



POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Technologie sieci urządzeń mobilnych oraz chmury obliczeniowej

Szymon Strzelka

Nr albumu s20122

Bazy danych NoSQL jako alternatywa baz relacyjnych w aplikacjach rozproszonych

Praca magisterska

dr inż. Mariusz Trzaska

Warszawa, czerwiec 2021

STRESZCZENIE

Poniższa praca opisuje możliwość zastosowania baz typu NoSQL jako alternatywy dla baz relacyjnych w aplikacjach rozproszonych. Bazy relacyjne przez dziesiątki lat dominowały rynek rozwiązań bazodanowych. Ostatnie lata przyniosły jednak niespotykany wcześniej przyrost ilości użytkowników oraz danych. Bazy relacyjne zaczęły stawać się “wąskim gardłem” aplikacji w kontekście wydajności. Wykorzystanie jednego serwera bazodanowego stanowi także pojedynczy punkt awarii, gdyż w przypadku braku dostępu do bazy, aplikacje nie mogą funkcjonować prawidłowo.

Praca przedstawia porównanie baz relacyjnych i NoSQL z perspektywy wspieranych funkcjonalności, charakterystyki oraz zastosowań. Autor skupił się również na zestawieniu wydajności obu systemów, jak również wpływu na odczucia użytkownika.

W ramach pracy magisterskiej powstała również aplikacja webowa, składająca się z aplikacji serwerowej oraz klienckiej. Stworzona aplikacja pozwoliła na ukazanie podobieństw i różnic pomiędzy systemami bazodanowymi oraz przeprowadzenie testów wydajnościowych. Zastosowane rozwiązania mogą także posłużyć jako prototyp podczas trzycy wykorzystywanego typu bazy danych, zapewniając transparentność rozwiązania.

ABSTRACT

The master's thesis presents NoSQL database as an alternative to relational database in distributed systems. Relational databases used to be the first choice as a datastore solution for dozens of years. However, due to rapid growth in the number of users and data processed by the applications, relational databases started to become the bottleneck of distributed systems. Also, having a single database instance is considered a high risk to the application's reliability, as the database becomes a single point of failure.

The thesis compares the relational and NoSQL databases in terms of supported functionalities, internal characteristics, and usual usage patterns. Special consideration was given to the performance of the database systems, as well as to end-user experience.

In addition to the theoretical considerations, a web application has been created, containing a server and frontend parts. The application was used to present the similarities and differences between the database systems. Also, having a dedicated application supporting both types of databases at the same time had allowed performing performance tests. The solution may later be used as a prototype for database-type migration, providing full transparency for the end-users.

SPIS TREŚCI

1. WSTĘP	7
1.1 Zakres pracy	7
1.2 Motywacja	7
1.3 Zawartość pracy	8
2. PORÓWNANIE BAZ RELACYJNYCH I NOSQL	10
2.1 Cechy baz relacyjnych	10
2.2 Cechy baz NoSQL	11
2.3 Teoria CAP	13
2.4 Teoria PIE	14
2.5 Porównanie rozwiązań	16
3. WYBRANE SYSTEMY ZARZĄDZANIA BAZAMI DANYCH	17
3.1 PostgreSQL	17
3.1.1 Indeksy	18
3.1.2 Wyszukiwanie	18
3.1.3 Język zapytań	19
3.1.4 Wyszukiwanie pełnotekstowe	19
3.2 Redisearch	20
3.2.1 Indeksy	20

3.2.2 Wyszukiwanie	21
3.2.3 Język zapytań	21
3.2.4 Wyszukiwanie pełnotekstowe	22
4. ZASTOSOWANE TECHNOLOGIE	23
4.1 REST	23
4.2 Python	25
4.3 Flask	26
4.4 SQLAlchemy	27
5. PROJEKT I IMPLEMENTACJA	29
5.1 Projekt baz danych	29
5.2 Wyszukiwanie	31
5.3 Aktualizowanie rekordów	33
5.4 Porównanie spójności obu rozwiązań	34
6. TESTY WYDAJNOŚCIOWE	36
6.1 Środowisko testowe	36
6.2 Założenia testów	38
6.3 Wydajność wstawiania wierszy	38
6.4 Wydajność aktualizacji	39
6.5 Wydajność wyszukiwania	41

6.5.1 Wyszukiwanie pełnotekstowe	41
6.5.2 Wyszukiwanie prefiksowe	42
6.5.3 Wyszukiwanie po atrybutach tekstowych i numerycznych	44
6.5.4 Wyszukiwanie z użyciem skomplikowanych zapytań	46
6.6 Problem głębokiego stronicowania	47
6.7 Wnioski	51
7. WPŁYW NA ODCZUCIA UŻYTKOWNIKA	53
7.1 Spójność danych	53
7.1.1 Scenariusz aktualizacji danych	54
7.1.2 Problem “hałaśliwych sąsiadów”	56
7.2 Wspierane funkcjonalności wyszukiwania pełnotekstowego	57
8. PODSUMOWANIE	58
WYKAZ RYSUNKÓW	59
WYKAZ TABEL	60
WYKAZ LISTINGÓW	61
WYKAZ LITERATURY	63

1. WSTĘP

Tematem pracy magisterskiej jest zaprezentowanie baz typu NoSQL jako alternatywy baz relacyjnych w aplikacjach rozproszonych. Celem pracy jest pokazania zalet i wad obu rozwiązań, z naciskiem na porównanie wydajności oraz doświadczeń użytkownika (ang. *user experience*). Ponadto przeprowadzone badania pokażą różnice we wsparciu wyszukiwania pełnotekstowego (ang. *full text search*) dla obu typów silników bazodanowych. Niniejsza praca prezentuje zbiór rozważań teoretycznych, badań oraz doświadczeń z użytkowania. Częścią pracy jest również implementacja aplikacji księgarni internetowej, która umożliwi ukazanie różnic pomiędzy typami baz danych.

1.1 Zakres pracy

Zakres pracy obejmuje następujące elementy:

1. Porównanie teoretyczne baz relacyjnych oraz baz typu NoSQL
2. Zaproponowanie bazy typu NoSQL jako alternatywy dla bazy relacyjnej
 - a. Znalezienie i opisanie optymalnego sposobu modelowania danych przechowywanych w bazie relacyjnej dla bazy nierelacyjnej
 - b. Implementacja aplikacji wykorzystującej oba typy baz danych, w celu przedstawienia różnic pomiędzy nimi
 - c. Porównanie doświadczeń użytkownika i wydajności obu systemów, z szczególnym naciskiem na wyszukiwanie pełnotekstowe
3. Krytyczna ocena efektów wykorzystania przedmiotowego systemu.

1.2 Motywacja

Popularność relacyjnych baz danych utrzymuje się wciąż na wysokim poziomie [1]. Głównymi powodami takiego stanu rzeczy jest ich uniwersalność oraz prostota wykorzystania i późniejszego utrzymania. Dużą zasługą jest także obecność języka SQL, który zapewnia usystematyzowany sposób zarządzania danymi.

Przełom drugiego tysiąclecia przynosi niespotykany do tej pory wzrost ilości przechowywanych informacji, głównie za sprawą popularyzacji dostępu do Internetu oraz spadku cen nośników danych. Powstanie portali społecznościowych, sklepów internetowych,

oraz szeroko pojęty trend *big data*, powoduje pojawienie się nowych wyzwań. Wyzwań, które przekroczyły możliwości relacyjnych baz danych. Aby móc sprostać tym warunkom zaczęły powstawać nowe silniki bazodanowe, które odchodziły od postulatów transakcyjności i klasycznej, tabelarycznej reprezentacji danych.

Mimo wyraźnych zalet nowych silników, bazy relacyjne wciąż są standardem na rynku. Składa się na to wiele czynników, między innymi brak kompleksowego wsparcia transakcji, problemy z zarządzaniem spójnością czy inny sposób mapowania obiektowo-relacyjnego. Ponadto, dzięki dobrej znajomości tego typu rozwiązań przez firmy tworzące oprogramowanie, a także szeroką ofertę szkoleń i ogólnodostępnych materiałów edukacyjnych, wybór technologii bazodanowej często pada na silniki relacyjne.

Celem autora pracy jest pokazanie, iż bazy danych typu NoSQL mogą być wartościową opcją dla aplikacji rozproszonych. W Internecie, szczególnie polskojęzycznym, brakuje materiałów dotyczących udanych migracji z baz relacyjnych. Autor ma nadzieję, że stworzona praca ułatwi podjęcie decyzji o wykorzystaniu bazy danych typu NoSQL w produktach polskich firm.

1.3 Zawartość pracy

Pracę podzielono na 8 rozdziałów. Poniżej przedstawiono ich krótką charakterystykę.

- W rozdziale drugim porównano bazy relacyjne i bazy typu NoSQL, skupiając się na ich najistotniejszych cechach. Ukazane zostały także teorie CAP oraz PIE.
- Rozdział trzeci opisuje propozycje rozwiązania oraz dokładniejszy opis użytych technologii bazodanowych.
- Rozdział czwarty przedstawia rozwiązania technologiczne użyte podczas tworzenia aplikacji. Każda z technologii została krótko scharakteryzowana, przedstawiono także argumenty za wyborem danych rozwiązań
- W rozdziale piątym zaprezentowane zostały projekty baz danych oraz trudności implementacyjne dla problemu aktualizacji rekordów w bazie typu NoSQL. Zaprezentowano również mechanizm porównywania spójności rozwiązań przy tranzycji z jednego systemu bazodanowego na alternatywny.

- Rozdział szósty to testy wydajnościowe aplikacji stworzonej na potrzeby pracy magisterskiej. Każdy podrozdział opisuje inny typ testu, od aktualizacji rekordów, po wyszukiwanie i problem głębokiego stronicowania.
- W rozdziale siódmym autor pracy przedstawił wpływ wykorzystanego silnika bazodanowego na odczucia użytkownika. Autor skupił się na problemie spójności danych oraz współdzielenia infrastruktury chmurowej przez wielu użytkowników aplikacji.
- Rozdział ósmy to krótkie podsumowanie całości pracy magisterskiej.

2. PORÓWNANIE BAZ RELACYJNYCH I NOSQL

W poniższym rozdziale przedstawione zostaną różnice pomiędzy dwoma systemami bazodanowymi. Celem obu systemów jest przechowywanie danych oraz zapewnienie efektywnego sposobu dostępu do nich. Na pierwszy rzut oka różnice pomiędzy typami baz danych mogą wydawać się nieznaczące, ponieważ mechanizmy wykorzystywane przez nie ukrywają przed użytkownikiem szczegóły implementacyjne. Jednak przy bliższym poznaniu tych rozwiązań uwypuklają się bardzo znaczące różnice pomiędzy nimi.

2.1 Cechy baz relacyjnych

Relacyjne bazy danych pozwalają przechowywać powiązane ze sobą elementy, oferując do nich łatwy dostęp. Każdy wiersz jest rekordem z unikatowym identyfikatorem zwanym kluczem. Model danych zapewnia standardowy sposób reprezentowania danych, możliwy do wykorzystania przez rozmaite aplikacje klienckie. Dane przechowywane są w sposób standaryzowany, co przekłada się na intuicyjny, wydajny oraz elastyczny dostęp [2].

W modelu relacyjnym baz danych wszystkie dane są przechowywane w dwuwymiarowych tabelach (relacjach). W skład bazy danych może wchodzić wiele relacji. Tabela to układ poziomych wierszy, nazywanych rekordami lub krotkami, i pionowych kolumn, nazywanych polami rekordu lub atrybutami. Tabela jest identyfikowana poprzez nazwę.

Standardowo wyróżnia się trzy rodzaje relacji:

- Jeden do jednego - jednemu wierszowi z tabeli A odpowiada dokładnie jeden wiersz z tabeli B. Model ten stosowany jest rzadko, ponieważ nie oferuje on większych zysków niż przechowywanie danych w jednej tabeli
- Jeden do wielu - jednemu wierszowi z tabeli A przypada wiele rekordów z tabeli B
- Wiele do wielu - każdemu rekordowi z tabeli A może przypadać wiele rekordów z tabeli B oraz odwrotnie, rekord z tabeli B może być w relacji z wieloma rekordami w tabeli A.

Jako, że dane naturalnie rozmieszczone są pomiędzy wieloma tabeli, konieczne jest istnienie mechanizmu, który gwarantuje poprawność zapisu danych w każdej sytuacji. Szczególnie trudny jest przypadek współbieżnego dostępu do bazy. Mechanizm zapewniający poprawne działanie nazywamy transakcją. Jego istotą jest integrowanie wielu operacji (na przykład zapisu do wielu tabel) w jedną operację. Współbieżne wykonywanie transakcji wymaga zachowania własności ACID [3], w skład której wchodzi następujące wymagania:

- Niepodzielność (ang. *atomicity*) transakcji oznacza, że każda transakcja albo zostanie wykonana w całości, albo w ogóle – przykładowo jeśli w ramach jednej transakcji odbywać się ma przelew bankowy (zmniejszenie wartości jednego konta i powiększenie innego o tę samą kwotę), to nie może dojść do sytuacji, że z jednego konta ubędzie pieniędzy, a kwota na koncie docelowym będzie bez zmian.
- Spójność (ang. *consistency*) transakcji oznacza, że po wykonaniu transakcji system będzie spójny, czyli nie zostaną naruszone zasady integralności.
- Izolacja (ang. *isolation*) transakcji oznacza, że jeśli dwie transakcje wykonują się współbieżnie, to zwykle (w zależności od poziomu izolacji) nie widzą wprowadzanych przez siebie zmian. Poziom izolacji w bazach danych jest zazwyczaj konfigurowalny i określa, jakich anomalii możemy się spodziewać przy wykonywaniu transakcji [4].
- Trwałość (ang. *durability*) danych oznacza, że system potrafi uruchomić się i udostępnić spójne, nienaruszone i aktualne dane zapisane w ramach zatwierdzonych transakcji, na przykład po nagłej awarii zasilania

2.2 Cechy baz NoSQL

Bazy nierzelacyjne z założenia nie spełniają zasad ACID, za to oferują koncept *eventual consistency* (w wolnym tłumaczeniu - ostatecznie spójne). Silnik bazy danych zapewnia, że procedura zapisu zostanie rozpropagowana na wszystkie węzły w skończonym czasie (zazwyczaj kilka do kilkunastu milisekund) i baza stanie się spójna [5]. Oczwistą wadą tego rozwiązania jest ograniczona możliwość odczytywania danych zaraz po zapisie (ang. *write-after-read hazard*), co może przełożyć się na niepoprawną odpowiedź dla użytkownika końcowego, np. niezaktualizowany koszyk w sklepie internetowym po dodaniu produktu.

Bazy typu NoSQL wykorzystują do przechowywania i wyszukiwania inny model danych niż relacyjny. Definicja ta jest bardzo szeroka i obejmuje bardzo zróżnicowane modele danych. Wybór odpowiedniego jest podyktowany problemem, który musi on rozwiązać. Czasami struktury danych używane przez bazy danych NoSQL są również postrzegane jako „bardziej elastyczne” niż relacyjne bazy danych [6].

