Uniwersytet Mikołaja Kopernika

Wydział Matematyki i Informatyki

Przemysław Piotrowski

nr albumu: 260141

Informatyka

Praca inżynierska

Bazy NoSQL w kontekście serwerów aplikacji webowych

Opiekun pracy dyplomowej
prof. dr hab. Krzysztof Stencel

Toruń 2016

Spis treści

Streszczenie 5

Wstęp 6

1. O bazach danych 8

1.1 Typy baz danych i ich popularność 8

1.2 Pojęcia bazodanowe 9

1.3 SQL 13

1.4 NoSQL 16

1.5 Typy baz NoSQL 16

1.5.1 Bazy klucz-wartość 17

1.5.2 Dokumentowe bazy danych 18

1.5.3 Bazy zorientowane kolumnowo 19

1.5.4 Grafowe bazy danych 20

2. Sposób przeprowadzania badań 22

2.1 Technologie 22

2.2 Pomiar wydajności 24

2.3 Sprzęt 25

2.4 Kryteria 26

3. Problemy 27

3.1 Problem dużych danych 27

3.1.1 Wstęp, opis problemu 27

3.1.2 Opis aplikacji 29

3.1.3 Wyniki 30

3.1.3.1 Generowanie rekordów 30

3.1.3.2 Metoda like_every_post_of_user 31

3.1.3.3 Metoda get_new_posts_from_subscriptions 31

3.1.4 Dyskusja, interpretacja wyników 31

3.2 Problem częstych zmian 33

3.2.1 Wstęp, opis problemu 33

3.2.2 Opis aplikacji 34

3.2.3 Wyniki 35

3.2.4 Dyskusja, interpretacja wyników 36
3.3 Problem sztywnego schematu danych 38
3.3.1 Wstęp, opis problemu 38
3.3.2 Opis aplikacji 39
3.3.3 Wyniki 41
3.3.3.1 Generowanie rekordów 41
3.3.3.2 Metoda get_jazz_albums 41
3.3.3.3 Metoda get_cheapest_game 41
3.3.3.4 Metoda finish_sale 41
3.3.4 Dyskusja, interpretacja wyników 42
3.4 Problem wielu relacji 44
3.4.1 Wstęp, opis problemu 44
3.4.2 Opis aplikacji 45
3.4.3 Wyniki 47
3.4.3.1 Generowanie rekordów 47
3.4.3.2 Metoda distant_friends 47
3.4.3.3 Metoda distant_friends2 48
3.4.3.4 Metoda distant_friends3 48
3.4.3.5 Metoda get_friends_with_interest 48
3.4.3.6 Metoda get_anniversary_date 48
3.4.4 Dyskusja, interpretacja wyników 48
4. Wnioski 50
Rekomendacje 54
Źródła 56

Streszczenie

Rozwiązania NoSQL powstały w odpowiedzi na wiele problemów, które napotyka się podczas tworzenia aplikacji, jak na przykład problem Big data. Bazy relacyjne zapewniają integralność danych i spójność transakcji, jednak robią to kosztem sztywnego schematu danych i skomplikowanego zarządzania. Zdecydowanie, spójność i integralność danych oraz transakcji jest wymagana w wielu przypadkach takich jak aplikacje finansowe, ale te cechy nie zawsze są potrzebne.

Celem tej pracy jest przedstawienie zalet i wad głównych modeli baz NoSQL. Wyjaśniam pewne pojęcia bazodanowe i na ich podstawie prezentuję zasady działania baz, ze zwróceniem uwagi na różnice między bazami NoSQL a SQL. Porównuję typy baz nierelacyjnych i pokazuję jakie zadania wykonują lepiej lub gorzej od baz relacyjnych. Na kilku przykładach staram się sprawdzić jak z punktu widzenia programisty może wyglądać użycie baz NoSQL w prawdziwej aplikacji webowej.

W związku z tym, że bazy NoSQL nie są rozwiązaniem dojrzałym i rozwijają się różnym tempem, architekci aplikacji webowych muszą uważnie wybierać między bazami NoSQL oraz relacyjnymi mając na uwadze ich specyficzne właściwości jak spójność, wydajność, skalowalność, dostępność danych, odporność na awarie oraz szereg innych kryteriów, które omawiam w tej pracy.

