Dane w systemach rozproszonych

Replikacja · Partycjonowanie · Spójność · CAP/PACELC

mgr inż. Jakub Woźniak

Politechnika Poznańska · Instytut Informatyki · Semestr letni 2025/26

Plan wykładu

Agenda

- Database per Service — konsekwencje projektowe
- Dobór bazy danych pod przypadek użycia
- Replikacja — leader-follower, multi-leader, leaderless
- Partycjonowanie — range, hash, consistent hashing
- Spójność — ACID vs BASE, CAP, PACELC
- Change Data Capture (CDC)
- ☁ Managed databases w chmurze

Database per Service

Zasada: każdy serwis = własne dane

Database per Service

Każdy mikroserwis **jest właścicielem** swojego stanu. Żaden inny serwis nie sięga bezpośrednio do jego bazy — komunikacja wyłącznie przez API lub zdarzenia.

Konsekwencje:

- Brak JOIN-ów między serwisami — denormalizacja lub API composition
- Brak transakcji ACID między bazami — potrzebne sagi (wykład 4)
- Wolność wyboru technologii — serwis A na PostgreSQL, B na Redis
- **Właściciel danych = właściciel schematu** — niezależne migracje

C. Richardson — microservices.io/patterns/data/database-per-service.html

Anti-pattern: Shared Database

Shared Database

Wiele serwisów pisze/czyta z **jednej bazy**. Każda zmiana schematu wymaga koordynacji wszystkich zespołów. Tight coupling przez współdzielone tabele.

Sygnały ostrzegawcze:

- Serwis zamówień robi `SELECT * FROM users` bezpośrednio
- Migracja kolumny wymaga deploymentu 5 serwisów jednocześnie
- „Nie możemy zmienić tabeli, bo ktoś z innego zespołu z niej czyta“

Dobór bazy danych

Nie ma jednej bazy na wszystko

Typ	Przykłady	Silne strony	Przypadek użycia
Relacyjna	PostgreSQL, MySQL	ACID, JOIN-y, dojrzałość	Zamówienia, finanse
Dokumentowa	MongoDB, Firestore	Elastyczny schemat, JSON	Katalog produktów, CMS
Wide-column	Cassandra, HBase	Write-heavy, skala	Logi, IoT, time-series
Klucz-wartość	Redis, DynamoDB	Sub-ms latencja, prostota	Cache, sesje, leaderboard
Grafowa	Neo4j, Neptune	Relacje N:M, traversal	Social graph, fraud detect.

Kleppmann: „*Nie pytaj »jaka baza jest najlepsza?« — pytaj »jaki mam wzorzec dostępu?«*“

Replikacja

Po co replikować?

Trzy cele replikacji

1. **Wysoka dostępność** — awaria jednego węzła nie zatrzymuje systemu
2. **Niskie opóźnienie odczytów** — replika blisko użytkownika (multi-region)
3. **Skalowalność odczytów** — rozkładanie read load na wiele replik

Fundamentalny kompromis (Kleppmann, DDIA rozdz. 5):

Im więcej replik, tym trudniej utrzymać ich spójność.

Leader-Follower (single-leader)

Zasada działania

Jeden węzeł (**leader**) przyjmuje zapisy. Followerzy replikują log asynchronicznie lub synchronicznie.

- **Synchroniczna replikacja**: follower potwierdza przed commit — gwarantuje spójność, ale zwiększa latency
- **Asynchroniczna**: leader commituje natychmiast — ryzyko utraty danych przy awarii leadera
- **Semi-synchroniczna** (PostgreSQL): 1 follower synchroniczny, reszta async — kompromis

Problemy: **failover** — kto zostaje nowym leaderem? Split-brain jeśli poprzedni leader wraca.

Multi-Leader i Leaderless

Multi-Leader

Wielu leaderów w różnych DC. Zapis do najbliższego.

- Niższe opóźnienie zapisu (lokalny DC)
- **Konflikty zapisu** — ten sam rekord zmieniony w dwóch DC
- Rozwiązywanie: last-write-wins, merge, CRDT
- Np. CouchDB, Cassandra (multi-DC)

Leaderless (Dynamo-style)

Brak leadera. Zapis do **N replik** jednocześnie.

- Klient pisze do W replik, czyta z R
- Quorum: **W + R > N** gwarantuje overlap
- Np. Cassandra, Riak, DynamoDB

G. DeCandia et al. — „Dynamo: Amazon's Highly Available Key-value Store" (SOSP 2007)

Quorum: W + R > N

Parametry quorum (N=3)

- **W=2, R=2**: zapis na 2 z 3 replik, odczyt z 2 ⟶ zawsze overlap z najnowszym zapisem
- **W=1, R=3**: szybki zapis, wolny odczyt ⟶ read-repair naprawia niespójne repliki
- **W=3, R=1**: wolny zapis, szybki odczyt ⟶ gwarancja read-after-write

Quorum nie gwarantuje linearizability!