Sam pomysł nierzelacyjnych baz danych pochodzi z lat 50. XX wieku. Duży rozkwit przeżywały one w latach 60., w czasie komputerów mainframe [7]. W późniejszych latach uwidoczniła się dominacja baz relacyjnych, co przełożyło się na zastosowanie baz NoSQL tylko

dla bardzo specjalistycznych zastosowań. Jednak wraz z początkiem XXI wieku i wzrostem trendu *big data*, popularyzacja baz nierelacyjnych stała się tylko kwestią czasu. Dużą w tym zasługą firmy Google, która na początku obecnego tysiąclecia oparła swoją wyszukiwarkę oraz produkty takie jak Gmail czy Google Maps właśnie na tego typu bazach. Pozwoliło to na osiągnięcie ponadprzeciętnej wydajności oraz dostępności, co bezpośrednio przełożyło się na dużą popularność tych usług. Od tego czasu popularność baz nierelacyjnych stale się zwiększa. Obecnie duża część produktów “gigantów technologicznych” takich jak Amazon, Facebook, Netflix czy eBay opiera się na tego typu NoSQL [4]. Ponadto, każdy z wiodących dostawców usług chmurowych posiada w swojej ofercie przynajmniej jedną bazę NoSQL, np. Amazon DynamoDB, Google Cloud Datastore czy Azure Cosmos DB.

Modele danych, wykorzystywane najczęściej w bazach typu NoSQL to [8]:

- Klucz- wartość - wykorzystuje mapę (inaczej słownik) jako podstawowy model danych. Klucze muszą pozostać unikatowe, tak aby wyszukiwanie po ich wartości zawsze zwracało te same rezultaty. Jest to najprostszy z modeli danych, inne modele danych często bazują na nim, oferując dodatkowe funkcjonalności za pomocą rozszerzeń. Typ klucza zazwyczaj pozostaje taki sam dla całej bazy danych, natomiast wartość nie ma z góry określonego typu. Część rekordów może posiadać wartość typu prostego, takie jak string lub integer, natomiast inne mogą korzystać z złożonych obiektów. Najpopularniejsze bazy wykorzystujące ten model to Amazon DynamoDB czy Memcached.
- Dokument - jak sama nazwa wskazuje, bazy wykorzystujące ten model danych korzystają z dokumentów do przechowywania danych. Najczęściej dokumenty są przechowywane w jednym ze standardowych formatów, takich jak JSON (ang. *JavaScript Object Notation*), XML (*Extensible Markup Language*) lub YAML (*YAML Ain't Markup Language*). Dostęp do dokumentów jest możliwy za pomocą klucza głównego lub z wykorzystaniem zapytań o wartości dokumentu. Podobnie jak w modelu klucz-wartość, dokumenty nie posiadają sztywno narzuconej struktury. Model dokumentowy jest wykorzystywany między innymi przez Oracle NoSQL Database i MongoDB.
- Graf - najbardziej specyficzny z wymienionych modeli, przechowujący dane w węzłach grafu oraz wykorzystujący krawędzie do modelowania relacji. Ponadto, model ten może

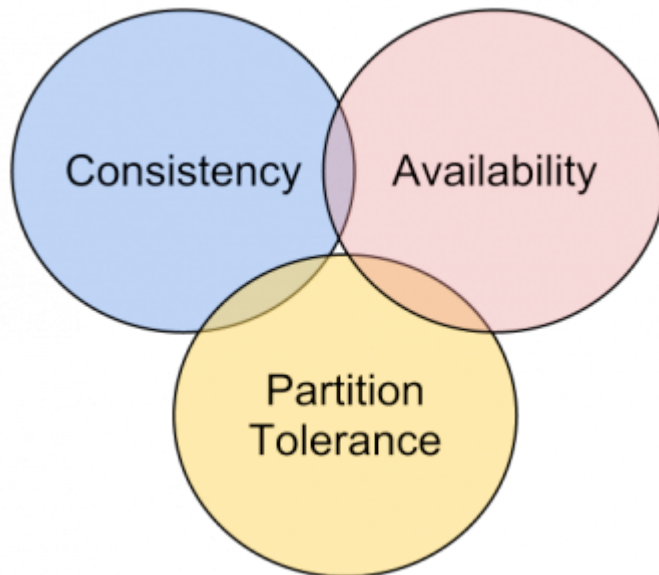
być prezentowany jako pojedynczy graf, co czyni go idealnym do do mocno połączonych danych (np. sieć transportu publicznego). Relacje w modelu grafowym stają się obiektami “pierwszej kategorii” i mogą posiadać kierunek, atrybuty oraz etykiety. Najbardziej znaną bazą wykorzystującą ten model jest Neo4j.

2.3 Teoria CAP

Teoria znana także jako teoria Brewera, od nazwiska jej autora Erica Brewera, mówiąca, że rozproszony system nie może równocześnie zapewnić trzech z poniżej wymienionych cech:

- Spójność (ang. *consistency*) - każdy odczyt powoduje zwrócenie najnowszych danych
- Dostępność (ang. *availability*) - system odpowiada na każde zapytanie użytkownika
- Odporność na awarię sieci (ang. *partition tolerance*) - system jest w stanie pozostać w pełni responsywny, pomimo braku łączności z częścią węzłów lub utratą części pakietów [9].

Wizualną reprezentację teorii CAP przedstawiono na rysunku 2.1.



Rys. 2.1. Przedstawienie teorii CAP za pomocą diagramu Venna [10]

Niezależnie od infrastruktury żadna sieć nie zapewnia 100% dostępności, odporność na jej awarię jest dla rozproszonych systemów wymaganiem, a nie opcją. W momencie wystąpienia awarii, system może albo poświęcić dostępność, a więc zwrócić odpowiedź sugerującą problem z systemem, lub spójność, zwracając dane bez gwarancji ich aktualności.

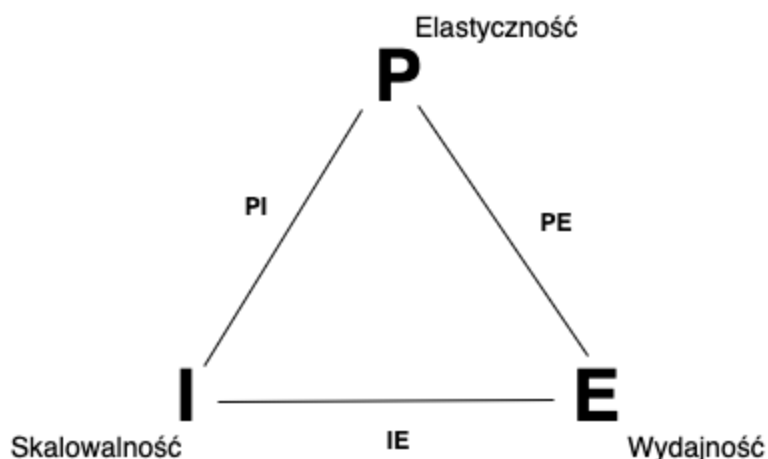
Kompromis pomiędzy dostępnością i spójnością jest wykorzystywany tylko w momencie wystąpienia awarii sieci, w innym przypadku baza może posiadać obie te cechy równocześnie. Brak zrozumienia tego faktu często prowadzi do błędnego zrozumienia całej teorii.

2.4 Teoria PIE

Rick Houlihan, starszy architekt Amazon Web Services, przedstawił w 2018 roku teorię stającą niejako w opozycji do przedstawionej w poprzednim rozdziale teorii CAP. Podczas konferencji AWS re:Invent 2018 [11], zwrócił uwagę na płytkość twierdzenia CAP. Według prezentera w 95% przypadków należy wybrać odporność na awarię sieci oraz spójność, ponieważ dostępność bliska 100% jest bardzo rzadko rzeczywistym wymaganiem systemu.

Przedstawiona przez Ricka Houlihana teoria, podobnie do teorii Brewera, może być przedstawiona za pomocą trójkąta, co zaprezentowano na rysunku 2.2. Analogicznie do teorii CAP, wymaga ona wyboru dwóch z trzech cech systemów rozproszonych. Cechy te, to:

- Elastyczność (ang. *pattern flexibility*) - elastyczny system posiada łatwość dostosowania się do zmiany przypadków użycia
- Wydajność (ang. *efficiency*) - czy użytkownicy systemu wymagają szybkich (maksymalnie kilkaset milisekund) odpowiedzi nawet przy dużym natężeniu ruchu czy będą oni w stanie zaakceptować czasy na poziomie minut lub nawet godzin?
- Skalowalność (ang. *infinite scale*) - jak bardzo będzie rosła ilość danych w czasie, czy liczba użytkowników będzie liczona w tysiącach, setkach tysięcy czy milionach?



Rys. 2.2 Teoria CAP przedstawiona na modelu trójkąta

Najpopularniejszym wyborem wciąż pozostają bazy danych oparte na krawędzi PE, a więc skupione na elastyczności oraz wydajności. Te dwie cechy zaspokajają większość wymagań biznesowych, gwarantując szybki odczyt i zapis oraz możliwości rozbudowy lub zmiany aplikacji w trakcie jej życia. Relacyjne bazy danych pasują do tej kategorii praktycznie idealnie, zapewniając elastyczność za pomocą postaci normalnych i indeksów oraz wydajność z użyciem wydajnych typów pamięci.

Wzrost znaczenia systemów czasu rzeczywistego wpłynął na większe zainteresowanie bazami opartymi na krawędzi IE, czyli zorientowanymi na skalowalność i wydajność. Ilość danych przetwarzanych przez gigantów technologicznych takich jak Google, Amazon czy Facebook rośnie wykładniczo w czasie, znacznie wyprzedzając obecne możliwości pojedynczych maszyn. Odrzucenie postaci normalnych na rzecz denormalizacji pozwala na osiągnięcie skalowania poziomego niemożliwego do osiągnięcia dla baz relacyjnych. Odbywa się to kosztem elastyczności, gdyż wiele systemów NoSQL wymaga przebudowania modelu danych w momencie dodania nowego przypadku użycia.

Systemy cechujące się dużą elastycznością oraz skalowalnością są zdecydowanie najrzadziej spotykane. Najbardziej znanym przykładem takiego systemu jest hurtownia danych, która może wykonywać obliczenia o dużym stopniu skomplikowania na gigantycznych zbiorach danych. Możliwość uruchamiania zróżnicowanych zapytań na zbiorach danych uwzględniających dane historyczne dla wielu użytkowników odbywa się w nich kosztem wydajności, która nie jest tu priorytetem.

2.5 Porównanie rozwiązań

Tabela 2.1 podsumowuje najważniejsze różnice pomiędzy bazami typu NoSQL oraz bazami relacyjnymi.

Tabela 2.1 Najważniejsze różnice pomiędzy bazami relacyjnymi i typu NoSQL [12]

3. WYBRANE SYSTEMY ZARZĄDZANIA BAZAMI DANYCH

Niemożliwe jest porównanie każdego rodzaju bazy relacyjnej z każdym typem bazy typu NoSQL, szczególnie biorąc pod uwagę ilość dostępnych na rynku systemów drugiego rodzaju. Na potrzeby pracy magisterskiej autor postanowił wykorzystać po jednym przykładzie

	bazy NoSQL	bazy relacyjne
Model danych	Elastyczny, możliwość przechowywania różnych struktur danych	Tabele z arbitralnie ustaloną liczbą wierszy
Skalowalność horyzontalna	Liniowa skalowalność horyzontalna	Skalowalność wertykalna, ograniczona możliwościami sprzętowymi
Zachowania spójności	Zazwyczaj brak, przeniesienie odpowiedzialności na użytkownika	Wysoki poziom spójności, wspierany przez zróżnicowane mechanizmy. Kompleksowe wsparcie dla transakcji
Odporność na awarię	Replikacja danych pomiędzy węzłami zapewnia wysoką dostępność	Pojedynczy punkt awarii powoduje niską odporność
Zastosowanie	Zróżnicowane, uzależnione od wspieranego modelu danych	Ogólnego użytku
Zorientowanie obliczeniowe	Pamięć, denormalizacja danych, potrzebne dane dostępne w jednym dokumencie	Moc obliczeniowa, normalizacja danych, złączenia podczas wykonywania zapytań
Elastyczność	Niska, wymagane wprowadzenie zmian w modelu danych w celu dostosowania do nowych przypadków użycia	Wysoka, postać normalna umożliwia szybki i łatwy dostęp do danych rozmieszczonych w różnych tabelach

każdego systemu. Wybór padł na PostgreSQL jako przykład bazy relacyjnej oraz RedisSearch jako alternatywnego rozwiązania.

3.1 PostgreSQL

Darmowy system zarządzania relacyjną bazą danych (ang. RDBMS - *relational database management system*) skupiający się na elastyczności oraz wsparciu dla języka SQL. Początki systemu sięgają lat 80. XX wieku, gdy system ewoluował z projektu Ingres na

Uniwersytecie w Berkeley, USA. Początkowo projekt nazywał się Postgres, jednak w 1996 roku został dodany sufix “SQL”, w celu podkreślenia wsparcia dla tegoż języka. System ten jest wydawany na licencji typu open-source, a od 1997 roku jest w pełni rozwijany przez wolontariuszy [13].

3.1.1 Indeksy

PostgreSQL posiada wbudowane wsparcie dla indeksów opartych na B-drzewach oraz na tablicach haszowanych. Spośród wspieranych funkcjonalności można wyróżnić między innymi indeksy funkcyjne (ang. expression indexes), które zbudowane są na wartościach funkcji operującej na kolumnach tabeli, indeksy częściowe (ang. partial indexes), które przechowują wartość tylko dla wierszy spełniających warunek zdefiniowany przy tworzeniu czy indeksy pełne, które pozwalają na odczytanie wszystkich danych z indeksu, bez konieczności sięgania do właściwej tabeli. Ponadto, od wersji 9.5 wspierane są indeksy blokowe (ang. block range index), które charakteryzują się wysoką wydajnością dla dużych zbiorów danych.

Schematy zachowują się jak przestrzenie nazw, znane z platformy .NET. Pozwalają one na logiczne podzielenie bazy danych, umożliwiając istnienie tabel o tej samej nazwie w różnych schematach. Każda baza danych posiada schemat o nazwie public, następne schematy mogą być tworzone przez użytkowników bazy.

3.1.2 Wyszukiwanie

Do wyszukiwania danych w PostgreSQL wykorzystuje się język SQL (ang. *Structured Query Language*). Listing 3.1 przedstawia proste zapytanie służące do pobrania wszystkich rekordów z tabeli book.

Listing 3.1 Przykład wyszukiwania danych z użyciem języka SQL

```
SELECT * FROM book;
```

W celu pobrania rezultatów spełniających dane kryteria wykorzystuje się operator *WHERE*, po którym należy podać listę warunków. Listing 3.2 przedstawia zapytanie wyciągające książki o liczbie stron przekraczającej 1000 lub których autorem jest Bob Johnson.

Listing 3.2 Przykład wyszukiwania danych spełniających podane kryteria z użyciem języka SQL

```
SELECT * FROM book WHERE pages > 1000 OR author = 'Bob Johnson';
```

3.1.3 Język zapytań

Język SQL istnieje od lat 70. i wciąż pozostaje liderem popularności w rankingu języków dostępu do baz relacyjnych. Służy on nie tylko do wyszukiwania danych, ale również do tworzenia, edycji oraz zarządzania bazami relacyjnymi. Złożone zapytania SQLowe mogą mieć nawet setki linii, zawierając wiele złączeń (ang. *join*), podzapytań czy warunków ograniczających. SQL w żaden sposób nie ogranicza programisty przed użyciem tak skompilowanych, często suboptymalnych, zapytań. Wraz ze wzrostem skomplikowania zapytania, silnik bazodanowy wykonuje bardziej złożone operacje (często wykorzystując struktury pomocnicze) co może negatywnie odbić się na czasach odpowiedzi.