Wstęp

Efektywne przechowywanie i przetwarzanie informacji jest problemem, który dotyczy ludzkości od zawsze. Przed powstaniem komputerów sposobów gromadzenia danych było wiele i każdy z nich był niedoskonały: wolny, podatny na zniszczenia, fałszerstwa i kradzieże, w związku z czym cały czas szukano nowych rozwiązań. Wchodząc w drugą połowę XX wieku, kiedy do popularnego użycia wkraczały urządzenia elektroniczne, ludzkość otrzymała nowe narzędzie w postaci pamięci cyfrowej, którego wykorzystanie w przechowywaniu danych zdaje się rozwiązywać wiele z wymienionych wcześniej problemów. Z czasem komputerowe zarządanie danymi zdobyło zdecydowaną większość rynku. Efektywne przechowywanie danych to jeden z powodów, dla którego komputer znajduje się prawie w każdym domu lub w kieszeni każdego człowieka. Informatyzacja z pewnością położyła kres wielu wyzwaniom przechowywania danych, jednak stworzyła również nowe, które będą tematem dalszej części pracy. Oprogramowanie komputerowe, którego przeznaczeniem jest kolekcjonowanie i przetwarzanie informacji nazywamy bazą danych.

Historia baz danych zaczyna się w latach sześćdziesiątych dwudziestego wieku, kiedy komputery zaczęły wypierać używane w tamtym czasie karty perforowane. Powstały wtedy dwa kluczowe modele danych - hierarchiczny, rozwijany przez IBM oraz sieciowy, opracowany przez CODASYL. Najpopularniejszy obecnie model relacyjny został zaproponowany w 1970 roku przez Edgara F. Codda, którego pierwszą realizacją był system R utworzony przez IBM sześć lat później. W latach 90tych utworzono pierwsze bazy obiektowe, które pozwalały zapisywać informacje w postaci obiektowej, czyli takiej samej w jakiej dane są przechowywane w aplikacjach napisanych w obiektowych językach programowania. Następna generacja baz danych, na których uwaga programistów została skupiona na początku dwudziestego pierwszego wieku opiera się na modelu NoSQL, wprowadzając bazy typów takich jak grafowe, czy klucz-wartość.

Wraz z rozwojem informatyki pojawia się coraz więcej baz danych, które producenci reklamują jako nowocześniejsze, szybsze i bezpieczniejsze. Architekt serwera aplikacji może się czuć przytłoczony wyborem tej ‘najlepszej’ bazy danych. Każdy producent bazy stara się przekonac programistę, że wybór jego technologii będzie dawał najwięcej korzyści, jednak

praktyka pokazuje, że dobór odpowiedniego narzędzia jest złożonym problemem, który wymaga podjęcia wielu decyzji podczas projektowania aplikacji. Nieudolny dobór optymalnych narzędzi bazo danowych jest spotykanym problemem w informatyce, który generuje takie nieprawidłowości jak: wolne działanie aplikacji, awaryjność, większe zużycie zasobów, czy też większe koszty utrzymania serwera aplikacji. Obecnie mało jest poradników, które pomagałyby architektowi podjąć decyzję, która obecnie dostępna na rynku baza danych spełnia jego wymagania. W mojej pracy postaram się opisać ten problem w kontekście baz NoSQL. Rozważę wady i zalety wybranych przeze mnie baz danych i wysunę wnioski, które będą mogły stanowić podstawę dla programisty przy wyborze odpowiedniej bazy danych.

1. O bazach danych

1.1 Typy baz danych i ich popularność

Ze względu na sposób konstrukcji baz danych, można je podzielić na dwie grupy:

1. Bazy relacyjne
2. Bazy nie tylko relacyjne

Ten sam podział można przedstawić za pomocą innych pojęć: SQL oraz NoSQL, które są częściej spotykane. Ich znaczenie wyjaśnię w następnych rozdziałach.

Według rankingu popularności prowadzonego przez portal db-engines.com bazy relacyjne posiadają znaczną przewagę nad innymi typami baz danych, mimo rosnącego zainteresowania bazami NoSQL [Rysunek 1.1].

