in-memory (Spark, 2014)

Wyniki pośrednie w pamięci (RDD → DataFrame). Zapis na dysk tylko przy shuffle. **10–100× szybszy** dla iteracyjnych obliczeń.

Kluczowe innowacje: **Catalyst Optimizer** (predicate pushdown, join reordering), **AQE** (dynamiczna optymalizacja w runtime), **Tungsten** (codegen + off-heap).

Wzorzec ogólny: **deklaratywne API + optymalizator** — użytkownik opisuje cel, system planuje wykonanie.

Wzorzec: Transform-in-Place

Zamiast wyciągać dane, transformować na zewnętrznym silniku i ładować z powrotem — transformuj **wewnątrz** warehouse za pomocą SQL. Warehouse ma już compute i dane.

Co to zmienia architektonicznie:

- **Modularność** — modele SQL z zależnościami (DAG transformacji)
- **Testowalność** — assertions na danych (not_null, unique, własne reguły)
- **Wersjonowanie** — SQL w Git, CI/CD, code review
- **Inkrementalność** — przetwarzaj tylko nowe/zmienione rekordy

Pozycja w Modern Data Stack:

Ingestion (EL) → **Warehouse** (surowe) → **dbt** (Transform) → BI / ML

Przetwarzanie strumieniowe (streaming)

Fundamentalny problem: czas

Event-time vs Processing-time

- **Event-time**: kiedy zdarzenie **naprawdę się wydarzyło** (timestamp z urządzenia/serwisu)
- **Processing-time**: kiedy zdarzenie **dotarło do systemu** przetwarzającego

Dlaczego to problem?

Zdarzenia docierają **nie w kolejności**. Opóźnienia sieci, retransmisje, buffering.

Zdarzenie z 10:00 może dotrzeć o 10:05. Jeśli okno `[10:00–10:05]` już zamknięte — dane utracone.

Konsekwencje architektoniczne:

- Nie możesz przetwarzać „po kolei" — musisz **tolerować nieporządek**
- Musisz zdefiniować: jak długo czekam na spóźnione dane?
- Musisz wybrać: **kompletność** (czekaj dłużej) vs **świeżość** (emituj szybciej)

Watermarks — sygnał kompletności

Watermark = „wszystkie zdarzenia z event-time ≤ T prawdopodobnie dotarły“

Watermarki zamykają okna.

Strategie: **perfekcyjne** albo **heurystyczne**.

Watermark za wcześnie:

- Dane jeszcze docierają
- Dozwolone spóźnienie (**allowed lateness**)

Watermark za późno:

- Wyniki później

Okna czasowe (windowing)

Rodzaje okien — mechanizm grupowania zdarzeń w czasie

- **Tumbling (stałe)** — nieprzekrywające się okna o stałym rozmiarze (np. co 5 min)
- **Sliding (przesuwne)** — nakładające się okna (np. 5 min co 1 min) — jedno zdarzenie w wielu oknach
- **Session (sesyjne)** — dynamiczne, zamykane po okresie nieaktywności (gap)

Typ okna	Rozmiar	Zdarzenie należy do	Zastosowanie
Tumbling	Stały	Dokładnie 1 okna	Raporty co N minut, billing
Sliding	Stały	Wielu okien naraz	Średnia krocząca, alerting
Session	Dynamiczny	1 sesji per klucz	Aktywność użytkownika, clickstream

Late events: **allowed lateness** (okno otwarte dłużej, retraction) lub **side output** (osobny strumień).

Gwarancje dostarczenia — semantyka przetwarzania

Trzy poziomy gwarancji

- **At-most-once**: zdarzenie przetwarzane 0 lub 1 raz. Utrata danych możliwa. Najszybsze.
- **At-least-once**: zdarzenie przetwarzane 1+ razy. Duplikaty możliwe. Wymaga idempotencji konsumenta.
- **Exactly-once**: zdarzenie przetwarzane dokładnie 1 raz. Najdroższe. Wymaga koordynacji.

