

**Politechnika Rzeszowska im. Ignacego Łukasiewicza**

**Wydział Matematyki i Fizyki Stosowanej**

## **USŁUGI SIECIOWE W BINZESIE**

---

### **PRACA PROJEKTOWA**

**”System przetwarzania strumieniowego i porównanie PostgreSQL z Cassandra”**

---

Pracę wykonał student: **Karol Szeląg,**  
**Inżynieria i Analiza Danych, gr. nr 06, 2024/25-Z**

Pracę sprawdził: dr inż. Mariusz Borkowski prof. PRz

**Rzeszów, 22.05.2025**

# Spis treści

1.	Wprowadzenie .....	3
1.1.	Cel projektu.....	3
1.2.	Narzędzia i Technologie .....	3
	Apache Kafka.....	3
	Apache Spark (Structured Streaming) .....	3
	PostgreSQL .....	4
	Apache Cassandra .....	4
	Docker.....	4
1.3.	Dlaczego temat jest istotny? .....	4
2.	Architektura systemu .....	5
2.1.	Przepływ danych.....	5
2.2.	Rola poszczególnych komponentów.....	6
3.	Przebieg przetwarzania danych.....	8
3.1.	Skąd pochodzą dane? .....	8
3.2.	Jak trafiają do Kafki? .....	8
3.3.	Jak Spark je przetwarza?.....	8
3.4.	Jak dane są zapisywane do baz? .....	9
4.	Porównanie PostgreSQL vs Cassandra .....	10
4.1.	Model danych.....	10
4.2.	Operacje zapisu i odczytu .....	11
4.3.	Skalowalność i dostępność.....	13
4.4.	Ograniczenia i dobre praktyki.....	13
4.5.	Kiedy używać której bazy? .....	15
5.	Eksperymenty i wyniki .....	16
5.1.	Zakres testów .....	16
5.2.	Wyniki testów .....	17
5.3.	Wnioski .....	18
6.	Wnioski .....	19
6.1.	Która baza sprawdziła się lepiej?.....	19
6.2.	Rekomendacje do przyszłych zastosowań .....	19

# **1. Wprowadzenie**

---

## **1.1. Cel projektu**

---

Celem projektu jest zaprojektowanie i implementacja wydajnego, skalowalnego systemu przetwarzania danych w czasie rzeczywistym z wykorzystaniem nowoczesnych technologii Big Data, takich jak **Apache Kafka**, **Apache Spark** oraz **Docker**. Kluczowym aspektem projektu jest również przeprowadzenie praktycznej analizy porównawczej dwóch popularnych typów baz danych, **relacyjnej (PostgreSQL)** oraz **rozproszonej bazy NoSQL (Apache Cassandra)**, pod kątem ich przydatności w kontekście przetwarzania strumieni danych.

Projekt zakłada:

- **Zbudowanie kompletnego pipeline'u danych** — od pozyskiwania danych, przez ich przetwarzanie w czasie rzeczywistym, aż po zapis do różnych systemów bazodanowych.
- **Porównanie wydajności baz danych** w zakresie operacji zapisu, odczytu, obsługi zapytań oraz odporności na awarie.
- **Identyfikację przypadków użycia**, w których konkretna technologia bazodanowa sprawdza się lepiej, z uwzględnieniem charakterystyki danych, wymagań biznesowych i technicznych ograniczeń.

Efektem końcowym projektu jest nie tylko działający system do przetwarzania danych w czasie rzeczywistym, ale również zbiór praktycznych wniosków wspierających podejmowanie decyzji architektonicznych w systemach Big Data.

## **1.2. Narzędzia i Technologie**

---

### **Apache Kafka**

Rozproszona platforma do przesyłania strumieni danych w czasie rzeczywistym. Umożliwia niezawodne kolejkowanie, buforowanie i dystrybucję danych między komponentami systemu. W projekcie Kafka pełni rolę **systemu ingestującego dane (pobieranie)** i zapewnia **asynchroniczną komunikację** między źródłem danych a modułem przetwarzania.

---

### **Apache Zookeeper**

System koordynacji rozproszonych usług, wykorzystywany przez Apache Kafka do zarządzania metadanymi klastra, kontrolą dostępu, rozpoznawaniem awarii brokerów i utrzymywaniem informacji o podziale partycji. W projekcie pełni funkcję **nadzorcy klastra Kafka (jednego brokera)**, zapewniając jego spójność i niezawodność działania.

---

### **Apache Spark (Structured Streaming)**

Silnik do szybkiego przetwarzania danych, działający zarówno w trybie wsadowym, jak i strumieniowym. W projekcie wykorzystywany do **przetwarzania i transformacji danych w czasie rzeczywistym**, przy użyciu API Structured Streaming, co umożliwia pisanie zapytań przypominających SQL na strumieniach danych.

## **PostgreSQL**

Zaawansowany relacyjny system zarządzania bazą danych (RDBMS), znany z silnego modelu transakcyjnego, wsparcia dla złożonych zapytań i relacji między tabelami. W projekcie służy jako **bazodanowe repozytorium danych o wysokiej spójności**, umożliwiające wykonywanie analitycznych zapytań typu ad hoc.

---

## **Apache Cassandra**

Rozproszona baza danych typu NoSQL oparta na modelu kolumnowym. Zapewnia **wysoką dostępność, odporność na awarie i skalowalność poziomą**. W projekcie wykorzystywana jako **alternatywne rozwiązanie do przechowywania danych**, optymalizowane pod kątem bardzo szybkich operacji zapisu i odczytu przy dużym wolumenie danych.