3.1.4 Wyszukiwanie pełnotekstowe

Wyszukiwanie pełnotekstowe to bardzo ogólna nazwa problemu, w skład której wchodzi szerokie spektrum problemów wyszukiwania fraz w przechowywanych danych tekstowych. Początki wyszukiwania pełnotekstowego sięgają lat 90. XX wieku, gdy pojawienie się systemów dla bibliotek ujawniło potrzebę stworzenia silników wyszukiwania (ang. *search engine*) radzących sobie z gigantycznymi zbiorami danych. W ogólności, wyszukiwanie pełnotekstowe skupia się na wyszukaniu danego wyrażenia (ang. *term*) lub zbioru słów w dużych zbiorach danych.

Do wsparcia wyszukiwania pełnotekstowego w PostgreSQL wykorzystuje się typ *tsvector* [14]. Na listingu 3.3 przedstawiono operacje tworzącą instancję tego typu.

Listing 3.3 Utworzenie instancji typu *tsvector* w PostgreSQL

```
SELECT to_tsvector('The quick brown fox jumped over the lazy  
dog.');
```

Silnik bazodanowy sprowadzi przekazany tekst do formy leksemu oraz usunie słowa z tak zwanej “stop listy” (ang. *stop words*), w celu minimalizacji ilości przechowywanych danych. Przykładowo, “stop lista” dla języka angielskiego zawiera wyrazy takie jak *a*, *the* oraz *and*.

Do wyszukiwania pełnotekstowego wykorzystuje się funkcję *tsquery* wraz z operatorem '@@'. Listing 3.4 prezentuje wyszukiwanie pełnotekstowe na atrybucie *title*. Operator kreski pionowej (ang. *pipe*) pełni rolę łącznika logicznego OR. Istnieje także wsparcie dla operatora koniunkcji (z wykorzystaniem symbolu '&') oraz negacji (symbol '!').

Listing 3.4 Wyszukiwanie pełnotekstowe w PostgreSQL

```
SELECT * FROM book WHERE title @@ to_tsquery('night | day');
```

PostgreSQL posiada także wsparcie dla bardziej złożonych operacji wyszukiwania pełnotekstowego, takich jak wyszukiwanie frazowe (ang. *phrase search*) czy sortowanie z uwzględnieniem wag pól. Dokładniejszy opis wspieranych funkcjonalności znajduje się w rozdziale siódmym.

3.2 RediSearch

RediSearch to baza danych typu NoSQL zbudowana na rozwiązaniu zwanym Redis, rozwijanym przez firmę Redis Lab. Trzy najważniejsze funkcjonalności, które wyróżniają ten system na tle innych baz NoSQL to [15]:

- Wyszukiwanie pełnotekstowe (ang. *full text search*) pozwalające na użyciu indeksów tekstowych na nieprzetworzonych danych. Umożliwia to wydajne wyszukiwanie fraz, w zbiorach liczących wiele milionów rekordów. Ponadto, wbudowana obsługa stemmingu oraz tokenizacji pozwala na znalezienie rekordów z zadany stopniem podobieństwa.
- Indeksy dodatkowe (ang. *secondary indexes*), które umożliwiają używanie wielu indeksów na pojedynczym dokumencie.
- Sugestie, które mogą być wykorzystane w momencie wprowadzania fraz przez użytkownika. Bazują one na modelu prefiksów, podpowiadając możliwe zakończenia słowa, które użytkownik wprowadza.

3.2.1 Indeksy

Aby umożliwić wydajne wyszukiwanie, RediSearch musi znać strukturę dokumentu. Każdy dokument może zawierać wiele par klucz-wartość, również różnego typu, które powinny być zarządzane w inny sposób. Możliwe jest również nadanie wag wartościom, tak aby silnik bazodanowy mógł nadawać odpowiednie priorytety dokumentom podczas sortowania [16].

Pierwszy krok przy tworzeniu instancji bazy danych to dodanie definicji indeksu, przykładowy indeks dla danych książek jest przedstawiony na listingu 3.5.

Listing. 3.5 Instrukcja utworzenia schematu

```
FT.CREATE book_index SCHEMA
title TEXT
author TEXT
description TEXT
page_count NUMERIC
```

Po dodaniu pierwszego dokumentu, tak jak przedstawiono na listingu 3.6, RediSeach podzieli pola tekstowe na tokeny oraz dokona indeksacji, co umożliwi natychmiastowe wyszukiwanie. Oczywiście, dodawane dokumenty mogą zawierać więcej pól niż wyspecyfikowano w indeksie, jednak sposób indeksowania zostanie wybrany przez silnik bazy danych.

Listing. 3.6 Dodanie dokumentu do indeksu

```
FT.ADD book_index doc1 1.0 FIELDS
title "Clean code"
author "Robert Cecil Martin"
description "Even bad code can function. But if code isn't
clean, it can bring a development organization to its knees."
```

3.2.2 Wyszukiwanie

Przykład wyszukiwania dokumentów przedstawiono na listingu 3.7. Zaprezentowane zapytanie spowoduje przeszukanie dokumentów w celu znalezienia wyspecyfikowanych termów, zwrócone wyniki zostaną posortowane według oceny (ang. *score*), w która wlicza się waga pól oraz stopień podobieństwa do zadanego termu.

Listing 3.7 Przykład wyszukiwania danych w indeksie

```
FT.SEARCH book_index "bad clean knee"
```

3.2.3 Język zapytań

RediSearch wspiera tworzenie złożonych zapytań z wykorzystaniem prostych klauzul połączonych odpowiednimi operatorami. Wśród wspieranych funkcjonalności należy wyróżnić dokładne dopasowywanie (ang. *exact match*), negacje, wyszukiwanie po prefiksie (ang. *prefix match*) czy wsparcie dla wyszukiwania geolokalizacyjnego. Aby ograniczyć wyszukiwanie do pól indeksu, należy poprzedzić dane termy symbolem '@' oraz nazwą pola, tak jak pokazano na listingu 3.8.

Listing 3.8 Złożone wyszukiwanie danych w indeksie

```
FT.SEARCH book_index  
@title: (code) | @description: (bad | clean | knee)
```

Powyższy listing wykorzystuje także operator *OR* wyrażony za pomocą symbolu kreski pionowej (ang. *pipe*).

3.2.4 Wyszukiwanie pełnotekstowe

Szerokie wsparcie dla wyszukiwania pełnotekstowego to jedna z cech RediSearcha. Zapewnienie całego wachlarza funkcjonalności oraz dobrej wydajności tego typu wyszukiwania jest jedną z motywacji twórców tego systemu.

Celem umożliwienia wyszukiwania pełnotekstowego należy poprawnie zdefiniować definicje indeksu podczas jego tworzenia. Przykład poprawnie skonfigurowanego indeksu pokazany jest na listingu 3.5, w rozdziale 3.2.1.

Wśród wspieranych funkcjonalności wyszukiwania pełnotekstowego należy wyróżnić [17]:

- wyszukiwanie frazowe
- wsparcie dla operatorów logicznych
- wyszukiwanie prefiksowe (ang. *prefix match*)
- wyszukiwanie rozmyte (ang. *fuzzy match*)

Listing 3.9 to przykład wyszukiwania prefiksowego dla pola *title* oraz wyszukiwania rozmytego dla pola *author*. Wyszukiwanie rozmyte wykorzystuje odległość Levensthein [18], który jest miarą odmienności napisów. Odmienność dwóch napisów jest równa ilości znaków, dla których należy wykonać działania (usunięcie, dodanie lub zamianę), po których napisy będą identyczne. Przykładowo, odległość dla napisów *test* oraz *task* wynosi 1 (należy zamienić *t* na *k*), natomiast dla *bug* i *baggy* wynosi 3 (wymiana *u* na *a* oraz dodanie *g* i *y*).

Listing 3.9 Wyszukiwanie z wykorzystaniem prefiksów oraz *fuzzy match*

```
FT.SEARCH book_index  
@title: "day*" | @author: "%bug%"
```

Ilość wykorzystanych par znaków procenta (w powyższym listingu jedna) to maksymalna odległość od zadanego terminu.

4. ZASTOSOWANE TECHNOLOGIE

W aplikacji stworzonej na potrzeby pracy magisterskiej wykorzystane zostały bardzo popularne technologie do tworzenia aplikacji webowych. Magistrat zdecydował się na stworzenie aplikacji bezstanowej, wspierającą architekturę REST. Wybór Pythona, jako języka programowania wykorzystanego do utworzenia, części serwerowej aplikacji, był podyktowany łatwością implementacji, a także dostępnością bezpłatnych bibliotek. Ich użycie znacznie ułatwiło pracę nad aplikacją, zmniejszając liczbę komponentów, które musiały powstawać w celu realizacji założonego projektu.

4.1 REST

REST (ang. *Representational State Transfer*) to jeden ze stylów architektury oprogramowania, korzystający z HTTP jako protokołu transportowego. Najważniejsze cechy tej architektury to:

- jednolity interfejs
- bezstanowość
- model klient - serwer
- możliwość zapisania wyniku zapytania w pamięci podręcznej (ang. cacheable)
- warstwowość

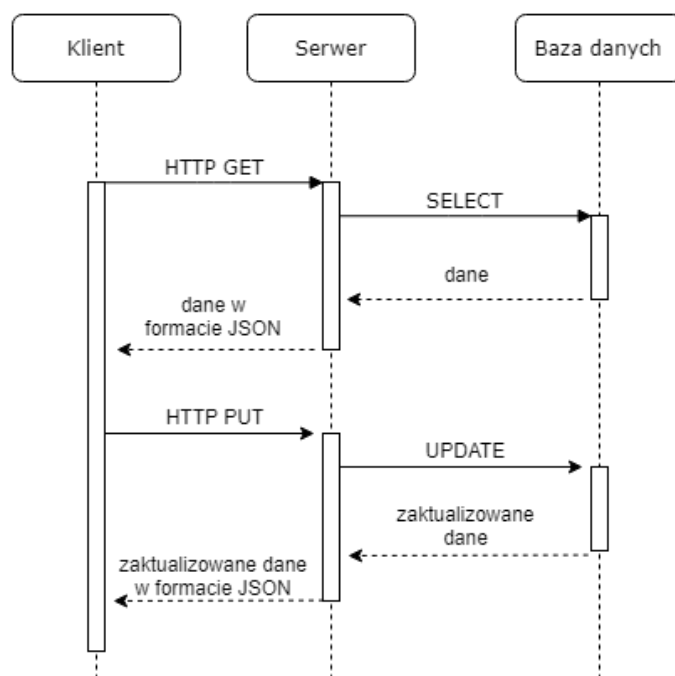
Każdy zasób identyfikowany jest unikatowym URI (ang. *Uniform Resource Identifier*). Zasoby są uporządkowane hierarchicznie, zazwyczaj w strukturze drzewiastej, pozwalając na łatwą eksplorację. W architekturze RESTowej nie ma wymogu użycia jednego formatu danych. Zasoby mogą być reprezentowane przez różne formaty. Ułatwia to integracje z aplikacjami klienckimi. Do najpopularniejszych formatów należą: JSON i XML. W projekcie magistrant skorzystał z pierwszego z podanych formatów.

Do operacji na zasobach wykorzystywane są metody protokołu HTTP. W specyfikacji występuje aż dziewięć metod [19], najczęściej wykorzystywane z nich to:

- GET - służy do pobrania kolekcji lub pojedynczego zasobu. Zapytanie tego typu nie powinno nigdy powodować żadnych efektów ubocznych, na przykład zmiany danych
- POST - wykorzystywany do tworzenia nowych zasobów.

- PUT - służy do edycji istniejących zasobów. Zapytania tego typu powinny być idempotentne, zapewnia się to poprzez wymuszanie przesłania wszystkich właściwości aktualizowanego zasobu.
- DELETE - wykorzystywany do usuwania elementów po stronie serwera.
- OPTIONS - najczęściej używany w celu poznania możliwych operacji na danym zasobie.

Przykładowy przepływ danych wykorzystujący metody GET i PUT znajduje się na rysunku 4.1.



Rys 4.1. Przykładowy przepływ danych w architekturze REST

Jako odpowiedź klient otrzymuje kod zapytania HTTP i opcjonalnie ciało odpowiedzi. Specyfikacja standardu HTTP definiuje wiele kodów odpowiedzi, w praktyce najczęściej wykorzystywanych jest poniżej dziesięciu. Tabela 4.1 przedstawia najpopularniejsze z nich.

Tabela 4.1 Najpopularniejsze kody odpowiedzi HTTP

Kod	Znaczenie
200	OK- zapytanie zakończone sukcesem
400	Nieprawidłowe żądanie- błędna składnia zapytania
401	Nieautoryzowany dostęp- użytkownik nie dołączył wymaganego nagłówka z tokenem dostępu lub token został odrzucony przez serwer
403	Zabroniony- użytkownik nie ma wystarczających uprawnień, aby wykonać tą operację
404	Nie znaleziono- serwer nie odnalazł danego zasobu
409	Konflikt- wystąpił konflikt z już istniejącym zasobem (np. złamanie unikatowości identyfikatorów)
413	Encja zapytania zbyt długa- przesyłany plik przekroczył dozwolony limit
500	Wewnętrzny błąd serwera- zwracany tylko w przypadku błędów, na które nie ma wpływu klient

4.2 Python

Python jest uznawany za najbardziej czytelny oraz najprostszy do nauki język programowania [20]. Jego składnia jest bardzo zbliżona do języka naturalnego, ułatwiając zrozumienie go osobom spoza branży IT. Co więcej, język ten nie wspiera nawiasów klamrowych, zamiast tego do wydzielania bloków kodu wykorzystywane są wcięcia. Python

posiada bogate wsparcie społeczności oraz rozbudowane biblioteki standardowe, co niewątpliwie jest przewagą nad języki takimi jak C++.

Python posiada dynamiczny system typów, w odróżnieniu od innych popularnych języków do tworzenia aplikacji serwerowych (jak na przykład Java czy C#). Oznacza to, że typ zmiennej zależy od chwilowo przechowywanej wartości. Może się on zmienić w momencie przypisania innej wartości do zmiennej.

Python nie wspiera wszystkich paradygmatów programowania obiektowego. Programiści mogą korzystać z abstrakcji, polimorfizmu oraz dziedziczenia. Nie posiada on wsparcia dla hermetyzacji, wszystkie pola oraz metody klasy są zawsze publiczne.

Język ten tworzony jest jako wolne oprogramowanie (ang. *open source*) na licencji, która pozwala na tworzenie przy jego pomocy komercyjnych aplikacji. Python cały czas jest aktywnie rozwijany, w połowie roku 2021 posiadał około 500 wersji, dostępnych na portalu github.com.

Z Pythona korzystają między innymi NASA, Google, Youtube, Netflix oraz Facebook.

4.3 Flask

Flask to „micro” framework do tworzenia części serwerowej aplikacji webowych. Określenie „micro” nie oznacza, że jest on przeznaczony tylko do małych aplikacji ani, że brakuje mu zaawansowanych funkcjonalności. Twórcy przez ten przymiotnik rozumieją fakt, że tworzony przez nich framework nie posiada zintegrowanych, narzuconych przez nich zewnętrznych komponentów oraz bibliotek [21]. Wybór modułu do mapowania obiektowo-relacyjnego, walidacji formularzy czy wysyłania e-maili pozostawiają użytkownikom. Ta decyzja wydaje się bardzo rozsądna, ponieważ na rynku istnieje wiele przetestowanych, udokumentowanych i utrzymywanych rozwiązań.