Exactly-once nie istnieje „za darmo“

Checkpointing, atomowy commit offsetu i 2-Phase Commit. Koszt: większe opóźnienie.

Pragmatyczna rekomendacja

At-least-once + idempotentne odbiorniki = najczęstszy wybór.

Stan w przetwarzaniu strumieniowym

Problem: operacje stanowe (aggregation, join, deduplication)

Streaming to nie tylko `map/filter`. Agregacja wymaga **pamiętania** dotychczasowych wyników. JOIN dwóch strumieni wymaga **buforowania** jednego w oczekiwaniu na drugi.

Wzorzec: Local State + Checkpoint

- Stan lokalny
- Zapis do storage
- Awaria ⟶ odtworzenie + replay

Konsekwencje architektoniczne:

- Stan rośnie ⟶ potrzebne TTL / eviction
- Rebalancing = migracja stanu

Stan to ukryta baza danych

Stan streamingu traktuj jak bazę danych.

Narzędzia streamingowe — przegląd

Narzędzie	Model deploymentu	Silna strona	Ograniczenie
Kafka Streams	Biblioteka	Niskie opóźn.	Kafka
Apache Flink	Klaster	Stan	Złożoność
Spark Streaming	Micro-batch	Batch + stream	Sekundy

Kafka jako centralny szkielet danych

Wzorzec: Log jako źródło prawdy

Z wykładu 3: Kafka = rozproszony log zdarzeń

Teraz Kafka jako **centralny szkielet danych** — nie tylko transport, ale fundament architektury.

Immutable append-only log — fundamentalne właściwości

- **Niezmienność**: zdarzenie zapisane nigdy nie jest modyfikowane (append-only)
- **Uporządkowanie**: zdarzenia w partycji mają gwarantowaną kolejność
- **Replay**: konsument cofa offset do dowolnego punktu ⟶ przetworzenie historii od nowa
- **Wielokrotna konsumpcja**: wiele konsumentów czyta te same dane niezależnie

Architektonicznie Kafka jako szkielet oznacza:

- Każdy serwis **produkuje** zdarzenia do centralnego logu
- Nowy serwis? Podłącz się do istniejących topiców — dane już tam są
- **Decoupling w czasie**: producent i konsument nie muszą działać jednocześnie
- **Jedno źródło prawdy** dla przepływu danych między systemami

Kafka Connect — wzorzec integracji

Wzorzec: Source/Sink Connector

Standaryzowany interfejs integracji z logiem centralnym.

Dlaczego connector framework: konfiguracja zamiast kodu, restart i tracking offsetów, gotowe konektory.

Typowe pipeline'y

- CDC → Kafka → Elasticsearch
- CDC → Kafka → S3
- CDC → Kafka → ClickHouse/Pinot

Lambda vs Kappa Architecture

Lambda Architecture (Nathan Marz, 2011)

Trzy warstwy Lambda

1. **Batch** — historia. + **Speed** — świeże dane. + **Serving** — scala wyniki.

Dlaczego: stream dawał świeżość, batch dokładność.

Problem: dwie bazy kodu muszą dawać identyczne wyniki.

Kappa Architecture (Jay Kreps, 2014)

Jeden pipeline zamiast dwóch

Jeden pipeline streamingowy przetwarza **wszystkie** dane — bieżące i historyczne. Reprzetworzenie = replay logu od początku.

Wymagania:

1. **Długa retencja logu** — dni/miesiące, nie godziny (cały dataset w logu)
2. **Immutable events** — append-only, nigdy nie mutujemy przeszłości
3. **Determinizm** — ten sam input ⟶ ten sam output przy replay
4. **Logika wyrażalna inkrementalnie** — rolling aggregations, counters, session windows

Kiedy Kappa nie wystarczy

- ML training z wieloma epokami
- Globalne agregacje
- Batch różny od streaming

Lambda vs Kappa — decyzja architektoniczna

Aspekt	Lambda	Kappa
Duplikacja	Tak	Nie
Historia	Batch	Replay
Złożoność	Wysoka	Średnia
Logika	Opcj.	Wymagana

Change Data Capture — w praktyce

CDC — przypomnienie i rozwinięcie pipeline'owe

Z wykładu 5 i 7

CDC jako element **pipeline'u danych**.