---

## **Docker**

Platforma do konteneryzacji aplikacji, umożliwiająca uruchamianie wszystkich komponentów systemu w izolowanych, przenośnych środowiskach. Ułatwia **zarządzanie środowiskiem uruchomieniowym**, automatyzuje procesy uruchamiania i pozwala szybko odtworzyć cały pipeline na różnych maszynach.

### ***1.3. Dlaczego temat jest istotny?***

---

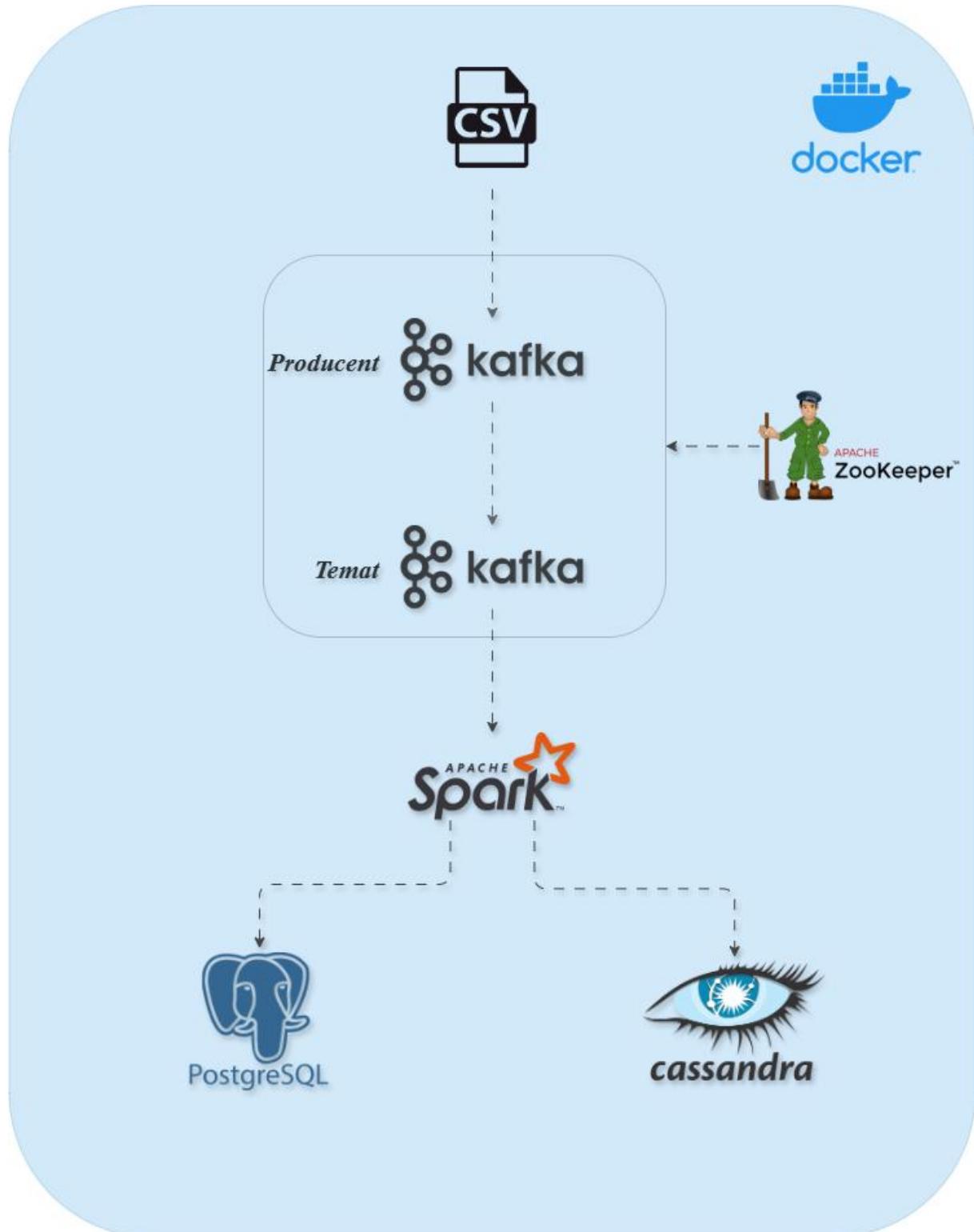
W dobie **dynamicznego wzrostu ilości danych (Big Data)** oraz rosnącej potrzeby **natychmiastowego reagowania na zdarzenia** w systemach informatycznych, kluczowe staje się wykorzystanie technologii umożliwiających **przetwarzanie danych w czasie rzeczywistym**. Firmy z różnych sektorów, od e-commerce i finansów, po IoT i cybersecurity, potrzebują systemów, które potrafią szybko analizować dane i reagować bez opóźnień.

Jednocześnie rozwój **systemów rozproszonych** wymusza nowe podejścia do przechowywania danych. Tradycyjne bazy relacyjne (RDBMS), mimo swojej dojrzałości, nie zawsze są w stanie sprostać wymaganiom wysokiej dostępności i skalowalności. Z drugiej strony, bazy NoSQL takie jak Cassandra oferują elastyczność i wydajność, ale kosztem spójności czy złożoności zapytań.

W tym kontekście projekt ukierunkowany na stworzenie real-time pipeline'u danych oraz porównanie dwóch skrajnie różnych podejść do przechowywania danych (relacyjnego i NoSQL) dostarcza **praktycznych odpowiedzi na współczesne wyzwania inżynierii danych**. Pomaga lepiej zrozumieć, kiedy warto stosować konkretne rozwiązania i jak je skutecznie integrować w skalowalnych architekturach danych.