303 systems in ranking, April 2016

Rank Apr 2016	Rank Mar 2016	Rank Apr 2015	DBMS	Database Model	Score Apr 2016	Score Mar 2016	Score Apr 2015
1.	1.	1.	Oracle	Relational DBMS	1467.53	-4.48	+21.40
2.	2.	2.	MySQL	Relational DBMS	1370.11	+22.39	+85.53
3.	3.	3.	Microsoft SQL Server	Relational DBMS	1135.05	-1.45	-14.07
4.	4.	4.	MongoDB	Document store	312.44	+7.11	+33.85
5.	5.	5.	PostgreSQL	Relational DBMS	303.73	+4.10	+35.41
6.	6.	6.	DB2	Relational DBMS	184.08	-3.85	-13.56
7.	7.	7.	Microsoft Access	Relational DBMS	131.97	-3.06	-10.22
8.	8.	8.	Cassandra	Wide column store	129.67	-0.66	+24.78
9.	9.	↑ 10.	Redis	Key-value store	111.24	+5.02	+16.69
10.	10.	↓ 9.	SQLite	Relational DBMS	107.96	+2.19	+5.67
11.	11.	↑ 14.	Elasticsearch	Search engine	82.58	+2.41	+17.92
12.	12.	↓ 11.	SAP Adaptive Server	Relational DBMS	73.32	-3.33	-13.37
13.	13.	13.	Teradata	Relational DBMS	72.26	-1.81	+2.00
14.	14.	↓ 12.	Solr	Search engine	66.02	-3.35	-15.98
15.	15.	15.	HBase	Wide column store	51.49	-0.92	-9.65
16.	16.	↑ 17.	Hive	Relational DBMS	49.08	-1.43	+6.33
17.	17.	↓ 16.	FileMaker	Relational DBMS	46.10	-1.83	-5.72
18.	18.	18.	Splunk	Search engine	42.35	-1.38	+4.32

Rysunek 1.1 Ranking baz danych wg portalu db-engines.com [1]

Trzy bazy danych zajmujące pierwsze miejsce w rankingu to Oracle, MySQL oraz Microsoft SQL Server, każda z których jest typu relacyjnego. Po analizie punktów przydzielonych przez autorów portalu widać, że pierwsza trójka deklasuje pozostałe bazy danych i na dzień dzisiejszy nie ma podstaw by stwierdzić, że ich pozycja może być zagrożona. Metoda przydzielania punktów przez db-engines.com jest rzetelna i uwzględnia takie czynniki jak: zainteresowanie bazą wg Google Trends, liczba ofert pracy w danej technologii, liczba wspomnień bazy danych na stronach internetowych i wiele innych [2].

Ważne jest by zaznaczyć, że ranking ten mierzy popularność baz danych, a nie to, czy dana baza jest 'lepsza' od drugiej. Każda baza posiada indywidualne cechy, które czynią ją stosownym wyborem w konkretnej sytuacji.

1.2 Pojęcia bazodanowe

Zanim przejdę do opisywania poszczególnych rodzajów baz, ważne jest, by wyjaśnić pewne właściwości baz danych, do których będę się odnosił w dalszych rozdziałach pracy, w kontekście baz SQL oraz NoSQL.

ACID (ang. *Consistency Atomicity Isolation Durability*) to zbiór właściwości, który gwarantuje poprawność transakcji w bazach danych. Transakcją oznaczamy zbiór operacji na bazie danych, które stanowią pewną całość i które powinny być wykonane w całości lub nie powinna być wykonana żadna z nich. Przykładem transakcji może być wykonanie przelewu bankowego, który składa się z wielu operacji takich jak obciążenie jednego rachunku i uznanie drugiego. Taki system bankowy wymagałby bazy danych obsługującej transakcje. Właściwości ACID to:

1. Atomowość (ang. *Atomicity*) - wymaga by dana transakcja wykonała się w całości lub w ogóle. Jeśli część transakcji nie powiedzie się, to proces powinien zostać przerwany, a baza danych wrócić do stanu sprzed wykonania transakcji.
2. Spójność (ang. *Consistency*) - jest to bardzo ogólna właściwość, która zapewnia, że każda transakcja przeprowadzi bazę danych z jednego prawidłowego stanu do drugiego. Oznacza to, że dane przed i po transakcji muszą być prawidłowe z perspektywy wszystkich zdefiniowanych zasad integralności.

3. Izolacja (ang. *Isolation*) - oznacza, że w przypadku dwóch transakcji zachodzących równelegle wynik jest taki sam jak w przypadku, gdy wykonywałyby się one sekwencyjnie. Można to osiągnać przez na przykład blokadę zasobów. Sposoby uzyskania izolacji są różne i zależą od implementacji.
4. Trwałość (ang. *Durability*) - zapewnia, że jeśli transakcja została ukończona, to zmiany przez nią wprowadzone będą trwałe i będą obowiązywały nawet w przypadku awarii takiej jak utrata zasilania.