Sloppy quorum (hinted handoff), network partitions, concurrent writes — nawet z W+R>N możliwe odczytanie starych danych. Wicej w “Dynamo: Amazon…”

Kleppmann — DDIA, rozdz. 5: „Quorums for reading and writing“

Partycjonowanie

Po co partycjonować (sharding)?

Cel

Dane nie mieszczą się na jednym węźle → **podziel je** na partycje (shardy) rozłożone na wiele maszyn.

Dwa cele:

- **Skalowalność zapisu** — rozpraszamy write load
- **Skalowalność danych** — dane rozłożone na klaster

Klucz partycjonowania (**partition key**) determinuje, na który węzeł trafi rekord.

Range vs Hash partitioning

Range-based

Klucze posortowane, zakres per partycja.

- **1.** Efektywne range scany
- **Hot spots** — np. klucz = data, dziś = cały ruch na jednej partycji
- Np. HBase, BigTable

Hash-based

Klucz ⟶ hash ⟶ partycja.

- **1.** Równomierny rozkład
- Brak range scanów (hash niszczy kolejność)
- Np. Cassandra (z opcją compound key)

Consistent Hashing (Karger et al., 1997)

Węzły i klucze mapowane na **pierścień**. Klucz trafia do najbliższego węzła w kierunku zgodnym z wskazówkami zegara. Dodanie/usunięcie węzła przesuwa tylko 1/N kluczy.

Użyte w: DynamoDB, Cassandra, Riak, Memcached, load balancerach.

Wariant: **virtual nodes** (vnodes) — każdy fizyczny węzeł ma wiele pozycji na pierścieniu → lepszy balans.

Karger et al. — „Consistent Hashing and Random Trees“ (STOC 1997)

Spójność danych

ACID vs BASE

ACID

Atomicity — all or nothing
Consistency — niezmienniki zachowane
Isolation — transakcje nie widzą siebie nawzajem
Durability — zapisane = trwałe

BASE

Basically Available — system odpowiada (możliwe stare dane)
Soft state — stan może być niespójny tymczasowo
Eventual consistency — po czasie repliki się zsynchronizują (spójność ostateczna)

ACID = gwarancja dla jednej bazy. W systemie rozproszonym (wiele baz) ⟶ BASE + Saga.

CAP Theorem (Brewer, 2000)

CAP

W obliczu **partycji sieciowej** (P) musisz wybrać:
Consistency (C) — każdy odczyt zwraca najnowszy zapis, albo błąd
Availability (A) — każdy odczyt zwraca odpowiedź (niekoniecznie najnowszą)

Typowe nieporozumienia

- CAP nie mówi „wybierz 2 z 3″ — **partycje się zdarzają**, więc wybór to C vs A *podczas partycji*
- Kiedy sieć działa normalnie — możesz mieć i C, i A
- CP: odrzuć zapisy/odczyty przy partycji (np. HBase, Spanner)
- AP: odpowiadaj, ale dane mogą być nieaktualne (np. Cassandra, DynamoDB default)

E. Brewer — „CAP Twelve Years Later: How the 'Rules' Have Changed" (2012)

PACELC — bardziej użyteczny model

PACELC (Abadi, 2012)

Jeśli **Partition** (P): wybierz **A** vs **C**

Else (E) — gdy sieć OK: wybierz **Latency** (L) vs **Consistency** (C)

System	P: A vs C	E: L vs C
DynamoDB	A	L
Cassandra	A	L
Spanner	C	C
PostgreSQL (1n)	C	C
MongoDB	A	C
Cosmos DB	konfigurowalne	konfigurowalne

D. Abadi — „Consistency Tradeoffs in Modern Distributed Database System Design“ (2012)

Transakcje rozproszone

2PC — Two-Phase Commit

Fazy

Faza 1 (Prepare): koordynator pyta uczestników „czy możesz commitować?“

Faza 2 (Commit/Abort): jeśli wszyscy OK → commit, w przeciwnym razie → abort

Problemy 2PC

- **Blokujący** — uczestnicy trzymają locki czekając na decyzję
- **Single point of failure** — awaria koordynatora = zablokowane transakcje
- **Nie skaluje się** — latency rośnie z liczbą uczestników
- **Niewspierany** przez wiele NoSQL i brokerów (Kafka, DynamoDB)

W mikroserwisach: preferuj **Sagi** (wykład 4) zamiast 2PC.

Change Data Capture

CDC — propagacja zmian między serwisami

Change Data Capture

Czytaj **log transakcji** bazy danych (WAL/binlog) i publikuj zmiany jako zdarzenia do Kafki. Inne serwisy konsumują te zdarzenia i aktualizują swoje projekcje.