Wzorzec: Log-based Change Propagation

Czytaj **log transakcji** i publikuj zdarzenia. Konsumenci budują projekcje.

Zastosowania pipeline'owe:

- **Materialized views** — CDC → streaming processor → OLAP (dashboard real-time)
- **Search index sync** — CDC → search engine (pełnotekstowe wyszukiwanie)
- **Cache invalidation** — CDC → usunięcie/aktualizacja w cache
- **Data lake ingestion** — CDC → object storage (Parquet/Avro) → analytics
- **Cross-region replication** — CDC → inna region (disaster recovery)

CDC — architektura i tryby

Trzy modele wdrożenia CDC

1. **Connector** (Kafka Connect)
2. **Standalone server**
3. **Embedded engine**

Outbox Pattern + CDC (dobra praktyka)

Outbox daje atomowość jednej transakcji DB. CDC czyta `outbox` z logu transakcji i publikuje natychmiast.

Porównanie

Dual writes: brak atomowości. Polling: sekundy i duże obciążenie. CDC: atomowość i szybkie.

Analityka czasu rzeczywistego

Wzorce OLAP real-time

Dlaczego zwykła baza relacyjna nie wystarczy?

Analityka = agregacje na milionach/miliardach wierszy. Tradycyjna baza (PostgreSQL) zoptymalizowana pod OLTP (pojedyncze wiersze). OLAP wymaga innej architektury.

1. Magazyn kolumnowy (columnar storage)

Dane kolumnami.

2. Wstępna agregacja (pre-aggregation)

Skraca zapytania kosztem elastyczności.

3. Partycjonowanie czasowe

Dane podzielone na segmenty po czasie.

Wzorce OLAP real-time (cd.)

4. Indeks odwrócony + indeks bitmapowy

- **Indeks odwrócony**: lista wierszy
- **Indeks bitmapowy**: bit per wiersz

5. Rozdzielenie ścieżki zapisu i odczytu

Zapis: szybkie ładowanie. **Odczyt**: szybkie zapytania.

Kiedy który model analityczny?

Konsument	Potrzeba	Model
Analityk biznesowy	Ad-hoc SQL, pełna historia	Data warehouse (BigQuery, Redshift)
Dashboard / BI	Predefiniowane metryki, niskie opóźnienie	Real-time OLAP (ClickHouse, Druid)
Użytkownik końcowy	Wynik w < 100 ms	Pre-agregacja + cache
Data scientist	Pełny skan, ML	Data lake (Parquet, Iceberg)

Nie ma jednego silnika na wszystko

Real-time OLAP to **kompromis**: elastyczność, opóźnienie i koszt ładowania danych. Jeden system nie obsłuży wszystkiego.

ETL vs ELT

Dwa paradygmaty przetwarzania danych

ETL (Extract, Transform, Load)

1. Wyciągnij dane ze źródeł
2. **Przekształć** na oddzielnym silniku
3. Załaduj do warehouse

Era on-premise: compute drogi, storage drogi, SQL wolny na dużą skalę.

ELT (Extract, Load, Transform)

1. Wyciągnij dane ze źródeł
2. Załaduj **surowe** do warehouse
3. **Przekształć** wewnątrz warehouse (SQL)

Era cloud: compute elastyczny, storage tani, SQL = najlepsza transformacja.

Dlaczego ELT wygrało w chmurze

Compute tani, **storage** tani, **SQL** jako transformacja.

Kiedy ETL nadal wygrywa

ETL nie jest martwy

PII / compliance, **Złożone transformacje non-SQL**, **Ekstremalny wolumen**, **Integracje operacyjne**.