Transakcje są dostępne w większości relacyjnych baz danych. W przypadku baz NoSQL właściwości ACID nie są standardem i są implementowane tylko przez część z nich. Dla wielu przypadków użycia, transakcje ACID dbają o bezpieczeństwo danych bardziej niż aplikacja tego wymaga [3]. W świecie NoSQL często kosztem spójności poprawia się takie cechy jak skalowalność i elastyczność.

Twierdzenie CAP *(ang. Consistency Availability Partition tolerance)* - nazywane także Twierdzeniem Brewera określa, że niemożliwe jest dla rozproszonego systemu komputerowego na spełnienie wszystkich trzech następujących właściwości:

1. Spójność (ang. *Consistency*) - wszystkie węzły mają dostęp do takich samych danych w jednym momencie
2. Dostępność (ang. *Availability*) - oznacza dużą dostępność danych w danym momencie
3. Odporność na partycjonowanie (ang. *Partition tolerance*) - system kontynuuje działanie mimo utraty części węzłów

Reguła CAP została sformułowana w roku 1998 przez E. Brewera, a w 2002 roku naukowcy z MIT przeprowadzili jej formalny dowód, dzięki czemu nadano jej miano twierdzenia [4].

Najłatwiej wyobrazić sobie CAP z użyciem dwóch węzłów znajdujących się po dwóch stronach partycji. Pozwolenie jednemu węzłowi na zaktualizowanie danych spowoduje, że w kontekście całego zbioru węzłów dane nie będą spójne, tracąc właściwość numer 1 twierdzenia. Jeśli celem jest zachowanie spójności, wówczas drugi węzeł musi być niedostępny, tracąc właściwość 2. Dopiero kiedy węzły się komunikują możliwe jest utrzymanie tych dwóch cech, przez co system nie będzie odporny na partycjonowanie, tracąc właściwość 3 [5]. Oczywiście w praktyce sytuacja nie jest tak jednoznaczna - wybór

pomiędzy cechami CAP może zajść wielokrotnie podczas działania aplikacji, a jej podsystemy mogą podjąć różne decyzje.

CAP wchodzi w użycie w miarę skalowania aplikacji. Przy niskich wielkościach danych zapewnienie spójności nie ma dużego wpływu na ogólną wydajność aplikacji. Jednak zwiększający się rozmiar danych może komplikować działanie systemu i powodować duże opóźnienia, co może prowadzić do błędów aplikacji. W systemach finansowych, które wymagają bezawaryjności jest to dużym problemem.

Bazy danych spełniają spójność, dostępność i odporność na partycjonowanie w różnych stopniach, zależnie od ich celów biznesowych. Na przykład zwiększanie liczby węzłów daje większą dostępność, lecz ogranicza spójność i odporność na partycjonowanie. Kiedy planuje się budowę rozproszonej bazy danych, należy zastanowić się, które z tych trzech właściwości są dla aplikacji najważniejsze. Dla przykładu, bazy BigTable oraz HBase twierdzą, że są bardzo spójne i zapewniają dużą dostępność danych [6][7], a bazy typu Cassandra poświęcają spójność na rzecz dostępności i odporności na partycjonowanie [8]. Ogólnie uznaje się, że bazy NoSQL stawiają większą wagę na dostępność, a później na spójność. Bazy relacyjne spełniające ACID robią to na odwrót [9].

BASE (ang. *Basically Available, Soft state, Eventually consistent*) - zbiór właściwości powstały jako alternatywa dla ACID. ACID oraz BASE reprezentują dwie filozofie, które można umieścić na różnych końcach spektrum spójności - dostępności. Właściwości ACID przykładają dużą wagę do spójności danych i są tradycyjnym podejściem do baz danych. BASE jest wzorem dla systemów, którym zależy na dużej dostępności. Nowoczesne, duże systemy, jak na przykład chmury, używają kombinacji tych dwóch podejść. Rozwinięcie skrótu BASE to:

- Basically Available - dane są dostępne, ale niekoniecznie wszystkie w każdym momencie
- Soft State - stan węzłów może się zmienić w czasie, nawet bez wykonywania na nim operacji. Jest to związane z zachowaniem spójności.
- Eventually consistent - po pewnym czasie wszystkie węzły będą spójne, jeśli nie otrzymują w tym czasie dodatkowych rekordów

Interpretacja cech BASE daje dużo więcej swobody od ACID i nie służą też jako bezpośredni odpowiednik właściwości ACID [10]. BASE jest optymistyczny i akceptuje

fakt, że spójność bazy będzie ulegać ciągłym zmianom. Dzięki takiemu podejściu, to jest wymiany spójności na dostępność, można uzyskać dużą poprawę skalowalności, których nie byłoby się w stanie osiągnąć podążając nurtem ACID. Z tego względu bazy NoSQL przystosowane do Big data takie jak kolumnowo zorientowane, klucz-wartość oraz dokumentowe opierają się w dużej mierze na właściwościach BASE.