## 2. Architektura systemu

### 2.1. Przepływ danych



## **2.2. Rola poszczególnych komponentów**

---

### **Apache Kafka – Streaming danych**

Kafka pełni rolę **systemu pośredniczącego** w transmisji danych. Odpowiada za **przesyłanie danych strumieniowych** między komponentami systemu – odczytuje dane z pliku CSV za pomocą producenta i publikuje je do tzw. tematów (topics), skąd mogą być odczytywane przez konsumentów, np. Spark. Zapewnia **odporność na awarie i skalowalność**, co czyni go fundamentem niezawodnego przepływu danych.

---

### **Apache Zookeeper – Koordynacja klastra**

Zookeeper pełni funkcję koordynatora i menedżera stanu w środowiskach rozproszonych. W kontekście projektu jest niezbędny do poprawnego działania Apache Kafka, odpowiadając za zarządzanie metadanymi klastra, synchronizację brokerów, wybór liderów partycji oraz kontrolę dostępności tematów. Dzięki Zookeeperowi Kafka może działać w trybie wysokiej dostępności i zapewniać spójność działania w sytuacjach awaryjnych.

---

### **Apache Spark – Przetwarzanie w czasie rzeczywistym**

Spark Structured Streaming działa jako **moduł przetwarzający dane "on-the-fly"**, które odbiera z Kafki. Umożliwia transformacje, filtrowanie i wstępna analizę danych w locie. Spark odgrywa kluczową rolę w przekształcaniu surowych danych na uporządkowaną postać gotową do dalszej analizy lub zapisu do bazy danych.

---

### **Docker – Konteneryzacja i izolacja środowisk**

Docker umożliwia **uruchamianie wszystkich komponentów systemu w odizolowanych kontenerach**, co znacznie upraszcza zarządzanie środowiskiem, konfiguracją oraz wdrożeniami. Dzięki Dockerowi cały pipeline może zostać łatwo uruchomiony na różnych maszynach bez konfliktów środowiskowych, co poprawia **powtarzalność i mobilność rozwiązania**.

---

### **PostgreSQL i Apache Cassandra – Zapis i analiza danych**

Obie bazy danych służą jako **docelowe systemy przechowywania danych**, ale mają różne cechy:

- **PostgreSQL** – relacyjna baza danych, idealna do **złożonych zapytań analitycznych**, operacji JOIN i zachowania silnej spójności danych.
- **Cassandra** – rozproszona baza NoSQL zoptymalizowana pod kątem **wysokowydajnego zapisu i odczytu**, skalowalności poziomej i odporności na awarie.

Ich równoległe użycie pozwala na **porównanie zalet i ograniczeń** obu podejść w kontekście przetwarzania strumieniowego.

## **2.3. Docker-Compose file**

---

Plik docker-compose.yml definiuje środowisko kontenerowe złożone z kilku współpracujących ze sobą usług, umożliwiające lokalne testowanie oraz analizę danych z wykorzystaniem takich technologii jak Apache Kafka, Apache Cassandra, PostgreSQL i Apache Spark wraz z JupyterLab.

Środowisko składa się z pięciu głównych usług:

1. **pyspark-jupyter** – kontener oparty na obrazie easewithdata/pyspark-jupyter-lab, zawierający zintegrowane środowisko JupyterLab oraz PySpark. Kontener ten nasłuchuje na portach 8888 (interfejs Jupyter) i 4040 (Spark UI). Współdzieli dane z innymi kontenerami za pomocą wolumenu spark\_data. Uruchamiany jest po starcie usług Kafka, Cassandra i PostgreSQL, co zapewnia poprawną kolejność inicjalizacji zależnych komponentów.
2. **zookeeper** – kontener z Apache Zookeeper, niezbędny do działania Kafki, zarządzający konfiguracją klastra Kafka. Udostępnia port 2181, który służy do komunikacji z usługą Kafka. Parametry środowiskowe konfigurowane są tak, by zapewnić podstawową synchronizację i działanie w środowisku lokalnym.
3. **kafka** – kontener uruchamiający Apache Kafka, który zależy od wcześniej zdefiniowanego Zookeepera. Umożliwia przesyłanie strumieni danych między systemami, co jest szczególnie przydatne w analizach czasu rzeczywistego. Kafka nasłuchuje na porcie 9092 oraz współdzieli dane poprzez spark\_data. Konfiguracja brokera obejmuje m.in. identyfikator brokera, ścieżkę do Zookeepera, konfigurację protokołów komunikacyjnych oraz automatyczne utworzenie tematu raw z jedną partycją i jedną repliką.
4. **cassandra** – kontener z bazą danych NoSQL Apache Cassandra, przeznaczoną do przechowywania dużych wolumenów danych w sposób rozproszony. Usługa nasłuchuje na porcie 9042 i działa w ramach klastra nazwanego KafkaCassandraCluster. Ustawienia dotyczą również konfiguracji logicznej infrastruktury (DC i RACK), co może mieć znaczenie w przypadku skalowania klastra.
5. **postgres** – klasyczna relacyjna baza danych PostgreSQL, udostępniająca port 5432. Konfiguracja obejmuje stworzenie użytkownika, bazy danych i hasła, co umożliwia szybkie rozpoczęcie pracy z bazą.