Flask wspiera tylko język Python, został również w nim stworzony. Wykorzystanie tego języka pozwala na korzystanie z niego na wszystkich najpopularniejszych platformach.

Jak przedstawiono na listingu 4.1, do stworzenia aplikacji serwerowej z jednym zasobem dostępnym w protokole HTTP wystarczy pięć linii kodu.

Listing. 4.1 Prosta aplikacja stworzona z użyciem frameworka Flask

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello World!'
```

4.4 SQLAlchemy

SQLAlchemy to biblioteka do mapowania obiektowo-relacyjnego, którego celem jest odwzorowanie obiektowej struktury utworzonej w wysokopoziomowym języku programowania na relacyjną strukturę bazy danych. Dzięki wykorzystaniu tego rodzaju mapowania, programiści mogą pracować na wyższym poziomie abstrakcji, są w bardzo niewielkim stopniu uzależnieni od implementacji silnika bazy danych. Dodatkowo zarówno SQLAlchemy, jak i inne biblioteki zapewniające podobne funkcjonalności, gwarantują wyższy poziom bezpieczeństwa pomiędzy aplikacją a bazą danych niż w przypadku własnej implementacji [22].

Listing. 4.2 Klasa *User* wykorzystująca SQLAlchemy

```
class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True, autoincrement=True)
    first_name = Column(String)
    last_name = Column(String)
    born_date = Column(Date)
    email = Column(String)
    password = Column(String)
    salt = Column(String)
    create_date = Column(DateTime)
```

Klasa przedstawiona na listingu 4.2 dziedziczy po klasie *Base* z SQLAlchemy. Jest to sygnał dla tej biblioteki, aby utworzyć tabelę o nazwie podanej w atrybucie “*__tablename__*”. Pozostałe atrybuty są inicjowane konstruktorem klasy *Column*, do którego podaje się typ, który ma przyjąć ta zmienna. Jednak Python, jako język dynamicznie typowany, nie ma możliwości sprawdzenia typów podczas inicjalizacji obiektu należącego do tej klasy. Ewentualne błędy

nastąpią dopiero podczas wywołania operacji na bazie danych. Takie zachowanie wymusza na programiście większą staranność podczas tworzenia kodu.

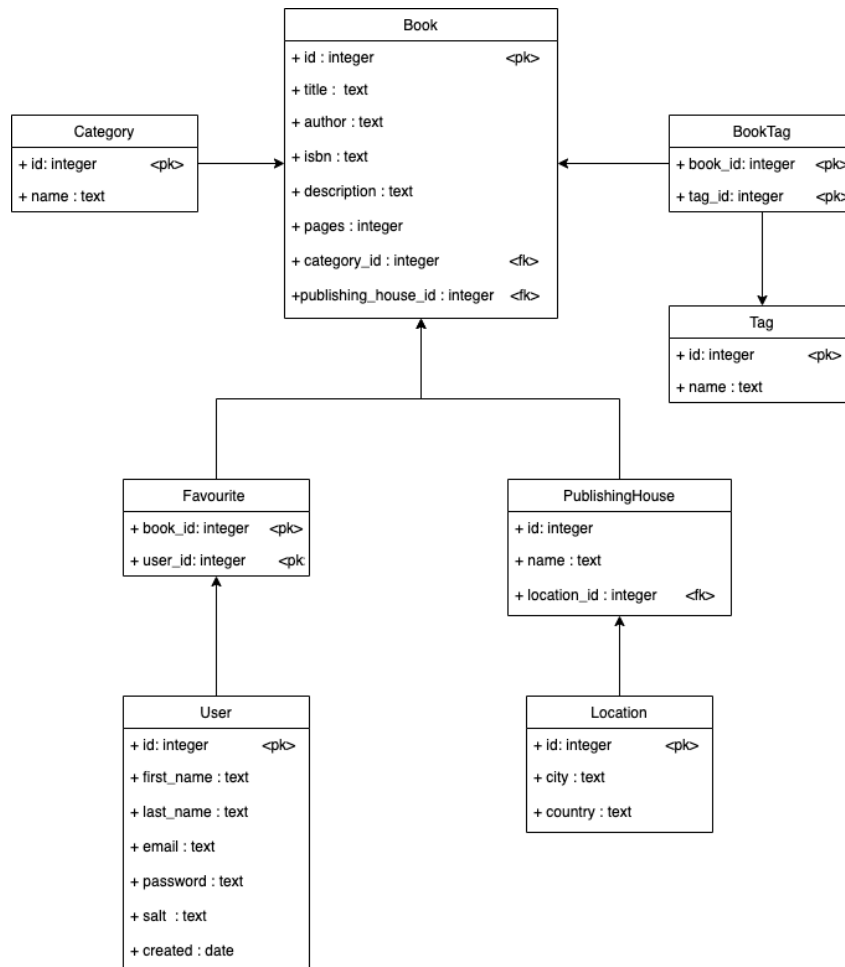
5. PROJEKT I IMPLEMENTACJA

Za pomocą przedstawionych w poprzednim rozdziale technologii została stworzona przez magistranta aplikacja webowa oferująca podstawowe funkcjonalności serwisu poświęconego książkom. Poniższy rozdział przedstawia techniczne rozwiązania zastosowane w aplikacji. Wybrane rozwiązania musiały umożliwić ukazanie różnic pomiędzy dwoma typami baz danych. Różnorodności te zostaną dokładniej opisane w następnych rozdziałach.

5.1 Projekt baz danych

Dla bazy relacyjnej oraz typu NoSQL stworzone zostały odrębne projekty bazy danych. Niektóre bazy dokumentowe umożliwiają użycie identycznego modelu jak w bazie relacyjnej. Niestety, silnik Redisearch nie pozwala na wyszukiwanie po atrybutach zagnieżdżonych (ang. *nested query*), co wymusiło użycie modelu w pełni zdenormalizowanego [23].

Na rycinie 5.1 przedstawiony został relacyjny model danych. Na potrzeby aplikacji utworzone zostało osiem tabel, sześć z nich było ściśle związanych z domeną książki. Dwie pozostałe zostały utworzone celem umożliwienia rejestracji i logowania użytkowników oraz wsparcia funkcjonalności dodawania książek do listy ulubionych. Zastosowany model danych umożliwił ukazanie podstawowych cech silnika PostgreSQL oraz przeprowadzenia testów wydajnościowych.



Rys. 5.1 Model danych dla bazy relacyjnej

Model w pełni zdenormalizowany wymusza umieszczenie wszystkich atrybutów książki w jednym dokumencie. Magistrant zdecydował się na umieszczenie każdego typu danych w osobnych indeksach, co nie jest wymagane dla bazy dokumentowej. Możliwe jest przechowywanie wszystkich danych w jednym indeksie, nawet dla bazy danych zawierającej dane różnych użytkowników. W takim przypadku kluczowa jest staranność i skrupulatność przy operacjach na danych, gdyż niewłaściwa logika biznesowa aplikacji może modyfikować dane innych użytkowników. W bazach relacyjnych najczęściej wykorzystuje się mnogość instancji na jednej instancji fizycznej, wymagają uwierzytelniania dla każdej instancji i każdego połączenia z bazą [24].

Listing 5.1 przedstawia definicję dokumentu *book* dla bazy Redisearch. Definicja pozostałych dokumentów jest analogiczna do przedstawionej na tym listingu oraz do modelu relacyjnego, więc została ona pominięta.

Słowo kluczowe *WEIGHT* pozwala na ustawienie istotności danego pola. Wykorzystywane jest to podczas sortowania rezultatów, encja z frazą znajdującą się w polu o najwyższej wadze będzie zwrócona na górze listy rezultatów. Drugie interesujące słowo kluczowe, czyli *SORTABLE*, umożliwia sortowanie po polach, które je wykorzystują.

Listing 5.1 Definicja indeksu użytego w aplikacji stworzonej przez magistranta

```
CREATE INDEX book SCHEMA
id NUMERIC
title TEXT WEIGHT 10.0
description TEXT WEIGHT 5.0
author TEXT WEIGHT 2.0
isbn TEXT WEIGHT 2.0
category TEXT
publishing_house_name TEXT
publishing_house_city TEXT
publishing_house_county TEXT
tags TEXT
pages NUMERIC SORTABLE
```

5.2 Wyszukiwanie

Aplikacja stworzona na potrzeby pracy magisterskiej umożliwia wyszukanie po wszystkich atrybutach książki. Aby zapewnić szerokie możliwości wyszukiwania został stworzony dedykowany język zapytań, interpretowany przez aplikację serwerową przed wykonaniem zapytania do bazy. Język ten wspiera następujące operatory:

- LIKE (~) - wyszukiwanie pełnotekstowe
- EQUALS (=) - wyszukiwanie frazowe
- GREATER THAN, LOWER THAN (>, <) - wyszukiwanie zakresowe
- OR - spójnik logiczny ‘lub’
- AND - spójnik logiczny ‘i’
- NOT - operator negacji

Zapytanie przedstawione na listingu 5.2 zwróci książki, w których tytule jest słowo ‘night’ i których autor nie ma na imię lub nazwisko ‘Johnson’ lub książki, których liczba stron przekracza 1000. Wykorzystanie nawiasów pozwala na logiczne pogrupowanie klauzul, pozwalając użytkownikom na tworzenie skomplikowanych zapytań.

Listing 5.2 Proste zapytanie wykorzystujące język zapytań użyty w aplikacji

```
(title ~ night AND NOT author = johnson) OR pages > 1000
```

Do tłumaczenia zapytań wysyłanych przez użytkownika na obiekty posłużyła biblioteka *Lark* [25]. Wspiera ona wykorzystanie notacji EBNF (ang. *Extended Backus-Naur Form*) [26] jako sposobu wyrażenia języka formalnego. Zbiór reguł wykorzystanych w aplikacji prezentuje listing 5.3. Każda reguła składa się ze zmiennej (po lewej stronie) oraz wartości, która najczęściej wykorzystuje inne zmienne lub wyrażenia regularne. Po dopasowaniu zapytania do reguł, tworzone są obiekty dziedziczące po klasie *Query*. Każda implementacja klasy bazowej wspiera jedną klauzulę, przykładowo operator *OR* spowoduje utworzenie instancji klasy *OrQuery*, a klauzula *title ~ night*, klasy *FullTextSearchQuery*. Instancje klasy *Query* tłumaczone są następnie na język SQL lub język zapytań Redisearch.

Listing 5.3 Zbiór reguł gramatycznych użytych do tłumaczenia zapytań użytkowników

```
expr: token operator value

and_expr: expr | con and con [and con]+ | "(" con ")"
or_expr: expr | con or con [or con]+ | "(" con ")"
not_expr: not " " expr
con: and_expr | or_expr | not_expr
token: SINGLE_WORD
value: WORDS
operator: "~" -> like | "=" -> equals | ">" -> greater_than |
"<" -> lower_than
and: "and"
or: "or"
not: "not"
LETTER: /[A-Z]+/
WORDS: /"[^"]+"|[\w]+/
SINGLE_WORD: /[A-Za-z\*\0-9"\-_\]+/
```

Proces tłumaczenia zapytań przypomina algorytm DFS (ang. *deep first search*) w teorii grafów, rozpoczynając od korzenia zapytania aż do uzyskania pojedynczych klauzul. Korzeniem zapytania najczęściej jest spójnik logiczny występujący w najmniejszej liczbie nawiasów.

Aplikacja stworzona na potrzeby pracy magisterskiej nie umożliwia sortowania po innych atrybutach niż domyślnie wykorzystywane przez silniki bazodanowe. W przypadku kontynuowania prac nad aplikacją powinna zostać ona rozszerzona o tę funkcjonalność.

5.3 Aktualizowanie rekordów

W przypadku korzystania z baz relacyjnych podczas aktualizowania rekordów można wykorzystać właściwości przetwarzania transakcji aby zapewnić spójność danych. W przypadku modelu dobrze znormalizowanego wystarczy zaktualizować dane w jednym miejscu w bazie, najczęściej w jednej tabeli. Wszystkie rekordy referujące do zaktualizowanej wartości powinny odwoływać się do niej wykorzystując sekwencyjny klucz główny, nie wymagając żadnych aktualizacji po drugiej stronie relacji. Od momentu potwierdzenia przez silnik bazodanowy wykonania operacji, użytkownicy mogą wyszukiwać po zaktualizowanej wartości [27].

Znacznie bardziej skomplikowane jest aktualizowanie rekordów w bazie gdzie esencją jest wykorzystanie denormalizacji i redundancji danych. W tym modelu danych wszystkie

informacje dotyczące encji są przechowywane w jednym dokumencie. W momencie zmiany wartości rekordu występującego w wielu krotkach, np. zmiany nazwy kategorii książki, należy zaktualizować wszystkie rekordy korzystające z tej wartości. W momencie gdy baza danych zawiera setki lub tysiące rekordów, taka operacja może zostać wykonana w trakcie trwania zapytania. Znacznie bardziej problematyczna jest sytuacja gdy baza danych aplikacji przechowuje dziesiątki lub setki milionów wierszy. Aktualizacja dotycząca znacznej części rekordów może w takim przypadku zająć nawet kilka minut, co dyskwalifikuje wykonywanie jej w trakcie trwania zapytania HTTP. Znacznie lepszym pomysłem jest zaplanowanie asynchronicznego zapytania oraz zwrócenie użytkownikowi informacji, że zmiany nie będą natychmiastowo widoczne.

W aplikacji stworzonej przez magistranta wykorzystany został wzorzec *Master-Worker* [28]. Zakłada on planowanie zadań, na przykład poprzez umieszczanie ich w kolejce, przez wątki aplikacji obsługujące ruch HTTP oraz wykonywanie ich przez wątki pracujące w tle. W aplikacji wykorzystano kolejkę przechowującą dane w pamięci operacyjnej, jednak w przypadku rozbudowy aplikacji oraz wdrożenia jej na zdalny serwer należałoby rozpatrzyć wsparcie dla bardziej rozbudowanych rozwiązań, takich jak na przykład Amazon SQS [29]. Kolejki rozproszone zapewniają znacznie większą niezawodność, gwarantując brak utraty danych nawet w przypadku krytycznych błędów aplikacji.

Ponadto, silnik Redisearch nie daje gwarancji uzyskania natychmiastowej spójności po zakończeniu zapytania aktualizującego. Dane po upływie pewnego czasu uzyskają spójność, proces ten nazywany jest ostatecznie spójnym (ang. *eventually consistent*)

5.4 Porównanie spójności obu rozwiązań

Oba silniki wyszukiwania posiadają swoje unikatowe cechy, takie jak sposoby tokenizacji, stemmingu czy rodzaje wyszukiwania pełnotekstowego. Naturalne więc jest, że identyczne zapytanie wysłane przez użytkownika skutkować będzie zwróceniem różnych wyników zapytania. Różnice te mogą wydawać się nieistotne w momencie tworzenia aplikacji, gdyż użytkownicy dopiero będą poznawać jej sposób działania. Jednak zmiana typu bazy danych w aplikacji obsługującej ruch produkcyjny może być dużo bardziej problematyczna. Użytkownicy mogą być zaskoczeni innymi rezultatami swoich zapytań. Co więcej, automatyzację wykorzystującą naszą aplikację, a w szczególności REST API, mogą zacząć działać niepoprawnie. Aplikacje typu *enterprise* często oferują możliwość tworzenia wtyczek,

które oparte są na rzeczonym REST API, rozszerzających funkcjonalności aplikacji. Przykładem może być portal *Atlassian Marketplace* [30], w którym znajdziemy tysiące produktów poszerzających możliwości aplikacji firmy Atlassian, takich jak Jira czy Confluence.

Aby zrozumieć wpływ zmian w silniku wyszukiwania na użytkowników możemy skorzystać z wzorca zwanego testami spójności (ang. *consistency checks*) [31]. Na początku trwania testów użytkownik otrzymuje wyniki zwracane przez starą ścieżkę, natomiast nowa ścieżka uruchamiana jest w tle. Porównanie obu podejść jest umieszczane w logach aplikacji, skąd później może być analizowane przez zespoły programistyczne. W momencie potwierdzenia zadowalającej spójności, można bezpiecznie rozpocząć zwracanie wyników z nowej ścieżki wykonania. Wykorzystanie testów spójności zmniejsza liczbę błędów, zwiększa pewność, że zmiana adresuje właściwy problem oraz pozwala na lepszą komunikację nadchodzących zmian.

W aplikacji stworzonej przez magistranta został przygotowany interfejs pozwalający na wykonanie testów spójności. Wyniki zwracane przez silnik PostgreSQL oraz Redisearch są przekształcane do wspólnej klasy, aby umożliwić porównanie wyników.

6. TESTY WYDAJNOŚCIOWE

Wydajność w aplikacjach rozproszonych najczęściej wyrażana jest za pomocą liczby zapytań jakie mogą być przez nią obsłużone w jednostce czasu lub percentyli czasu odpowiedzi. Wynik testów przedstawione w tym rozdziale będą korzystały z drugiej metody, z powodu większej popularności w ostatnich latach.

6.1 Środowisko testowe

Testy zostały przeprowadzone na lokalnej maszynie, aby wyeliminować wpływ sieci na wyniki. Specyfikacja komputera wykorzystanego do przeprowadzenia testów:

MacBook Pro (15 cali, 2018)

Processor: 2,6 GHz 6-Core Intel Core i7

32 GB 2400 MHz DDR4

System operacyjny: macOS Catalina, wersja 10.15.7

Silniki baz danych były uruchamiane w kontenerach Dockerowych, wykorzystując następujące obrazy:

- PostgreSQL - postgres w wersji 13.2 [32]
- Redisearch - redislabs/redisearch w wersji 2.0.6 [33]

Do symulowania ruchu produkcyjnego został wykorzystany program Locust [34]. Jest on prostym narzędziem do wykonywania testów obciążeniowych (ang. *load testing*) wykorzystującym język Python. Dzięki wykorzystaniu czytelnego języka programowania, zamiast na przykład notacji XML, próg wejścia dla tego narzędzia jest relatywnie niski.

Listing 6.1 Definicja klasy testowej wykorzystująca dwa typy zadań