Nie ma prawdziwej odpowiedzi na pytanie, który z modeli, ACID czy BASE, jest lepszy dla danej aplikacji. Przed wyborem bazy danych programiści powinni wiedzieć, jakie konsekwencje może ze sobą nieść swobodniejsza definicja spójności modelu BASE w kontekście ich aplikacji. Stawienie czoła ograniczeniom BASE może być problematyczne w porównaniu do na przykład prostoty transakcji modelu ACID.

Schemat bazy danych - struktura wymuszająca pewne cechy na modelu bazy danych, w celu organizacji danych. W przypadku baz relacyjnych, schemat definiuje tabele, rodzaje pól, relacje, indeksy, wyzwalacze i wiele innych elementów.

Bazy NoSQL w dużej części są bezschematowe. Brak schematów pozwala na przechowywanie dowolnej informacji bez wcześniejszej wiedzy o kluczach i typach danych. Daje to elastyczność danych i łatwiejszą możliwość denormalizacji, co niesie ze sobą lepszą wydajność i skalowalność.

Brak schematów można porównać do sposobu przechowywania zmiennych w dynamicznych językach programowania. Ze względu na to podobieństwo, struktury danych bazy bezschematowej łatwo jest przetłumaczyć na obiekty takiego języka dynamicznego.

Wadą braku zdefiniowanej struktury przechowywanych danych jest konieczność przeniesienia odpowiedzialności za poprawną manipulację danymi na kod aplikacji, czyli między innymi za przeprowadzanie walidacji i obsługę operacji zapisu / odczytu. Z tego względu, takie rozwiązanie wymaga od programisty mniejszego doświadczenia w projektowaniu baz danych kosztem większych umiejętności programistycznych.

Migracja bazy danych - proces przeniesienia danych z jednej bazy do drugiej. O ile wykonanie migracji dla relacyjnych baz danych jest łatwe ze względu na podobną strukturę danych, to migracja danych do bazy NoSQL może być bardzo kłopotliwa. Schematy baz NoSQL są bardzo elastyczne, w związku z czym ich reprezentacje danych znacząco się

różnią. Programista jest zmuszony własnoręcznie napisać program, który pobierze dane z pierwszej bazy i zapisze je w odpowiedniej formie w drugiej.

Klaster - grupa połączonych ze sobą komputerów, które pracując i komunikując się, razem są postrzegane jako pojedynczy system. Komputery wchodzące w skład klastra nazywamy węzłami.

Kworum - w bazach danych oznacza liczbę replik, które muszą potwierdzić wykonanie pewnego zadania, najczęściej zapisu lub odczytu, by transakcja została poprawnie wykonana.

Pamięć podręczna (ang. *cache*) - to pamięć, w której przechowywane są często używane dane, by przyspieszyć wykonywanie przyszłych zapytań. Gdy pamięć podręczna jest pusta, zapytanie do serwera aplikacji powoduje, że dane są pobierane z bazy danych i zwracane do klienta, po czym na serwerze wypełnia się pamięć podręczną tymi danymi. Przy następnych zapytaniach (do czasu usunięcia zawartości pamięci podręcznej) odpowiedź ze strony serwera zwracana jest bezpośrednio z pamięci podręcznej, z pominięciem bazy danych.

1.3 SQL

SQL (ang. Structured Query Language) - jest językiem zaprojektowanym do komunikacji z serwerami relacyjnych baz danych, który został zbudowany w oparciu o rachunek relacyjny. SQL wspiera operacje tworzenia i usuwania tabel, dodawania oraz wyszukiwania danych oraz tworzenia i modyfikowania schematów. W 1986 roku SQL stał się oficjalnym standardem, wspieranym przez Amerykański Narodowy Instytut Normalizacji [11].