Dlaczego CDC zamiast dual writes?

- Dual write (zapis do DB + zapis do Kafki) = **brak atomowości** — jedno może się udać, drugie nie
- CDC = **atomowe** — jeśli jest w DB, będzie w Kafce

Narzędzia: **Debezium** (open-source, wspiera PostgreSQL, MySQL, MongoDB), Kafka Connect, AWS DMS

Architektura

PostgreSQL (WAL) → **Debezium Connector** → Kafka topic per tabela → konsumenci (projekcje, cache, search index)

Typowe zastosowania:

- **Materialized views** między serwisami — serwis zamówień emituje zdarzenia, serwis analityki buduje dashboardy
- **Cache invalidation** — zmiana w DB → usunięcie z Redis
- **Search index sync** — zmiana w DB → update Elasticsearch
- **Migracja danych** — streaming z legacy DB do nowego systemu

debezium.io — Debezium Documentation

Case study: Key-Value Store

Od hash mapy do rozproszonego systemu

- **Consistent hashing** — dystrybucja kluczy na klaster
- **Quorum** (W, R, N) — kompromis spójność vs dostępność
- **Vector clocks** — wykrywanie konfliktów w leaderless replication
- **Gossip protocol** — propagacja membership i failure detection
- **Merkle trees** — synchronizacja danych między replikami

Amazon wybrał **spójność ostateczną jako domyślną** — „always-writable“ ważniejsze niż silna spójność.

DeCandia et al. — „Dynamo“ (SOSP 2007) · allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf

Managed databases w chmurze

Managed relational — za i przeciw

Usługa	Architektura	Kluczowa cecha
AWS RDS	Single-AZ lub Multi-AZ failover	6 silników, failover < 60s
AWS Aurora	6-krotna replikacja w 3 AZ, log-based	Do 15 read replik, 5x throughput PostgreSQL
Cloud SQL	Managed MySQL/PostgreSQL na GCP	Automatyczne backupy, repliki cross-region
AlloyDB	PostgreSQL-compatible, columnar engine	4x szybszy niż std. PostgreSQL (wg. Google)
Azure SQL DB	Managed SQL Server	Hyperscale tier, auto-tuning

Managed NoSQL i NewSQL

Usługa	Model	Kluczowa cecha
DynamoDB	Klucz-wartość / dokument	Single-digit ms, global tables, on-demand scaling
Cosmos DB	Multi-model (dokument, graf)	5 poziomów spójności (od silnej do ostatecznej)
Spanner	Relacyjna, globalnie spójna	TrueTime: external consistency + global sharding
CockroachDB	Distributed SQL (PostgreSQL)	Survives AZ/region failure, serializable default
PlanetScale	MySQL (Vitess)	Schema branching jak Git, zero-downtime migrations

Spanner — jak to działa?

GPS + zegary atomowe (**TrueTime**) → znany bound na clock skew → **commit-wait** protocol → globalna linearizability bez locków międzykontynentalnych.

Corbett et al. — „Spanner: Google's Globally-Distributed Database" (OSDI 2012)

Twój system ma 3 repliki bazy danych.
Ustawiasz W=2, R=2 (quorum).

Użytkownik zapisał dane i natychmiast je czyta — czy zawsze zobaczy swój zapis?

A jeśli W=1, R=3?

Narysuj scenariusz, w którym quorum nie gwarantuje odczytu swoich zapisów.

Podsumowanie

Kluczowe wnioski

1. **Database per Service** — niezależność kosztem JOIN-ów i transakcji
2. **Replikacja**: leader-follower (proste) vs leaderless (quorum, Dynamo-style)
3. **Partycjonowanie**: consistent hashing = standard w rozproszonych systemach
4. **CAP to nie „wybierz 2 z 3″** — PACELC lepiej opisuje codzienny kompromis L vs C
5. **CDC > dual writes** — atomowa propagacja zmian między serwisami
6. **Managed DB** — Aurora, Spanner, Cosmos DB to nie „hostuję Postgresa na VM“

Źródła i lektury

- M. Kleppmann — *DDIA*, rozdz. 5, 6, 7, 9
- G. DeCandia et al. — *Dynamo* (SOSP 2007)
- J. Corbett et al. — *Spanner* (OSDI 2012)
- E. Brewer — *CAP Twelve Years Later* (2012)
- D. Abadi — *Consistency Tradeoffs…* (2012) — PACELC
- W. Vogels — *Eventually Consistent* (2008)
- A. Xu — *System Design Interview*, rozdz. 6
- Jepsen.io — Kyle Kingsbury — testy spójności baz danych
- debezium.io — Debezium Documentation

Następny wykład

Wykład 6: Odporność i niezawodność — projektowanie na awarie

Circuit Breaker · Retry · Rate Limiting · Chaos Engineering