Na końcu pliku znajduje się definicja współdzielonego wolumenu spark\_data, który służy do trwałego przechowywania danych lub logów pomiędzy kontenerami. Dzięki jego zastosowaniu można łatwiej integrować dane przetwarzane przez różne komponenty systemu.

## **3. Przebieg przetwarzania danych**

---

### **3.1. Skąd pochodzą dane?**

---

Na potrzeby projektu dane nie pochodzą z rzeczywistego systemu produkcyjnego, lecz są **generowane syntetycznie** przy pomocy dedykowanego skryptu napisanego w języku **Python**. Celem było stworzenie dużej, realistycznej próbki danych umożliwiającej testy wydajnościowe i funkcjonalne całego pipeline'u.

Przy **każdym uruchomieniu skryptu generowane jest ok. 1 milion rekordów**, co pozwala dokładnie przetestować działanie systemu w warunkach zbliżonych do realnych (wysoka objętość danych i ciągły napływ nowych wpisów).

Dane są zapisywane do **lokalnego pliku CSV**, który pełni rolę źródła wejściowego dla dalszego etapu – przesyłania danych do Apache Kafka. (plik Random\_Logs.py)

### **3.2. Jak trafiają do Kafki?**

---

Po wygenerowaniu i zapisaniu dużego zbioru danych w pliku CSV, kolejnym krokiem jest ich przesłanie do systemu **Apache Kafka**, który pełni rolę **streamingowego pośrednika**. Proces ten realizowany jest za pomocą specjalnego **skryptu producenta (Kafka Producer)**, który został przygotowany w Pythonie. Skrypt ten:

1. **Odczytuje dane z pliku CSV** linię po linii, wczytując kolejne rekordy zawierające m.in. timestamp, wartości liczbowe oraz atrybuty tekstowe.
2. **Serializuje rekordy** do formatu odpowiedniego dla przesyłu (np. JSON lub prosty tekst).
3. **Publikuje rekordy do wybranego tematu (topic) w Apache Kafka**, gdzie każdy rekord jest zapisywany jako pojedyncza wiadomość.

Dzięki temu Kafka otrzymuje dane jako strumień wiadomości, które mogą być następnie konsumowane przez kolejne komponenty systemu, takie jak Apache Spark.

Ten model przesyłu zapewnia:

- **Asynchroniczność** – producent wysyła dane bez blokowania dalszych operacji,
- **Buforowanie i odporność na awarie** – Kafka przechowuje dane, dopóki konsumenci ich nie odbiorą,
- **Skalowalność** – łatwe dodawanie kolejnych producentów i konsumentów.

### **3.3. Jak Spark je przetwarza?**

---

Apache Spark, korzystając z mechanizmu **Structured Streaming**, działa jako konsument danych przesyłanych przez Apache Kafka. Główne zadania Spark'a w tym etapie to:

1. **Odbiór danych w czasie rzeczywistym**  
Spark subskrybuje odpowiedni temat (topic) w Kafka i odbiera strumień wiadomości zawierających rekordy danych.
2. **Deserializacja i parsowanie**  
Otrzymane dane (np. w formacie JSON lub CSV) są parsowane do struktur danych Spark (DataFrame), co umożliwia dalszą analizę i transformacje.

### 3. Transformacje i agregacje

Spark wykonuje różnorodne operacje na strumieniu danych, takie jak filtrowanie, mapowanie, agregowanie czy wzbogacanie danych. Dzięki temu możliwe jest przygotowanie wartości gotowych do zapisu lub analizy.

### 4. Zapis danych do baz

Po przetworzeniu dane są zapisywane do docelowych baz danych: PostgreSQL lub Cassandra. W zależności od konfiguracji, Spark może równolegle kierować dane do obu systemów lub do jednego z nich.

Dzięki zastosowaniu Spark Structured Streaming, cały proces jest **skalowalny, nadaje się do przetwarzania dużych wolumenów danych** i zapewnia **niski czas opóźnienia** między pojawieniem się danych w Kafka a ich zapisaniem w bazach.

## ***3.4. Jak dane są zapisywane do baz?***

---

W projekcie dane po przetworzeniu przez Apache Spark są **zapisywane do dwóch różnych systemów bazodanowych** - PostgreSQL oraz Apache Cassandra. Każda z tych baz pełni inną rolę i wymaga osobnego podejścia do zapisu danych. Dlatego też implementacja zapisu została podzielona na dwa odrębne skrypty:

### **1. Skrypt zapisu do PostgreSQL**

- Ten skrypt wykorzystuje konektor JDBC, który umożliwia bezpośrednią komunikację Spark z relacyjną bazą PostgreSQL.
  - Dane są zapisywane w postaci uporządkowanych tabel zgodnie ze schematem relacyjnym.
  - PostgreSQL zapewnia silną spójność i wsparcie dla złożonych zapytań SQL, co jest szczególnie przydatne przy analizie danych.
  - Skrypt obsługuje potencjalne konflikty (np. duplikaty kluczy głównych) zgodnie z zasadami modelu relacyjnego.
  - Warto zauważyć, że połączenie z **PostgreSQL okazało się znacznie prostsze** do skonfigurowania – wystarczyło wskazać parametry JDBC (adres, użytkownik, hasło i nazwę bazy), a zapis działał od razu bez dodatkowych komplikacji.
- 

### **2. Skrypt zapisu do Apache Cassandra**

- Do zapisu wykorzystywany jest natywny konektor Spark-Cassandra, który jest zoptymalizowany pod kątem wysokowydajnych operacji na danych NoSQL.
- Cassandra przyjmuje dane w formie denormalizowanej, zoptymalizowanej pod kątem szybkich odczytów i skalowalności poziomej.
- Skrypt umożliwia szybkie wstawianie dużych wolumenów danych, z zachowaniem odporności na awarie i rozproszenie danych na wiele węzłów klastra.
- Brak silnych zależności i wymagań dotyczących spójności pozwala na bardzo efektywne przetwarzanie dużych strumieni danych.