```
class PerfTestUser(HttpUser):
    wait_time = between(1, 2.5)

    @task
    def fts_nosql(self):
        headers = {'content-type': 'application/json'}

        self.client.post("books?engine=nosql", data=json.dumps({
            "query": next(ph_city_query()),
            "page_number": 1,
            "page_size": 10,
        }
        ),
            headers=headers,
            name="NoSQL - city") \

    @task
    def fts_sql(self):
        headers = {'content-type': 'application/json'}
        self.client.post("books?engine=sql", data=json.dumps({
            "query": next(ph_city_query()),
            "page_number": 1,
            "page_size": 10,
        }
        ),
            headers=headers,
            name="SQL - city")
```

Na listingu 6.1 została przedstawiona jedna z klas użytych podczas testów wydajnościowych. Wystarczające jest, aby implementacja dziedziczyła po klasie *HttpUser* oraz posiadała metody oznaczone adnotacją *@task*. Każdy użytkownik utworzony przez narzędzie Locust będzie wywoływał jedną z tych metod, wykonując jeden scenariusz testów wydajnościowych.

6.2 Założenia testów

Bazy danych zostały wypełnione 200 000 wpisów dla tabeli *book*, aby zasymulować ilość danych dostępną na środowisku produkcyjnym. Ponadto, atrybuty przechowywane w osobnych tabelach w bazie relacyjnej także posiadają licznosci na poziomie setek lub tysięcy. Dane umieszczone w obu bazach były identyczne, aby zapewnić jak najlepszą porównywalność testów.

W przypadku testów wyszukiwania liczba współbieżnych użytkowników została ustalona na 50, a liczba zapytań do każdego z silników na 10 000. Dzięki temu możliwe jest wyciągnięcie poprawnych statystycznie wniosków, zmniejszając wpływ szumu na rezultaty.

6.3 Wydajność wstawiania wierszy

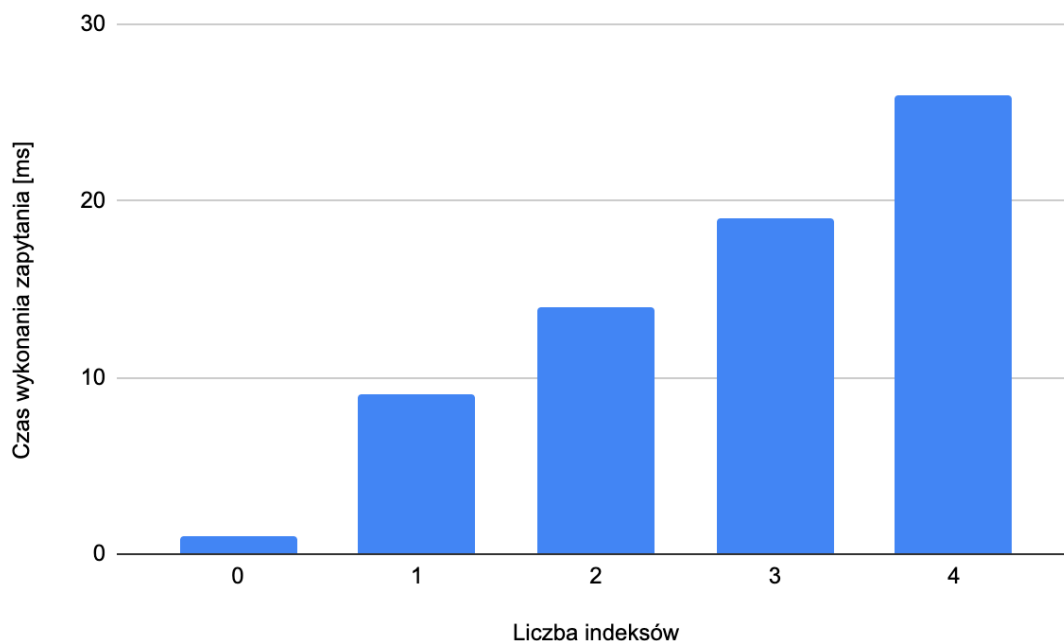
Do zmierzenia kosztu wstawiania wierszy w silnikach baz danych wykorzystane zostało zadania inicjalnej populacji danych. Do przygotowania danych zastosowano bibliotekę napisaną w języku Python o nazwie *faker* [35]. Pozwoliło to na jak najlepsze odzwierciedlenie danych produkcyjnych oraz wykorzystanie funkcjonalności silników w aspektach indeksowania tekstu oraz innych atrybutów. Wyniki przedstawione w tabeli 6.1, dla bazy relacyjnej, uwzględniają czas wstawiania do wszystkich tabel. W przypadku bazy NoSQL wszystkie dane dotyczące książek zostały umieszczone w jednym indeksie, więc koszt wstawienia ich równy jest kosztowi populacji całej bazy.

Tabela 6.1 Wydajność wstawiania wierszy (w sekundach) dla dwóch typów baz danych, mniej=lepiej

Liczba wierszy / typ bazy danych	NoSQL	SQL
10 000	21 s	11 s
50 000	105 s	60 s
100 000	210 s	110 s
200 000	380 s	230 s

6.4 Wydajność aktualizacji

Głównym czynnikiem wpływającym na wydajność aktualizacji istniejących wierszy dla baz relacyjnych jest ilość indeksów założonych na danej kolumnie. Jeden atrybut tabeli może być wykorzystywanych w kilku, a nawet w kilkunastu indeksach. Wykorzystanie tylko indeksów jednokolumnowych nie sprawdzi się w przypadku zapytań wykorzystujących kilka atrybutów danej tabeli. W przypadku gdy aplikacja pozwala użytkownikom na dużą dowolność przy tworzeniu zapytań, w szczególności gdy wspiera swój własny język zapytań, dobre pokrycie tabeli indeksami może stać się konieczne z punktu widzenia wydajności wyszukiwania. Każdy dodatkowy indeks to dodatkowy koszt dla silnika bazodanowego. Podczas wstawiania lub aktualizacji wiersza system zarządzania bazą danych musi nie tylko zaktualizować daną tabelę, ale także każdy indeks zawierający zmieniony atrybut. Jako, że większość indeksów wykorzystuje posortowane struktury drzewiaste, koszt aktualizacji indeksu może być dużo większy niż koszt aktualizacji tabeli. Spowodowane jest to potrzebą znalezienia odpowiedniego liścia oraz potencjalną koniecznością przebudowy drzewa [36]. Rycina 6.1 pokazuje wpływ liczby indeksów na tabeli na czas wstawiania lub aktualizacji rekordów w bazie relacyjne



Rys 6.1 Wpływ liczby indeksów na wydajność aktualizacji w bazie relacyjnej, mniej=lepiej

Silnik Redisearch nie pozwala na tworzenie struktur podobnych do indeksów w świecie relacyjnym. System ten posiada dużo większą autonomię niż bazy obiektowo-relacyjne, niejako decydując o wewnętrznej strukturze dokumentu samodzielnie. Jak wyjaśniono w poprzednich rozdziałach, kluczowe jest zbudowanie odpowiedniej struktury dokumentu na początkowych etapach prac, gdyż jakiegokolwiek zmiany poczynione na bazie produkcyjnej wymagają przebudowy indeksu. Może to skutkować koniecznością ograniczenia dostępu użytkowników do aplikacji, zmniejszając dostępność systemu.

Tabela 6.2 podsumowuje wpływ indeksów na aktualizowanie wierszy w tabeli book dla bazy relacyjnej. Wykonane zostały testy dla przypadku używania tylko indeksów na kolumnach wspierających wyszukiwanie pełnotekstowe, używania optymalnej liczby indeksów dla wydajności wyszukiwania (liczba ta została ustalona podczas testów wydajnościowych) oraz z nadmiarowymi indeksami.

Tabela 6.2 Wydajność aktualizowania wierszy (w sekundach) dla bazy relacyjnej w zależności od liczby indeksów, mniej=lepiej

Liczba wierszy	Tylko indeksy pełnotekstowe	Optymalna liczba indeksów (cztery)	Nadmiarowa liczba indeksów (osiem)
1 000	1 s	3 s	5 s
10 000	11 s	24 s	31 s
50 000	48 s	127 s	195 s

6.5 Wydajność wyszukiwania

Aplikacja stworzona na potrzeby pracy magisterskiej jest mocno zorientowana na optymalizację odczytów. Zakładana liczba odczytów danych znacznie przewyższa liczbę operacji zapisu. Podobny scenariusz jest obecny w wielu popularnych aplikacjach internetowych takich jak Netflix, gdzie 90% operacji na bazie danych stanowią odczyty [37], czy Legimi, gdzie użytkownicy nie mogą edytować danych książek, więc biznesowe przypadki użycia są skupione wokół ich wyszukiwania. Z danych dostępnych w internecie możemy wywnioskować, że około 90% operacji dla portalu Netflix to odczyt danych.

6.5.1 Wyszukiwanie pełnotekstowe

Pierwszy z eksperymentów wydajnościowych został oparty o wyszukiwanie pełnotekstowe, zapytania kierowane do aplikacji rozproszonej zawierały od 4 do 6 klauzuli wykorzystujących ten typ wyszukiwania. Wyszukiwane słowa były losowo wybierane z listy najpopularniejszych słów użytych w danej kolumnie oraz ogólnej listy słów, celem wprowadzenia większej rozbieżności testów. Listing 6.2 prezentuje przykładowe zapytanie użyte podczas testów.

Listing 6.2 Przykładowe zapytanie pełnotekstowe

```
title ~ bank OR
(title ~ less AND title ~ space) OR
(description ~ near AND description ~ blood)
```

Tabela 6.3 Wyniki testów wydajnościowych dla wyszukiwania pełnotekstowego, mniej=lepiej

Percentyl / silnik	NoSQL	SQL
p50	22 ms	120 ms
p75	32 ms	150 ms
p90	46 ms	190 ms
p95	57 ms	220 ms
p99	86 ms	290 ms
p99.9	120 ms	400 ms

Wyniki przedstawione w tabeli 6.3 uwypuklają różnice wydajności wyszukiwania pełnotekstowego pomiędzy zastosowanymi silnikami wyszukiwania. Kolejne wiersze przedstawiają wyniki na danych percentylach, przykładowo wartość p50 to wartość środkowa, inaczej zwana medianą. Powyżej i poniżej mediany znajduje się zawsze jednakowa liczba odczytów. Najwyższe percentyle pokazują wydajność dla najwolniejszych obserwacji, przykładowo wartość p99 na poziomie 86 ms oznacza, że 99% zapytań było szybszych niż ta wartość, a 1% było od niej wolniejszych.

Różnice w wydajności zapytań w tym eksperymencie widoczne są już na poziomie mediany, na wysokich percentylach różnice względne są niższe niż przy medianie, natomiast różnice bezwzględne są bardzo wysokie. Na poziomie p99.9 różnica ta wynosi prawie 300 ms.

6.5.2 Wyszukiwanie prefiksowe

Wyszukiwanie prefiksowe (ang. prefix search) polega na użyciu tylko przedrostka szukanej frazy. Silnik wyszukiwania ma za zadanie znaleźć wszystkie wartości, dla których przedrostek pasuje do jednego z wyrazów użytych w danym atrybucie tekstowym. Przykładowo, zapytanie o prefiks “dom” powinno zwrócić wpisy takie jak: “dom”, “domowy”, “domostwo”, “domino”, “Dominikana”, itp. Jak łatwo zauważyć, wyszukiwanie po przedrostku jest dużo szerszym pojęciem niż tokenizacja i stemming wyrazów. W przypadku zastosowania tylko tych procesów, nie zostałyby znalezione frazy “domino” i “Dominikana”, ponieważ posiadają one inny temat wyrazu niż słowo “dom”.

Warto zauważyć, że Redisearch domyślnie wspiera wyszukiwanie prefiksowe tylko dla fraz zawierających przynajmniej dwie litery [17]. To ustawienie może być zmienione, co wpłynie na zwiększenie koszt zapytania. PostgreSQL nie posiada takiego ograniczenia, jednak dla spójności wyników przeprowadzone testy korzystały z fraz składających się z przynajmniej 2 znaków. Struktura zapytania jest analogiczna do poprzedniego testu, przykładowe zapytanie jest widoczne na listingu 6.3.

Listing 6.3 Przykładowe zapytanie wykorzystujące wyszukiwanie prefiksowe