Kluczowe pytanie architektoniczne

Gdzie następuje transformacja? Odpowiedź zależy od: kto jest konsumentem, jakie są wymagania compliance, jaki jest wolumen, ile kosztuje compute.

Managed Streaming i OLAP

☁ Przegląd usług chmurowych

Kategoria	AWS	GCP	Azure
Streaming	Kinesis	Pub/Sub	Event Hubs
Kafka	MSK	—	Event Hubs
Processing	Kinesis A.	Dataflow	Stream Analytics
Warehouse	Redshift	BigQuery	Synapse
ETL	Glue	Dataflow	Data Factory
OLAP	Timestream	BigQuery	ADX

Case study: News Feed System

Design a News Feed (A. Xu, rozdz. 11)

Problem

10M DAU, 500M postów/dzień. Wymaganie: szybki feed i aktualizacje.

Fan-out on Write (Push)

Post → cache per user. Odczyt szybki, zapis drogi.

Fan-out on Read (Pull)

Request → merge przy odczycie. Zapis tani, odczyt drogi.

Wzorzec: Hybrid Fan-out

Routing na podstawie asymetrii grafu społecznego

Sieci społeczne są **asymetryczne** — 99% autorów ma < 10K followersów, ale 1% generuje większość wzmocnienia zapisu.

Rozwiązanie: **routing progowy** — poniżej progu Push, powyżej Pull.

Przykłady

Twitter/X i Facebook: hybrid. LinkedIn: fan-out on read.

News Feed — wzorce danych

ID-only cache — oszczędność pamięci

W cache: ID + score. Pełne obiekty pobierane dopiero przy renderowaniu.

Pipeline za News Feed

- **Write path** (streaming): nowy post → fan-out → zapis ID do cache followersów
- **Read path** (batch/on-demand): pobranie pełnych obiektów, ranking, hydration
- **Sync** (CDC): zmiany w profilu/poście propagowane do cache i indeksów

Wasz system e-commerce generuje 50M zdarzeń zamówień dziennie.

Analitycy chcą: dashboardy real-time (ostatnie 5 min) + raporty dzienne (pełna historia).

Lambda czy Kappa? Uzasadnijcie.

Jakie **wzorce** (nie narzędzia!) wybierzecie dla każdej warstwy?

Bonus: skąd dane trafią do analityki — polling, dual writes, czy CDC? Dlaczego?

Podsumowanie

Kluczowe wnioski

1. **Batch**: partycjonuj pracę, odporność przez lineage, optymalizuj DAG
2. **Streaming**: event-time ≠ processing-time ⟶ watermarki i okna
3. **Kafka**: immutable log umożliwia replay, decoupling i wielokrotną konsumpcję
4. **Kappa vs Lambda**: jeden pipeline upraszcza, ale nie zawsze wystarczy
5. **CDC**: log transakcji ⟶ zdarzenia, bez dual writes
6. **OLAP**: kolumny, pre-agregacja, rozdzielenie zapisu i odczytu

Źródła i lektury

- M. Kleppmann — *DDIA*, rozdz. 10 (batch), 11 (stream)
- J. Dean, S. Ghemawat — „MapReduce: Simplified Data Processing on Large Clusters“ (OSDI 2004)
- M. Zaharia et al. — „Resilient Distributed Datasets“ (NSDI 2012)
- T. Akidau — „The world beyond batch: Streaming 101″ (O’Reilly, 2015)
- J. Kreps — „Questioning the Lambda Architecture“ (O’Reilly Radar, 2014)
- J. Kreps — „I Heart Logs“ (O’Reilly, 2014)
- N. Marz, J. Warren — *Big Data* (Manning, 2015)
- A. Xu — *System Design Interview*, rozdz. 11 (News Feed)
- LinkedIn Engineering — „FollowFeed“ (2024)
- Debezium Documentation — debezium.io

Następny wykład

Wykład 9: Serverless, FaaS i architektura na brzegu

Serverless jako model · FaaS vs kontenery · Edge Computing · Koszty · WebAssembly