Podsumowując, zastosowanie **dowóch osobnych skryptów** pozwala na niezależne testowanie oraz zoptymalizację procesu zapisu do obu baz i na porównanie ich zachowania pod kątem wydajności, spójności i skalowalności. Dodatkowo **prostsza konfiguracja połączenia z PostgreSQL** pozwoliła na szybsze rozpoczęcie testów i eksperymentów analitycznych.

## **4. Porównanie PostgreSQL vs Cassandra**

---

### **4.1. Model danych**

---

#### **PostgreSQL**

PostgreSQL to **relacyjna baza danych**, oparta na dobrze znanym modelu tabelarycznym. Dane są przechowywane w formie tabel z jasno zdefiniowanymi **relacjami** między nimi, co pozwala na stosowanie:

- **Złożonych zapytań SQL** - w tym łączeń (JOIN), agregacji, podzapytań i operacji analitycznych.
- **Transakcji** - zapewniających silną spójność i integralność danych dzięki mechanizmowi ACID (Atomicity, Consistency, Isolation, Durability).
- **Normalizacji danych** — co minimalizuje redundancję i umożliwia logiczne uporządkowanie informacji.

Model ten jest idealny do zastosowań, gdzie ważna jest dokładność danych, relacje między encjami oraz skomplikowane zapytania analityczne.

---

#### **Apache Cassandra**

Cassandra to rozproszona baza danych typu NoSQL, oparta na modelu kolumnowym. Dane są przechowywane w tabelach, jednak w przeciwieństwie do relacyjnych baz danych, nie ma tu tradycyjnych relacji ani rygorystycznych ograniczeń spójności. Zamiast tego, Cassandra stawia na:

- **Bardzo szybkie operacje zapisu i odczytu** – zoptymalizowane pod kątem dużych wolumenów danych oraz pracy w systemach rozproszonych.
- **Skalowalność poziomą** – możliwość łatwego dodawania kolejnych węzłów do klastra bez przestojów i z automatycznym rozkładem danych.
- **Wysoką dostępność** – dzięki replikacji danych i odporności na awarie pojedynczych węzłów.
- **Projektowanie schematu „pod zapytania”** – struktura danych musi być przemyślana z góry, a klucze partycji i klastry odgrywają kluczową rolę w wydajności.
- **Denormalizację** – dane są często powielane i przechowywane w sposób dostosowany do konkretnych przypadków użycia, co zwiększa szybkość kosztem większego zużycia przestrzeni.

Model Cassandry jest idealny do zastosowań, w których liczy się maksymalna wydajność zapisu i odczytu, ciągła dostępność oraz możliwość łatwego skalowania w poziomie, nawet przy bardzo dużej ilości danych.

## **4.2. Operacje zapisu i odczytu**

---

### **Szybkość zapisu**

Testy wydajności wykazały, że **Apache Cassandra przewyższa PostgreSQL pod względem szybkości zapisu**. Cassandra została zaprojektowana jako baza zoptymalizowana pod kątem bardzo szybkich operacji zapisu, szczególnie przy dużych wolumenach danych i rozproszonym środowisku. Dzięki prostemu modelowi danych i mechanizmowi zapisu bez potrzeby utrzymywania relacji i spójności transakcyjnej, Cassandra może przetwarzać dane w sposób bardzo efektywny.

W moim projekcie zaobserwowałem, że zapis około miliona rekordów do obu baz różnił się nieznacznie - czas zapisu do Cassandry był krótszy o około **30 sekund** w porównaniu do PostgreSQL. Choć różnica ta nie jest dramatyczna, w profesjonalnych środowiskach, gdzie przetwarzane są setki milionów lub miliardy rekordów, każda optymalizacja czasowa może przekładać się na znaczące oszczędności zasobów i kosztów.

Dla wielu zastosowań różnica na poziomie kilkudziesięciu sekund może być akceptowalna, jednak w systemach wymagających ultra-niskich opóźnień i bardzo wysokiej przepustowości - zwłaszcza przy intensywnych operacjach zapisu - przewaga Cassandra w szybkości może okazać się kluczowa.

Cassandra:

```
gType,true),StructField(pid,IntegerType,true),StructField(level,  
StructField(content,StringType,true)),org.apache.spark.SparkConf  
Batch 1600: zapisano 502 rekordów (łącznie: 1000000)  
Zapisano 1000000 rekordów w 236.09 sekund.  
25/05/13 16:28:00 INFO DAGScheduler: Asked to cancel job group 0  
25/05/13 16:28:00 INFO ConsumerCoordinator: [Consumer clientId=c  
eff6d6a-3277-18005-driver-0-1, groupId=spark-kafka-source-549e14f
```

PostgreSQL:

```
25/05/13 16:37:38 INFO TaskSchedulerImpl: Killing all running tasks  
25/05/13 16:37:38 INFO DAGScheduler: Job 3880 finished: start at 1  
Batch 1940: zapisano 14 rekordów (łącznie: 1000000)  
Zapisano 1000000 rekordów do PostgreSQL w 266.61 sekund.  
25/05/13 16:37:38 INFO DAGScheduler: Asked to cancel job group 7f  
25/05/13 16:37:38 INFO ConsumerCoordinator: [Consumer clientId=co  
8621058-291791319-driver-0-1, groupId=spark-kafka-source-b91588ad
```

### **Złożoność zapytań i ich wydajność**

Podczas testów złożonych zapytań, które wymagały filtrowania po wielu kolumnach i warunkach, zauważylem, że zarówno PostgreSQL, jak i Cassandra potrzebowali podobnego czasu na wykonanie zapytań, bez wyraźnej przewagi jednej z baz.