```
title ~ "win*" AND description ~ "lap*" OR
(title ~ on AND title ~ "win*" AND (description ~ material AND
description ~ "buy*))
```

Tabela 6.4 prezentuje wyniki testów wydajnościowych dla wyszukiwania pełnotekstowego z wykorzystaniem wyszukiwania prefiksowego dla części klauzul.

Tabela 6.4 Wyniki testów wydajnościowych dla wyszukiwania pełnotekstowego wraz z wyszukiwaniem prefiksowym, mniej=lepiej

Percentyl / silnik	NoSQL	SQL
p50	18 ms	110 ms
p75	24 ms	120 ms
p90	34 ms	180 ms
p95	47 ms	280 ms
p99	83 ms	370 ms
p99.9	140 ms	560 ms

Wyniki dla bazy danych typu NoSQL pozostają zbliżone do wyników poprzednich testów, różnice nie przekraczają 10%. Podobnie jest dla bazy relacyjnej, ale tylko poniżej 95. percentyla. Na poziomie p99.9 zapytania stały się wolniejsze o 40%. Pokazuje to, że dla tego typu bazy i typu indeksu umożliwiającego wyszukiwanie pełnotekstowe, wyszukiwanie po prefiksach stanowi znaczący koszt.

6.5.3 Wyszukiwanie po atrybutach tekstowych i numerycznych

Tak jak opisano w poprzednim rozdziale, model bazy relacyjnej wykorzystuje relacje jeden do wielu pomiędzy tabelami `book` i `publishing_house` oraz `publishing_house` i `city`. Wyszukanie książek których wydawnictwo mieści się w danym mieście wymaga więc zastosowania podwójnego złączenia, co wpływa na skomplikowanie zapytania. Listing 6.4 pokazuje przykładowe zapytanie w języku SQL, które dla zastosowanego modelu danych, pozwala na wyszukiwanie książek, których wydawnictwo zlokalizowane jest w mieście “Warszawa”.

Listing 6.4 Przykładowe zapytanie w języku SQL wykorzystujące podwójne złączenie

```
SELECT book.id
FROM book
      JOIN publishing_house ON book.publishing_house_id =
publishing_house.id
      JOIN city ON publishing_house.city_id = city.id
WHERE city.city = 'Warsaw'
```

W modelu nierelacyjnych wszystkie dane książki umieszczone są w tym samym indeksie, więc zapytanie wyszukujące po mieście wydawnictwa nie różni się od zapytania szukającego po tytule czy numerze ISBN.

W tabeli 6.5 przedstawiono rezultaty testów wydajnościowych. Dla bazy danych typu NoSQL wyniki są dwu- lub trzykrotnie lepsze niż dla bazy relacyjnej. Mimo to, obie bazy zwracają wyniki poniżej 0.1 sekundy nawet dla najwolniejszego percentyla zapytań. Podwójne złączenie nie sprawia dużych problemów bazie typu SQL.

Tabela 6.5 Wyniki testów wydajnościowych dla wyszukiwania po mieście wydawnictwa, mniej=lepiej

Percentyl / silnik	NoSQL	SQL
p50	8 ms	20 ms
p75	9 ms	22 ms
p90	11 ms	26 ms
p95	13 ms	29 ms
p99	19 ms	41 ms
p99.9	44 ms	92 ms

Po wykonaniu powyższego eksperymentu, rozszerzono zapytania testowe tak, aby korzystały one z innych atrybutów książki. Listing 6.5 prezentuje przykładowe zapytanie wykorzystane w trakcie testów.

Listing. 6.5 Przykładowe zapytanie wykorzystujące wyszukiwania po atrybutach tekstowych i numerycznych

```
pages > 887 OR
(author = "Joseph Mooney" AND pages < 912) OR
(publishing_house_city = "Warsaw" OR publishing_house_city =
"West Robert") OR
publishing_house = "Garcia, Sanders and Johnson"
```

Tabela 6.6 Wyniki testów wydajnościowych dla wyszukiwania po atrybutach tekstowych i numerycznych, mniej=lepiej

Percentyl / silnik	NoSQL	SQL (bez indeksów)	SQL (z optymalnymi indeksami)
p50	12 ms	1400 ms	28 ms
p75	16 ms	1800 ms	28 ms
p90	22 ms	2300 ms	43 ms
p95	27 ms	2600 ms	51 ms
p99	45 ms	3400 ms	73 ms
p99.9	61 ms	4400 ms	110 ms

W tabeli 6.6 przedstawiono wyniki wydajnościowe dla zapytań wykorzystujących atrybuty numeryczne oraz tekstowe. Dopiero po dodaniu optymalnego zestawu indeksów, które przedstawia listing 6.6, czasy odpowiedzi dla PostgreSQL zbliżyły się do wyników Redisearcha. Wyniki bez zastosowania indeksów wynosiły powyżej sekundy już na medianie, mocno wpływając na doświadczenia użytkownika. Baza nierelacyjna znów okazała się szybsza, jednak dzięki zastosowaniu odpowiednich indeksów różnice nie przekraczają 50 ms.

Listing 6.6 Zestaw indeksów dla bazy relacyjnej pozwalający na uzyskanie optymalnych wyników wydajnościowych

```
CREATE INDEX pages_idx ON public.book USING btree (pages);
CREATE INDEX isbn_idx ON public.book USING btree (isbn);
CREATE INDEX pages_author_isbn_idx ON public.book USING btree
(pages, author, isbn);
CREATE INDEX pages_author ON public.book USING btree (pages,
author);
```

6.5.4 Wyszukiwanie z użyciem skomplikowanych zapytań

Celem tego eksperymentu było sprawdzenie jak wzrost ilości klauzul wpłynie na wydajność silników bazodanowych. Przygotowane zapytania miały pomiędzy dziesięć a piętnaście klauzul, wykorzystując wszystkie dostępne atrybuty. Wykorzystano również operator negacji, który nie był używany w poprzednich testach. Listing 6.7 prezentuje przykładowe zapytanie użyte w tym teście.

Listing 6.7 Przykładowe zapytanie wykorzystujące 10 klauzul

```
title ~ near OR
(pages > 688 OR (author = "Patrick Lee" AND pages < 810) OR
(description ~ store AND title ~ "bank*")) OR
(publishing_house_city = "Brianside" OR publishing_house_city =
"South Lisashire" OR (NOT publishing_house = "Lee LLC" AND NOT
publishing_house = "Scott, Randall and Stanley"))
```

Tabela 6.7 Wyniki testów wydajnościowych dla skomplikowanych zapytań, mniej=lepiej

Percentyl / silnik	NoSQL	SQL
p50	72 ms	42 ms
p75	110 ms	58 ms
p90	180 ms	94 ms
p95	230 ms	180 ms
p99	320 ms	240 ms
p99.9	470 ms	320 ms

Typ testów opisywanych w tym podrozdziale to pierwszy przypadek gdy baza relacyjna okazała się szybsza niż baza dla niej konkurencyjna. Tabela 6.7 prezentuje dokładne wyniki pomiarów. Po usunięciu operatora negacji z zapytań, czasy wykonania zapytania w silniku Redisearch zbliżyły się do poziomu wyników PostgreSQL.

6.6 Problem głębokiego stronicowania

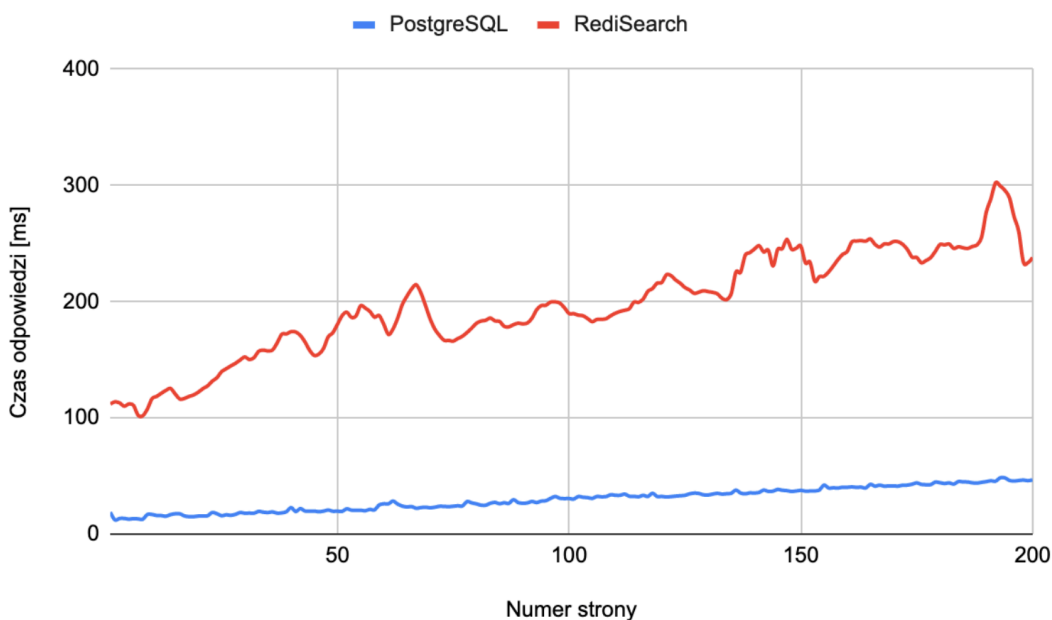
Głębokie stronicowanie (ang. *deep paging*) występuje w sytuacji gdy klient pobiera kolejne strony wyników, np. przeglądając wszystkie oferty dostępne na portalu aukcyjnym. Najprostsza implementacja po stronie bazy danych wykorzystuje operatory OFFSET oraz LIMIT, celem zwrócenia strony rozpoczynającej się od właściwego indeksu oraz posiadającej właściwą liczbę wyników. Przy niskich wartościach parametru OFFSET baza danych zachowują się optymalnie. Bardziej problematyczne są zapytania z wysoką wartością tego parametru. Przykładowo, zapytanie SQL przedstawione na listingu 6.8 skutkuje sortowaniem

dziesięciu tysięcy pięciuset rekordów, aby następnie zwrócić użytkownikowi końcowe pięćdziesiąt.

Listing. 6.8 Zapytanie w języku SQL z wysoką wartością parametru OFFSET

```
ORDER BY created OFFSET 10000 LIMIT 50
```

Na rysunku 6.2 przedstawiona jest wydajność zapytań w stworzonej przez magistranta aplikacji dla rosnących numerów stron. Zapytanie wykorzystane w tym eksperymencie nie wykorzystywało operacji filtrowania, a jedynie sortowanie. Następstwem tego było zwrócenie wszystkich dwustu tysięcy książek, korzystając ze stron o wielkości tysiąca elementów.



Rys. 6.2 Głębokie stronicowanie a czas odpowiedzi

Czasy odpowiedzi PostgreSQL wzrastały liniowo wraz z kolejnymi stronami, zapytanie o dwusetną stronę było dwuipółkrotnie wolniejsze od zapytania o początkowe strony. W przypadku RedisSearcha, czasy odpowiedzi były znacznie mniej stabilne niż w przypadku bazy relacyjnej. Co ciekawe, zapytania o ostatnie trzy strony były o 25% szybsze niż o strony poprzedzające. Czasy odpowiedzi wzrastały szybciej niż w przypadku bazy relacyjnej, a najwolniejsze zapytania były ponad trzy razy wolniejsze od początkowych.

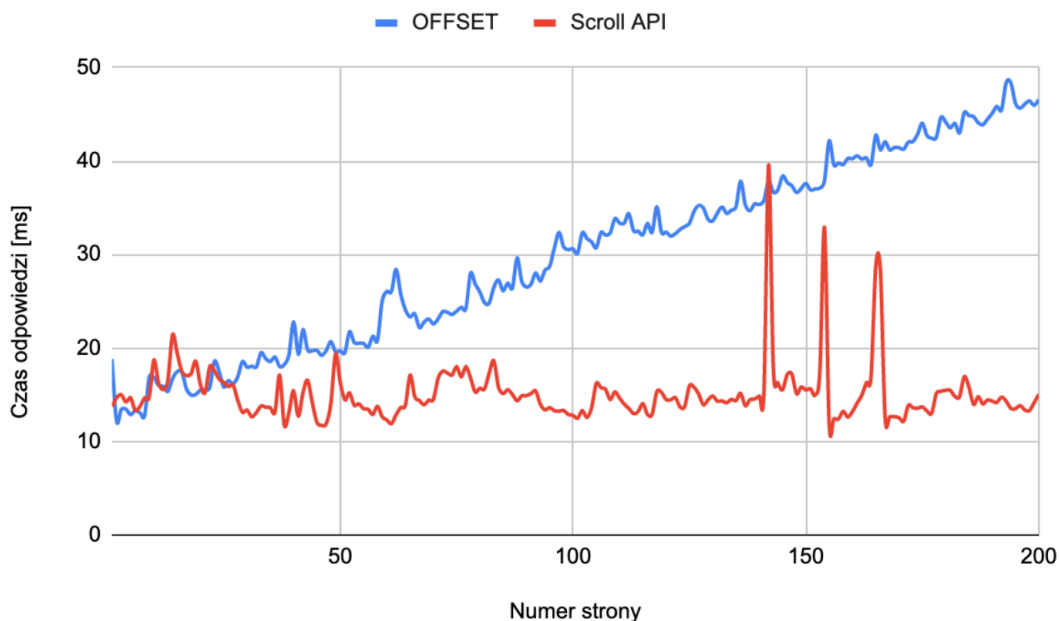
Alternatywą do stronicowania za pomocą operatorów LIMIT i OFFSET jest dodanie klauzuli ograniczającej zapytanie za pomocą unikatowego atrybutu. W przypadku aplikacji

stworzonej przez magistranta idealnym kandydatem okazał się klucz główny w modelu relacyjnym, a mianowicie atrybut *id*. W przypadku tej metody, zwanej *Scroll API*, operator OFFSET zastąpiony jest dodatkową klauzulą, natomiast zastosowanie parametru LIMIT pozostaje bez zmian [39]. Listing 6.9 prezentuje przykładowe zapytanie wykorzystujące tą metodę (wykorzystanie operatora OFFSET zostało przedstawione na listingu 6.8). Zapytanie to pobiera maksymalnie 50 rekordów, zaczynając od rekordu z atrybutem *id* równym 1001. To zapytanie mogłoby zostać wykorzystane do pobrania strony numer 21 (dla liczby rekordów na stronie = 50).

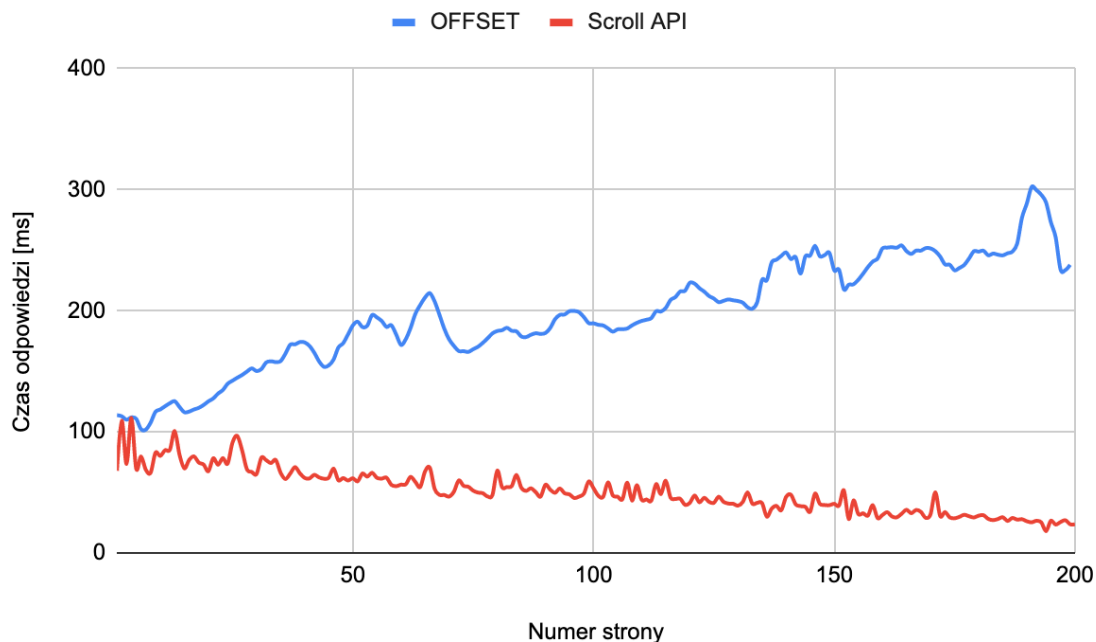
Listing. 6.9 Zapytanie w języku SQL wykorzystujące Scroll API