Jednakże w przypadku Cassandry, z racji jej rozproszonej architektury, zapytania często wymagają przeszukania wielu węzłów klastra, co może wpływać na wydajność i wprowadzać dodatkowe opóźnienia - zwłaszcza gdy zapytanie korzysta z ALLOW FILTERING, które wymusza skanowanie danych spoza klucza partycji.

Warto podkreślić, że w Cassandra stosowanie ALLOW FILTERING jest zazwyczaj niezalecane na produkcji, gdyż może prowadzić do pełnych skanów danych i obniżenia wydajności. Z kolei PostgreSQL, dzięki możliwościom indeksowania i optymalizacji zapytań SQL, radzi sobie lepiej z tego typu wielowymiarowymi filtracjami.

Podsumowując, choć w testach złożone zapytania miały podobny czas wykonania, w praktyce PostgreSQL oferuje większą elastyczność i wydajność przy skomplikowanych analizach, natomiast Cassandra jest bardziej efektywna w prostych, kluczowych operacjach odczytu i zapisu.

---

### **Co to jest ALLOW FILTERING?**

W Apache Cassandra, aby zapytanie było wydajne, musi korzystać z **klucza partycji** (partition key) lub kluczy klastrowych, które określają, na których węzłach w klastrze znajdują się dane. Cassandra bardzo szybko odnajduje dane, jeśli zapytanie precyzyjnie wskazuje klucz partycji.

**ALLOW FILTERING** to opcja, która pozwala na wykonanie zapytania, które **nie korzysta bezpośrednio z klucza partycji lub kluczy klastrowych**, czyli wymaga przeszukania większej części danych lub nawet całego klastra.

Dzięki temu można wykonać bardziej elastyczne zapytania, ale kosztem **znacznie niższej wydajności**, bo Cassandra musi filtrować dane już po odczycie ich z wielu węzłów.

### **Jak działają klucze partycji?**

Klucz partycji to zestaw kolumn, które definiują, jak dane są **rozproszone i przechowywane na węzłach klastra**. Cassandra korzysta z klucza partycji do:

- **Określenia, na którym węźle znajduje się dany wiersz danych** (poprzez haszowanie wartości klucza partycji).
- **Szybkiego dostępu do danych**, ponieważ Cassandra od razu wie, gdzie szukać. Jeśli zapytanie precyzyjnie podaje klucz partycji, Cassandra może skierować je do jednego lub kilku konkretnych węzłów, co znacznie zwiększa wydajność.  
Jeśli natomiast klucz partycji nie jest użyty, zapytanie wymaga przeszukania całego klastra (full scan), co jest kosztowne i wolne.

## ***4.3. Skalowalność i dostępność***

---

### **Apache Cassandra**

Cassandra została zaprojektowana z myślą o **wysokiej dostępności i odporności na awarie** w środowiskach rozproszonych. Jej architektura opiera się na **klastrze wielowęzłowym**, gdzie dane są automatycznie replikowane między węzłami. W przypadku awarii pojedynczego węzła, pozostałe węzły przejmują jego zadania, co zapewnia ciągłość działania systemu bez przestojów.

**Proces testowania odporności na awarie zwykle polega na odłączeniu jednego z węzłów klastra i obserwacji, czy zapisy i odczyty danych nadal przebiegają poprawnie - bez utraty spójności i dostępności.** Ze względu na problemy z konfiguracją klastra Cassandra w moim projekcie, test ten nie został przeprowadzony, jednak standardowo jest to kluczowy element weryfikacji odporności systemu.

Cassandra realizuje model spójności, który można konfigurować, co pozwala na balansowanie między szybkością a silną spójnością danych.

### **PostgreSQL**

PostgreSQL to tradycyjna baza relacyjna, która zapewnia **silną spójność danych** dzięki transakcjom ACID. Jednakże w swojej podstawowej konfiguracji PostgreSQL działa jako pojedynczy serwer (master), co oznacza, że **bez dodatkowej konfiguracji replikacji i klastrów ma ograniczoną odporność na awarie**.

Aby zwiększyć dostępność i skalowalność, PostgreSQL wymaga konfiguracji rozwiązań takich jak replikacja synchroniczna/asynchroniczna, czy klastry wysokiej dostępności. Bez tego, awaria serwera podstawowego skutkuje przerwą w dostępie do bazy.

### **Podsumowanie**

- **Cassandra** cechuje się natywną odpornością na awarie i skalowalnością poziomą (dodawanie kolejnych węzłów), co czyni ją idealnym rozwiązaniem dla systemów rozproszonych i aplikacji wymagających wysokiej dostępności.
- **PostgreSQL** zapewnia silną spójność i wsparcie dla skomplikowanych transakcji, ale wymaga dodatkowej konfiguracji, by osiągnąć podobny poziom odporności i skalowalności.

## ***4.4. Ograniczenia i dobre praktyki***

---

### **Apache Cassandra**

Cassandra oferuje wysoką wydajność i skalowalność, ale **nie jest bazą uniwersalną** — wymaga specyficznego podejścia do projektowania danych i zapytań.

#### **Ograniczenia:**

- **Brak elastyczności w zapytaniach ad hoc** – Cassandra nie wspiera złożonych zapytań typu JOIN, GROUP BY czy podzapytań. Każde zapytanie musi być **zaplanowane wcześniej**, a struktura danych zaprojektowana z myślą o konkretnych przypadkach użycia.
- **Wrażliwość na nieprzemysłyany schemat danych** – błędne zaprojektowanie kluczy partycji może prowadzić do tzw. hotspota (czyli nierównomiernego obciążenia węzłów) i znacznego spadku wydajności.

- **Potrzeba indeksowania z rozwagą** – wtórne indeksy mają ograniczoną wydajność i mogą spowolnić zapytania, jeśli nie są dobrze dopasowane do wzorców odczytu.

**Zapytania ad hoc - nieplanowane, spontaniczne zapytania** tworzone na bieżąco, zwykle do szybkiej analizy lub sprawdzenia danych.

#### Dobre praktyki:

- Projektuj schemat **z perspektywy zapytań**, a nie normalizacji.
- Unikaj ALLOW FILTERING w środowiskach produkcyjnych.
- Utrzymuj równomierne rozłożenie danych po kluczu partycji (shardowanie).
- Monitoruj replikację i liczbę węzłów przy zmianach obciążenia.

### PostgreSQL

PostgreSQL to zaawansowana baza relacyjna, która doskonale radzi sobie z transakcjami i złożonymi zapytaniami, jednak ma swoje ograniczenia w kontekście wysokiej skali i systemów rozproszonych.

#### Ograniczenia:

- **Nieprzystosowany do milionów równoległych insertów** w środowiskach o bardzo dużym ruchu. Choć PostgreSQL jest bardzo wydajny, przy braku odpowiedniej architektury (np. sharding, partycjonowanie, replikacja) może stać się wąskim gardłem w systemach czasu rzeczywistego.
- **Trudności w poziomej skalowalności** – w przeciwieństwie do Cassandra, PostgreSQL nie jest natywnie przystosowany do poziomego rozproszenia danych. Skalowanie wymaga dodatkowych warstw, takich jak np. Citus, Patroni lub inne rozwiązania klastrowe.

#### Dobre praktyki:

- Korzystaj z indeksów i partycjonowania dla dużych wolumenów danych.
- Optymalizuj zapytania za pomocą EXPLAIN i ANALYZE.
- Przy dużej liczbie insertów – używaj batch insertów oraz asynchronicznego przetwarzania.
- W przypadku wysokiej dostępności – wdrażaj mechanizmy replikacji i automatycznego failovera.

**EXPLAIN** — pokazuje plan wykonania zapytania (jak baza zamierza je przetworzyć).

**ANALYZE** — wykonuje zapytanie i podaje rzeczywiste statystyki jego działania (czas, liczba wierszy).

## 4.5. Kiedy używać której bazy?

Wybór odpowiedniej bazy danych zależy od konkretnych wymagań systemu, charakterystyki danych oraz oczekiwania względem dostępności, skalowalności i spójności. Zarówno **PostgreSQL**, jak i **Cassandra** mają swoje mocne i słabe strony, które czynią je odpowiednimi dla różnych scenariuszy.

Poniższa tabela przedstawia uproszczone porównanie:

Kryterium	PostgreSQL	Cassandra
Złożone zapytania (JOIN, GROUP BY)	✓	✗
Wysoka wydajność zapisu	⚠️ (średnia)	✓
Praca z denormalizacją danych	✗	✓
Silna spójność (ACID)	✓	⚠️ (zależna od ustawień)
Odporność na awarie	⚠️ (z replikacją)	✓
Skalowalność pozioma (dodawanie węzłów)	✗	✓
Elastyczność zapytań ad hoc	✓	✗
Przetwarzanie transakcji	✓	✗
Niskie opóźnienia odczytu (lookup)	⚠️	✓
Wsparcie dla struktur relacyjnych	✓	✗

### Interpretacja

- **PostgreSQL** to dobry wybór dla systemów, które wymagają silnej spójności, bogatych relacji między danymi oraz złożonych zapytań (np. raportowanie, systemy finansowe, klasyczne aplikacje webowe).

- **Cassandra** sprawdzi się lepiej w środowiskach rozproszonych, gdzie kluczowe są wysoka dostępność, duża skalowalność i szybki zapis danych (np. logi, IoT, dane telemetryczne, systemy działające 24/7 bez przestojów).
- W praktyce często stosuje się **hybrydowe podejście** — PostgreSQL do analizy i przetwarzania danych, a Cassandra do szybkiego gromadzenia i przechowywania danych w czasie rzeczywistym.

## **5. Eksperymenty i wyniki**

---

Celem eksperymentów było porównanie wydajności, stabilności oraz zachowania dwóch różnych typów baz danych — **PostgreSQL** (relacyjnej) i **Cassandra** (NoSQL) — w warunkach przetwarzania dużych wolumenów danych w czasie rzeczywistym.