```
SELECT * FROM book
WHERE id > 1000
ORDER BY id
LIMIT 50
```

Na rysunkach 6.3 i 6.4 przedstawiono różnice pomiędzy czasami odpowiedzi w stronicowaniu za pomocą obu metod.



Rys. 6.3 Czas odpowiedzi (w milisekundach) przy stronicowaniu z wykorzystaniem Scroll API oraz operatora OFFSET dla PostgreSQL, mniej=lepiej



Rys. 6.4 Czas odpowiedzi (w milisekundach) przy stronicowaniu z wykorzystaniem Scroll API oraz operatora OFFSET dla RediSearcha, mniej=lepiej

W przypadku obu silników czasy odpowiedzi były znacząco lepsze dla metody wykorzystującej Scroll API. Dla RediSearcha zauważalna jest tendencja spadkowa wraz ze wzrostem numeru strony, natomiast dla PostgreSQL czasy pozostały na stałym poziomie.

Wykorzystanie Scroll API wymaga dokonania zmian w projekcie aplikacji rozproszonej. Bezstanowe aplikacje nie pozwalają na utrzymanie sesji pomiędzy zapytaniami, więc użytkownik jest odpowiedzialny za przekazanie indeksu ostatniego elementu. Przy korzystaniu z aplikacji webowej informacja ta zostanie przed użytkownikiem ukryta, a jej wysyłanie zautomatyzowane. W przypadku REST API użytkownik jest zmuszony sam zadbać aby indeks ostatniego elementu został poprawnie uwzględniony w kolejnych pytaniach. Problematiczne mogą okazać się sytuacje, gdy użytkownik nie przechodzi przez wszystkie dostępne strony. Przeskoczenie jednej ze stron wiąże się z koniecznością zastosowania metody wykorzystującej operator OFFSET lub wykonania dodatkowego zapytania do bazy danych celem znalezienia ostatniego elementu z poprzedzającej strony. Oba rozwiązania negatywnie wpływają na czas wykonania zapytania w aplikacji serwerowej.

6.7 Wnioski

Przeprowadzone testy wydajnościowe pokazały, że nierelacyjna baza dokumentowa gwarantuje lepsze czasy odpowiedzi dla większości typów zapytań. Największe różnice na korzyść RediSeacha uwidaczniają testy wykorzystujące wyszukiwanie pełnotekstowe. Aplikacja często wykorzystujące ten typ zapytań powinny rozważyć zastosowanie tego systemu zarządzania bazą danych.

Nie wszystkie przeprowadzone testy pokazały wyższość baz dokumentowych nad bazami relacyjnymi. W przypadku skomplikowanych zapytań czy głębokiej paginacji lepszy okazał się PostgreSQL.

Testy wydajnościowe przeprowadzone przez magistranta potwierdziły, że nie istnieje jedna baza danych idealna dla każdego przypadku użycia. Typ zapytań, wielkość ruchu oraz model danych powinny być czynnikami decyzyjnymi w momencie wyboru optymalnego typu bazy danych. Podsumowanie testów jest przedstawione w tabeli 6.8

Tabela 6.8 Podsumowanie wyników testów wydajnościowych

Rodzaj testu	Typ bazy danych, która uzyskała lepsze rezultaty	Uwagi
wstawianie wierszy	SQL	Dzięki użyciu normalizacji, wstawianie wierszy w bazie relacyjnej było prawie dwukrotnie szybsze.
wyszukiwanie pełnotekstowe	NoSQL	We wszystkich trzech przypadkach, Redisearch okazał się dużo lepszym systemem, różnice były szczególnie widoczne na wyższych percentylach, gdzie czasy odpowiedzi potrafiły być ponad czterokrotnie lepsze (83 ms w porównaniu do 430 ms na p99 dla wyszukiwania prefiksowego).
wyszukiwanie prefiksowe	NoSQL	
wyszukiwanie po atrybutach tekstowych i numerycznych	NoSQL	
wyszukiwanie z użyciem skomplikowanych zapytań	SQL	Ten test uwidoczniał problemy obu baz danych, a więc pogorszenie wydajności wraz ze wzrostem skomplikowania zapytań. Ostateczne rezultaty były nieznacznie lepsze w przypadku użycia bazy relacyjnej.
głębokie stronicowanie	SQL	Dla Redisearcha głębokie stronicowanie okazało się bardzo dużym problemem, czas dostępu do końcowych stron był trzykrotnie wyższy niż do stron początkowych. W przypadku Postgresa różnice względne też były duże, aczkolwiek czasy odpowiedzi pozostały na znacznie niższym poziomie. Użycie Scroll API w obu przypadkach pozytywnie wpłynęło na wyniki, utrzymując czasy odpowiedzi na stałym poziomie.

7. WPLYW NA ODCZUCIA UŻYTKOWNIKA

Większość dzisiejszych aplikacji stara się ukryć przed użytkownikami jak najwięcej technicznych szczegółów. Jednak wybór technologii niesie za sobą konsekwencje dla użytkownika, często wpływając bezpośrednio na jego odczucia podczas użytkowania aplikacji. Nie wszystkie funkcjonalności są możliwe do implementacji w każdej technologii, a te, które będą możliwe, niekoniecznie będą działać identycznie w przypadku różnych rozwiązań. Poniższy rozdział opisuje wpływ wyboru technologii bazodanowej na końcowego użytkownika aplikacji.

7.1 Spójność danych

W systemach opartych na bazach wspierających reguły ACID, czyli najczęściej w bazach relacyjnych, zarządzanie aktualizacjami danych jest relatywnie proste. Większość operacji na danych może odbywać się w trakcie zapytania HTTP, gdyż aktualizację dotyczą zazwyczaj pojedynczych rekordów w bazie danych. W momencie zakończenia transakcji bazodanowej, użytkownik ma pewność, że wszystkie dane zostały zaktualizowane poprawnie. Zaraz po zakończeniu zapytania zawierającego aktualizacje danych, użytkownik może wysłać zapytanie wyszukujące nową wartość. Gwarancja otrzymania spójnych rezultatów w tym przypadku nazywana jest spójnością odczytu po zapisie (ang. *read-after-write consistency*). Co ciekawe, paradygmat ten nie zapewnia, że inni użytkownicy systemu będą widzieć daną aktualizację. Zapewnienie spójności dla wszystkich użytkowników systemu od momentu zakończenia zapytania jest nazywane silną spójnością (ang. *strong consistency*) [40].

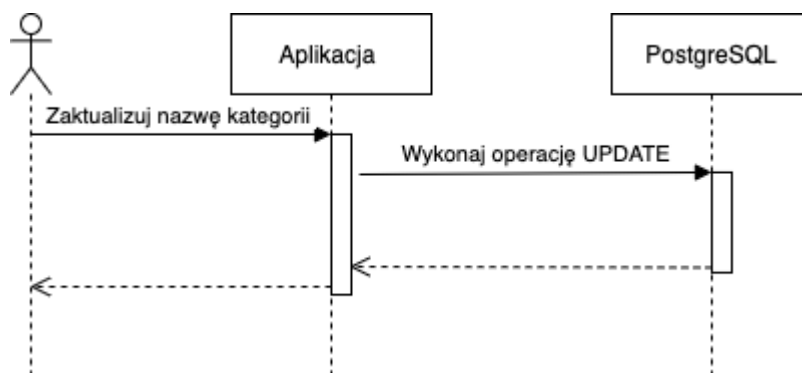
W opozycji do silnej spójności stoją rozwiązania wspierające ostateczną spójność (ang. *eventual consistency*). W tym przypadku użytkownik ma teoretyczną gwarancję, że dane po pewnym czasie uzyskają spójność, jednak (przeważnie) nie ma żadnych gwarancji czasowych. W ogólności, rozproszone systemy bazodanowe, w których skład wchodzi wiele węzłów preferują ten paradygmat. Wewnętrzne zarządzanie spójnością danych jest znacznie ułatwione, ponieważ wystarczy, że część węzłów potwierdzi otrzymanie aktualizacji. Jeśli któryś z węzłów jest w danym momencie przeciążony ilością operacji, system nie musi czekać na odpowiedź zwrotną od niego. Wystarczająca jest gwarancja, że dana operacja zostanie wykonana w najbliższym czasie. Przy systemach wspierających ostateczną spójność możliwa jest sytuacja, w której część zapytań zwraca zaktualizowane rekordy, a część przestarzałe dane. Dzieje się tak,

ponieważ zapytania mogą trafiać na różne węzły będące w różnych stanach. Możliwy jest zatem następujący scenariusz:

- Użytkownik aktualizuje swój profil
- Po przeładowaniu strony ukazują mu się zaktualizowane dane
- Po ponownym załadowaniu strony, serwer zwraca przestarzałe dane
- Kolejne załadowanie powoduje zwrócenie zaktualizowanych rekordów

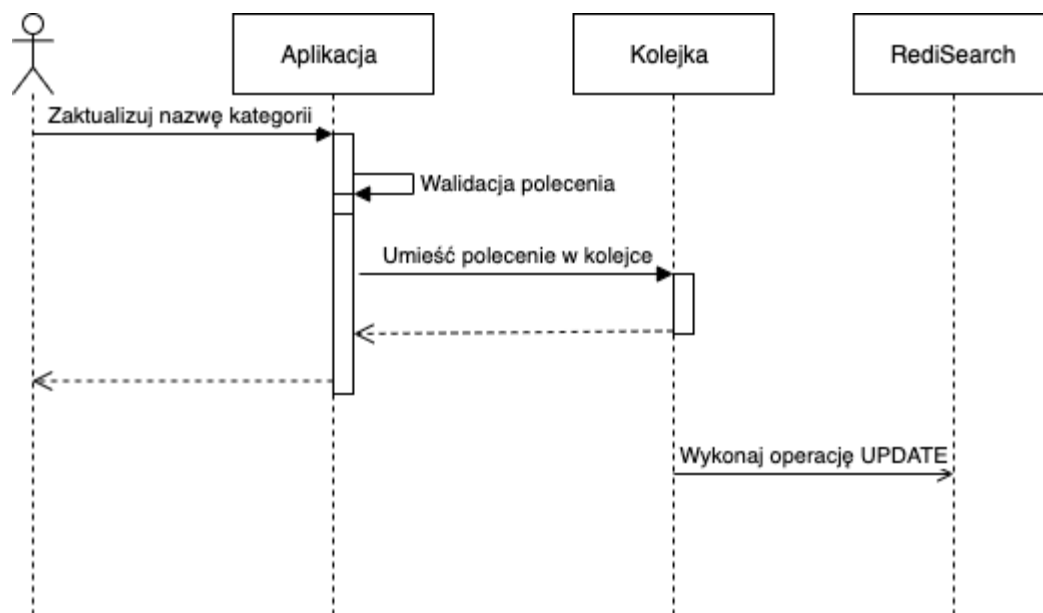
7.1.1 Scenariusz aktualizacji danych

Jak zostało opisane w rozdziale piątym, część atrybutów książki została umieszczona w dedykowanych tabelach w modelu relacyjnych, przykładowo atrybuty *category* oraz *tag*. W przypadku Redisearcha wszystkie atrybuty książki zostały umieszczone w głównym dokumencie. Rysunek 7.1 przedstawia diagram sekwencji dla aktualizacji jednego z atrybutów książki dla bazy relacyjnej.



Rys 7.1 Diagram sekwencji aktualizacji atrybutu książki dla bazy relacyjnej

Jak można zaobserwować na diagramie, przepływ sterowania powróci do użytkownika po otrzymaniu odpowiedzi od bazy danych. Daje to użytkownikowi gwarancję, że wszystkie odczyty wykonane po otrzymaniu odpowiedzi zwrócą aktualną wartość. Rysunek 7.2 prezentuje identyczny scenariusz dla bazy NoSQL.



Rys. 7.2 Diagram sekwencji aktualizacji atrybutu książki dla bazy typu NoSQL

W przypadku bazy NoSQL diagram sekwencji jest znacznie bardziej skomplikowany niż w przypadku bazy relacyjnej. Jako, że operacja aktualizacji musi zostać wykonana asynchronicznie, zapytanie użytkownika jest umieszczane w kolejce, tak aby mogło zostać później przetworzone przez wątki działające w tle. Dodatkowo, zapytanie to powinno być poddane walidacji przed umieszczeniem w kolejce. Dzięki temu ewentualne błędy poprawności zapytania mogą być zwrócone użytkownikowi już po etapie walidacji. W innym przypadku, błędy zostaną wykryte dopiero po otrzymaniu odpowiedzi od bazy danych, gdy aplikacja nie będzie miała już możliwości zwrócenia użytkownikowi odpowiedniego komunikatu.

W przypadku bardziej zaawansowanych, a zarazem długotrwałych operacji przydatne może okazać się zwrócenie użytkownikowi adresu zasobu, pod którym dostępny będzie status zapytania. Dzięki temu, użytkownik aplikacji będzie mógł na bieżąco sprawdzać aktualny stan, a także będzie miał dostęp do większej ilości informacji, gdy zapytanie zakończy się niepowodzeniem.

W aplikacji stworzonej na potrzeby pracy magisterskiej nie został stworzony powyższy mechanizm, jednak w przypadku kontynuacji pracy nad projektem, należałoby rozważyć dodanie tej funkcjonalności.

7.1.2 Problem „hałaśliwych sąsiadów”

Problem hałaśliwych sąsiadów (ang. *noisy neighbours problem*) występuje w przypadku współdzielenia części infrastruktury w architekturze chmurowej przez wielu użytkowników. Problematycznymi zasobami mogą być na przykład kolejki, dysk, operacje Wejścia/Wyjścia. „Hałaśliwy sąsiad” to użytkownik, który wykorzystuje znaczną część zasobów, tym samym powodując gorszą ich dostępność dla innych użytkowników. Najczęstszymi objawami są pogorszenie wydajności oraz niezawodności aplikacji [41].

W przypadku aplikacji stworzonej na potrzeby pracy magisterskiej problem ten może wystąpić dla kolejki przechowującej asynchroniczne zadania aktualizacji zasobów. Elementy w kolejce są przetwarzane zgodnie z zasadą „pierwszy na wejściu, pierwszy na wyjściu” (ang. *FIFO - first in, first out*). W sytuacji gdy jeden użytkownik aplikacji zaplanuje bardzo dużą ilość zadań, średni czas przetwarzania zadań dla innych użytkowników znacznie się zwiększy. Użytkownicy nie będą świadomi wydłużonego czasu przetwarzania zapytania, gdyż ilość elementów w kolejce stanowi wewnętrzną informację aplikacji. W przypadku rozszerzenia aplikacji o zasób przechowujący status zapytania, aplikacja mogłaby informować o ilości elementów w kolejce przed danym zadaniem oraz jego prognozowany czas ukończenia.

Dostawcy technologii chmurowych oferują produkty, które pomagają rozwiązywać podobne problemy, na przykład dzięki wykorzystaniu kolejek priorytetowych [42]. Dzięki możliwości ustawienia odpowiedniego priorytetu, kolejne zapytania od jednego użytkownika mogą być depriorytezywane, tak aby zapewnić niski czas przetwarzania dla zapytań od innych użytkowników. Kolejki priorytetowe mają jednak swoją cenę, ich użycie negatywnie wpływa na wydajność przetwarzania [43].

W przypadku najbardziej popularnych kolejek firmy Amazon Web Services, a więc Amazon SQS [28], wykorzystanie priorytetów nie jest wspierane [44]. Rekomendowanym podejściem jest użycie odrębnych kolejek dla grup zadań, przykładowo dla najbardziej aktywnych użytkowników. Takie rozwiązanie wpłynie jednak negatywnie na koszt oraz skomplikowanie aplikacji, gdyż będzie musiała ona równocześnie wspierać wiele kolejek.

7.2 Wspierane funkcjonalności wyszukiwania pełnotekstowego

PostgreSQL jest produktem bardzo uniwersalnym, sprawdzającym się w bardzo szerokim spektrum zastosowań. W momencie jego powstawania, wyszukiwanie pełnotekstowe, szczególnie po bardzo dużych zbiorach danych nie było popularne. Mimo tego funkcjonalności oferowane dzisiaj przez tę bazę danych w zakresie wyszukiwania pełnotekstowego są bardzo szerokie i nie ustępują narzędziom stworzonym ściśle do tego. Co więcej, PostgreSQL pozwala na łatwe tworzenie rozszerzeń i wtyczek [45], umożliwiając poszerzanie wspieranych funkcjonalności.