### **5.1. Zakres testów**

---

Przeprowadzone testy obejmowały następujące aspekty:

- **Czas zapisu miliona rekordów.**
- **Obsługa zapytań zliczających (COUNT(\*)).**
- **Zachowanie przy duplikatach kluczy.**
- **Złożone zapytania i możliwości języka zapytań (CQL vs SQL).**
- **Zarządzanie schematem danych.**
- **Zachowanie przy wysokim obciążeniu CPU/RAM.**
- **Stabilność systemów i konfiguracja klastra Cassandra**

## 5.2. Wyniki testów

Test	PostgreSQL	Cassandra
Zapis 1 miliona wierszy	⌚ 266.61 s	⌚ 236.09 s
Zliczanie rekordów (SELECT COUNT(*) )	✅ szybka odpowiedź	✗ zapytanie nie działało bez dodatkowych zabiegów
Zachowanie przy powtórzeniu klucza	✗ błąd unikalności	✅ nadpisanie istniejącego wpisu
Złożone zapytania (filtrowanie, JOIN)	✅ pełne wsparcie SQL	⚠ ograniczone funkcje, ALLOW FILTERING wymagane
Zmiana schematu (np. nowe kolumny)	⚠ wymaga migracji i ALTER TABLE	✅ można dodać kolumnę "w locie"
Obciążenie CPU/RAM	⚖ umiarkowane	⚖ podobne zachowanie
Konfiguracja klastra	✅ gotowe do użycia	✗ problem z uruchomieniem node'ów

PostgreSQL:

```
kafka_db=# select count(*) from csv_records;
   count
-----
 1000000
(1 row)
```

Cassandra:

```
root@cqlsh:log_data> select count(*) from csv_records;
ReadTimeout: Error from server: code=1200 [Coordinator node timed out waiting for replica node's responses] message="Operation timed out - received only 0 responses." info={'consistency': 'ONE', 'required_responses': 1, 'received_responses': 0}
root@cqlsh:log_data>
```

## **5.3. Wnioski**

---

- **Wydajność zapisu:** Cassandra zapisała milion wierszy o około **30 sekund szybciej** niż PostgreSQL. W warunkach produkcyjnych może to mieć znaczenie przy bardzo dużych strumieniach danych, choć różnica nie była ekstremalna.
- **Zliczanie rekordów:** PostgreSQL z łatwością poradził sobie z zapytaniem SELECT COUNT(\*). Cassandra natomiast nie wspiera tego typu zapytania w intuicyjny sposób bez wcześniejszego przygotowania danych lub stosowania workaroundów, co wynika z jej denormalizowanej architektury.
- **Zachowanie przy duplikatach:** PostgreSQL traktuje duplikaty kluczy głównych jako błąd, co jest zgodne z jego silnym modelem spójności. Cassandra natomiast nadpisuje dane o tym samym kluczu partycji, co jest typowe dla systemów eventual consistency.
- **Możliwości zapytań:** PostgreSQL oferuje pełny zestaw operacji relacyjnych (JOIN, agregacje, podzapytania), natomiast Cassandra ma bardzo ograniczony język zapytań CQL – dobre zapytanie musi być oparte na odpowiednim kluczu partycji. W przypadku złożonych filtrów konieczne było użycie ALLOW FILTERING, co jest niezalecane w środowisku produkcyjnym.
- **Elastyczność schematu:** Cassandra pozwala na dynamiczne dodawanie kolumn bez konieczności zmiany całego schematu, co jest wygodne w szybko zmieniających się środowiskach. PostgreSQL wymaga modyfikacji struktury tabeli.
- **Stabilność systemu i zużycie zasobów:** Podczas testów nie zaobserwowano znaczących różnic w wykorzystaniu CPU czy RAM. W momentach intensywnych operacji odczytu/zapisu zużycie zasobów było chwilowo wysokie w obu przypadkach.
- **Problemy z klastrem Cassandra:** Niestety nie udało się w pełni przetestować odporności na awarie, ponieważ uruchomienie wielowęzłowego klastra Cassandry napotkało problemy techniczne. Mimo to wiadomo, że w środowiskach produkcyjnych Cassandra jest projektowana z myślą o dużej odporności na awarie i braku pojedynczego punktu awarii.

## **6. Wnioski**

---

### ***6.1. Która baza sprawdziła się lepiej?***

---

Na podstawie przeprowadzonych testów można sformułować następujące wnioski dotyczące przydatności poszczególnych baz danych w różnych kontekstach:

- **PostgreSQL** sprawdził się znakomicie w przypadku:
  - pracy z relacyjnym modelem danych,
  - wykonywania złożonych zapytań SQL (JOIN, GROUP BY, podzapytania),
  - silnej spójności danych (transakcje, integralność kluczy),
  - jedno- lub kilkuwęzłowych środowisk o umiarkowanej skali.
- **Cassandra** wykazała przewagę w:
  - szybkim i skalowalnym zapisie danych,
  - prostych, bezpośrednich zapytaniach opartych na kluczu partycji,
  - elastycznym schemacie danych (łatwość dodawania kolumn),
  - środowiskach wymagających wysokiej dostępności i skalowalności poziomej (rozproszone systemy).

### ***6.2. Rekomendacje do przyszłych zastosowań***

---

Wybór bazy danych powinien być zawsze dopasowany do konkretnego przypadku użycia. Na podstawie testów można zaproponować:

- **PostgreSQL** jako bazę preferowaną w systemach:
  - analitycznych,
  - o wysokich wymaganiach dotyczących spójności,
  - gdzie dane mają złożone relacje.
- **Cassandra** jako bazę preferowaną w systemach:
  - czasu rzeczywistego,
  - o dużej skali zapisu (np. logi, IoT, dane telemetryczne),
  - rozproszonych, wymagających niezawodności i dostępności.