Przykładowo, wyszukiwanie rozmyte (ang. *fuzzy search*) [46] pozwalające na znalezienie dokumentów zawierających słowa podobne do wyszukiwanego (różniące się kilkoma literami lub zawierające podciąg wyszukiwanych fraz) jest wspierane natywnie w Redisearchu [44]. Natywne wsparcie gwarantuje dobre czasy wyszukiwania oraz wsparcie w przypadku problemów. PostgreSQL nie posiada takiego wsparcia, dostępne są jednak rozszerzenia, które pozwolą osiągnąć zamierzony efekt [48]. Wydajność takich zapytań może okazać się jednak problematyczna, co pokazał już poprzedni rozdział. Co więcej, wiele rozszerzeń nie jest aktywnie rozwijanych, potencjalne problemy czy ograniczenia mogą być więc trudne do rozwiązania.

Każdą z funkcjonalności wspieraną natywnie przez silnik Redisearch można uzyskać w Postgresie korzystając albo z wsparcia samego silnika, albo z dostępnych rozszerzeń. Użytkownicy aplikacji nie powinni więc poczuć różnicy w oferowanych możliwościach wyszukiwania. Używanie rozszerzeń w przypadku bazy relacyjnej niesie jednak dodatkowy koszt, zwiększając skomplikowanie rozwiązania. Brak natywnego wsparcia może być także źródłem problemów wydajnościowych.

8. PODSUMOWANIE

Mimo wzrastającej popularności baz typu NoSQL, w polskojęzycznym Internecie wciąż brakuje publikacji, które rzeczowo opisują różnice pomiędzy dwoma najpopularniejszymi typami baz. Zagadnienie zmiany typu bazy danych skłoniły autora pracy do próby przedstawienia najważniejszych różnic pomiędzy tymi systemami.

Kluczowa w trakcie tworzenia pracy magisterskiej okazała się aplikacja stworzona na jej potrzeby. Pozwoliła ona na praktyczne przedstawienie teoretycznych rozważań zawartych w pracy. Wyniki testów wydajnościowych potwierdziły wyższość baz NoSQL w wyszukiwaniu tekstowym, gdzie czasy na najwyższych percentylach były nawet trzykrotnie lepsze. Mimo bardzo dobrych wyników w tych testach, RediSearch nie uzyskał lepszych rezultatów w każdym ich typie. Skomplikowane zapytania oraz głębokie stronicowanie wypadło lepiej po stronie bazy relacyjnej.

Baza typu NoSQL okazała się świetnym wyborem dla aplikacji często wykorzystującej wyszukiwanie pełnotekstowe. Mnogość funkcji wspieranych natywnie oraz wysoka wydajność, przekłada się na bardzo dobre odczucia z korzystania z systemu. Jednak rozwiązanie to posiada swoje wady oraz wyzwania, największym z pewnością jest zarządzanie spójnością danych. Wybierając silnik bazodanowy bez wsparcia silnej spójności, twórcy aplikacji muszą wziąć pod uwagę jego charakterystykę, odpowiednio dostosowując interakcję wewnątrz aplikacji.

Rozważania przedstawione w pracy magisterskiej stanowią niewielki fragment bardzo szerokiego tematu jakim jest wybór odpowiedniego silnika bazodanowego dla aplikacji rozproszonych. Nie wszystkie tematy udało się autorowi poruszyć, ponadto, głębsze zbadanie części problemów wymaga ruchu produkcyjnego, którego z oczywistych powodów nie udało się uzyskać.

W przypadku kontynuowania pracy nad aplikacją, wartościowe mogłoby okazać się udostępnienie jej w formie biblioteki wolnego oprogramowania. Nowatorskie podejście do problemu wykorzystania dwóch baz typów baz danych przez jedną aplikację mogłoby wpłynąć pozytywnie na dalsze próby zmian typu baz danych w istniejących już aplikacjach.

WYKAZ RYSUNKÓW

Rys. 2.1. Przedstawienie teorii CAP za pomocą diagramu Venna [10]
13

Rys. 2.2 Teoria CAP przedstawiona na modelu trójkąta
15

Rys 4.1. Przykładowy przepływ danych w architekturze REST
24

Rys. 5.1 Model danych dla bazy relacyjnej
30

Rys 6.1 Wpływ liczby indeksów na wydajność aktualizacji w bazie relacyjnej, mniej=lepiej
40

Rys. 6.2 Głębokie stronicowanie a czas odpowiedzi
48

Rys. 6.3 Czas odpowiedzi (w milisekundach) przy stronicowaniu z wykorzystaniem Scroll API oraz operatora OFFSET dla PostgreSQL, mniej=lepiej
49

Rys. 6.4 Czas odpowiedzi (w milisekundach) przy stronicowaniu z wykorzystaniem Scroll API oraz operatora OFFSET dla Redisearcha, mniej=lepiej
50

Rys 7.1 Diagram sekwencji aktualizacji atrybutu książki dla bazy relacyjnej
54

Rys. 7.2 Diagram sekwencji aktualizacji atrybutu książki dla bazy typu NoSQL
55

WYKAZ TABEL

Tabela 4.1 Najpopularniejsze kody odpowiedzi HTTP
25

Tabela 6.1 Wydajność wstawiania wierszy (w sekundach) dla dwóch typów baz danych, mniej=lepiej
39

Tabela 6.2 Wydajność aktualizowania wierszy (w sekundach) dla bazy relacyjnej w zależności od liczby indeksów, mniej=lepiej
41

Tabela 6.3 Wyniki testów wydajnościowych dla wyszukiwania pełnotekstowego, mniej=lepiej
42

Tabela 6.4 Wyniki testów wydajnościowych dla wyszukiwania pełnotekstowego wraz z wyszukiwaniem prefiksowym, mniej=lepiej
43

Tabela 6.5 Wyniki testów wydajnościowych dla wyszukiwania po mieście wydawnictwa, mniej=lepiej
45

Tabela 6.6 Wyniki testów wydajnościowych dla wyszukiwania po atrybutach tekstowych i numerycznych, mniej=lepiej
46

Tabela 6.7 Wyniki testów wydajnościowych dla skomplikowanych zapytań, mniej=lepiej
47

Tabela 6.8 Podsumowanie wyników testów wydajnościowych
52

WYKAZ LISTINGÓW

Listing 3.1 Przykład wyszukiwania danych z użyciem języka SQL	18
Listing 3.2 Przykład wyszukiwania danych spełniających podane kryteria z użyciem języka SQL	18
Listing 3.3 Utworzenie instancji typu tsvector w PostgreSQL	19
Listing 3.4 Wyszukiwanie pełnotekstowe w PostgreSQL	19
Listing. 3.5 Instrukcja utworzenia schematu	20
Listing. 3.6 Dodanie dokumentu do indeksu	21
Listing 3.7 Przykład wyszukiwania danych w indeksie	21
Listing 3.8 Złożone wyszukiwanie danych w indeksie	22
Listing 3.9 Wyszukiwanie z wykorzystaniem prefiksów oraz fuzzy match	22
Listing. 4.1 Prosta aplikacja stworzona z użyciem frameworka Flask	27
Listing. 4.2 Klasa Team wykorzystująca SQLAlchemy	27
Listing 5.1 Definicja indeksu użytego w aplikacji stworzonej przez magistranta	31
Listing 5.2 Proste zapytanie wykorzystujące język zapytań użyty w aplikacji	32
Listing 5.3 Zbiór reguł gramatycznych użytych do tłumaczenia zapytań użytkowników	33
Listing 6.1 Definicja klasy testowej wykorzystująca dwa typy zadań	37
Listing 6.2 Przykładowe zapytanie pełnotekstowe	41
Listing 6.3 Przykładowe zapytanie wykorzystujące wyszukiwanie prefiksowe	43
Listing 6.4 Przykładowe zapytanie w języku SQL wykorzystujące podwójne złączenie	44
Listing. 6.5 Przykładowe zapytanie wykorzystujące wyszukiwania po atrybutach tekstowych i numerycznych	45

Listing 6.6 Zestaw indeksów dla bazy relacyjnej pozwalający na uzyskanie optymalnych wyników wydajnościowych
46

Listing 6.7 Przykładowe zapytanie wykorzystujące 10 klauzul
47

Listing. 6.8 Zapytanie w języku SQL z wysoką wartością parametru OFFSET
48

WYKAZ LITERATURY

- [1] <https://www.statista.com/statistics/1131568/worldwide-popularity-ranking-relational-database-management-systems/> (data dostępu 22.08.2020)
- [2] <http://www.agiledata.org/essays/relationalDatabases.html> (data dostępu 22.08.2020)
- [3] <https://www.essentialsql.com/sql-acid-database-properties-explained/> (data dostępu 22.08.2020)
- [4] https://db-engines.com/en/blog_post/23 (data dostępu 22.08.2020)
- [5] Neal Leavitt. "Will NoSQL Databases Live Up to Their Promise?"
- [6] <https://www.linkedin.com/pulse/why-amazon-google-netflix-facebook-switched-nosql-shannon-block-cfe/> (data dostępu 29.08.2020)
- [7] <https://www.dummies.com/programming/big-data/10-nosql-misconceptions/> (data dostępu 29.08.2020)
- [8] Silberschatz, Avi. Database System Concepts, Sixth Edition
- [9] Brewer, Towards Robust Distributed System, 2000
- [10] <https://medium.com/nerd-for-tech/understand-cap-theorem-751f0672890e> (data dostępu 29.08.2020)
- [11] <https://www.youtube.com/watch?v=hwnNbLXN4vA>
- [12] <https://www.guru99.com/sql-vs-nosql.html> (data dostępu 29.08.2020)
- [13] <https://www.postgresql.org/> (data dostępu 29.08.2020)
- [14] https://redislabs.com/blog/mastering-redisearch-part/?_ga=2.49620966.1740196089.1598980848-1282095969.1598980848 (data dostępu 14.09.2020)
- [15] <https://www.compose.com/articles/mastering-postgresql-tools-full-text-search-and-phrase-search/> (data dostępu 10.06.2021)
- [16] https://oss.redislabs.com/redisearch/Overview/#indexing_documents (data dostępu 14.09.2020)

- [17] https://oss.redislabs.com/redisearch/Query_Syntax/ (data dostępu 25.05.2021)
- [18] <https://medium.com/@ethannam/understanding-the-levenshtein-distance-equation-for-beginners-c4285a5604f0> (data dostępu 10.06.2021)
- [19] Http request methods, <https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods> (data dostępu 14.09.2020)
- [20] Pilgrim M.: Dive into Python (2004)
- [21] Flask Homepage, <http://flask.pocoo.org/> (data dostępu 14.09.2020)
- [22] SQLAlchemy- The Database Toolkit for Python, <https://www.sqlalchemy.org/>
- [23] <https://oss.redislabs.com/redisearch/Commands/#ftsearch> (data dostępu 14.09.2020)
- [24] <https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Overview.DBInstance.html> (data dostępu 21.01.2021)
- [25] <https://lark-parser.readthedocs.io/en/latest> (data dostępu 21.01.2021)
- [26] <https://www.ics.uci.edu/~pattis/misc/ebnf2.pdf> (data dostępu 21.01.2021)
- [27] <https://medium.com/system-design-blog/eventual-consistency-vs-strong-consistency-b4de1f92534d> (data dostępu 21.01.2021)
- [28] <https://java-design-patterns.com/patterns/master-worker-pattern/> (data dostępu 13.03.2021)
- [29] <https://aws.amazon.com/sqs/> (data dostępu 13.03.2021)
- [30] <https://marketplace.atlassian.com/> (data dostępu 13.03.2021)
- [31] <https://www.easytechjunkie.com/what-is-a-consistency-check.htm> (data dostępu 17.05.2021)
- [32] https://hub.docker.com/_/postgres (data dostępu 17.05.2021)
- [33] <https://hub.docker.com/r/redislabs/redisearch> (data dostępu 17.05.2021)
- [34] <https://locust.io/> (data dostępu 17.05.2021)
- [35] <https://faker.readthedocs.io/en/master/> (data dostępu 24.05.2021)
- [36] <https://www.gojek.io/blog/the-postgres-performance-tuning-manual-indexes> (data dostępu 17.05.2021)

- [37] <https://netflixtechblog.com/revisiting-1-million-writes-per-second-c191a84864cc> (data dostępu 25.05.2021)
- [38] https://oss.redislabs.com/redisearch/Query_Syntax/ (data dostępu 25.05.2021)
- [39] <https://blog.jooq.org/2013/10/26/faster-sql-paging-with-jooq-using-the-seek-method/> (data dostępu 27.05.2021)
- [40] <https://avikdas.com/2020/04/13/scalability-concepts-read-after-write-consistency.html> (data dostępu 27.05.2021)
- [41] <https://searchcloudcomputing.techtarget.com/definition/noisy-neighbor-cloud-computing-performance> (data dostępu 27.05.2021)
- [42] <https://aws.amazon.com/blogs/database/implementing-priority-queueing-with-amazon-dynamodb/> (data dostępu 27.05.2021)
- [43] <https://anmolsehgal.medium.com/heap-vs-priority-queues-vs-queues-b03398312c87> (data dostępu 01.06.2021)
- [44] <https://aws.amazon.com/sqs/features/> (data dostępu 01.06.2021)
- [45] <https://www.highgo.ca/2019/10/01/a-guide-to-create-user-defined-extension-modules-to-postgres> (data dostępu 01.06.2021)
- [46] <https://www.techopedia.com/definition/7356/fuzzy-search> (data dostępu 01.06.2021)
- [47] https://oss.redislabs.com/redisearch/Query_Syntax/#fuzzy_matching (data dostępu 01.06.2021)
- [48] <https://www.rdegges.com/2013/easy-fuzzy-text-searching-with-postgresql/> (data dostępu 01.06.2021)