W bazach relacyjnych dane przechowywane są w postaci wierszy, a te z kolei są pogrupowane w tabele. Każda tabela zawiera zestaw kolumn, który definiuje postać wierszy należących do tej tabeli. By skutecznie manipulować danymi w takiej bazie SQL dostarcza możliwość wykonywania zapytań, które można podzielić na:

- DDL (Data Definition Language) - do zarządzania tabelami. Podstawowymi poleceniami są CREATE, DROP oraz ALTER.

  Przykładowe zapytanie:

```
CREATE TABLE example(
     id INTEGER,
     name VARCHAR(50),
     PRIMARY KEY (id)
);
```

- DML (Data Manipulation Language) - do manipulacji danych. Do tej grupy należą INSERT, DELETE i UPDATE

  Przykładowe zapytanie:

```
INSERT INTO example(id, name) VALUES (1, 'John');
```

- DCL (Data Control Language) - do nadawania uprawnień do obiektów bazodanowych. Używane do tego instrukcje składają się ze słów kluczowych GRANT oraz REVOKE

  Przykładowe zapytanie:

```
GRANT SELECT
ON example
TO user;
```

- DQL (Data Query Language) - do formułowania zapytań do bazy danych. Wszystkie polecenia z tej grupy używają instrukcji SELECT

  Przykładowe zapytanie:

```
SELECT id, name
FROM example
WHERE id = 1;
```

Każde zapytanie SQL kończy się średnikiem. Dobrą praktyką jest zapisywanie instrukcji SQLowych wielkimi literami.

Mimo, że SQL w większości przypadków reprezentuje dane w postaci wygodnej dla programisty, to forma przechowywania informacji, czyli ograniczenie ich do tabel i relacji czasem może być kłopotliwe. SQL świetnie sprawdza się w bankach, na poczcie, w księgowości, czyli tam, gdzie wymagana jest duża ilość rekordów i tabel oraz mała w stosunku do nich liczba relacji.

Często zaprojektowanie bazy SQL w ten sposób, by było to rozwiązanie optymalne jest trudne i wymaga skrupulatnego planowania. Istnieją również takie przypadki, kiedy idealne rozwiązanie jest niemożliwe, więc programista musi zdecydować się, która niedoskonałość będzie dla niego najmniej uciążliwa. Jako przykład takiej sytuacji podam aplikację typu katalog produktów, której administrator chce przechować wiele różnych typów towaru. Dla relacyjnych baz danych istnieje kilka rozwiązań tego problemu, z czego każdy niesie ze sobą inny profil wydajnościowy [12]:

1. Utworzenie osobnych tabel dla każdego typu produktu. Wady tego rozwiązania to:
    - brak możliwości wyłączenia wspólnych kolumn z tabel
    - konieczność dostosowania wszystkich zapytań do dokładnego rodzaju produktu
2. Jedna tabela dla wszystkich produktów z uzupełnianiem kolumn odpowiadającym danemu typowi produktu. Ten sposób rozwiązuje się problem zapytań, jednak robi to kosztem zużywanej pamięci.
3. Utworzenie osobnych tabel dla każdego typu produktu wraz z wyłączeniem wspólnych kolumn do jednej tabeli i utworzeniem do niej relacji. Jest to rozwiązanie bardziej elastyczne od pierwszego i zużywa mniej pamięci niż rozwiązanie drugie, lecz wymaga korzystania z kosztownej operacji JOIN by otrzymać wszystkie cechy produktu.

Kolejną wadą baz relacyjnch jest możliwość ataku typu SQL injection [13]. W związku z tym, że SQL jest językiem interpretowanym, istnieje możliwość nadużyć w przypadku konstruowania zapytań z wykorzystaniem parametrów pochodzących z zewnątrz aplikacji. Jeśli twórca aplikacji nie zadba o sprawdzenie poprawności danych wejściowych

stanowiących część zapytania, atakujący może być w stanie zmienić sposób działania zapytania lub dopisać do niego dodatkowe komendy.

Nie należy zapominać, że SQL to taka grupa baz danych, w której każda z nich posiada unikatowe dla niej cechy, które mogą tworzyć znaczące różnice w jej funkcjonowaniu. W związku z tym prawidłowe przypadki użycia baz mogą się różnić mimo ich przynależności do grupy baz relacyjnch. Pewne wady znajdujące się w jednej bazie SQL mogą być nieobecne w drugiej.

1.4 NoSQL

NoSQL - oznacza "Not only relational". Sformułowanie 'NoSQL' zostało po raz pierwszy użyte przez Carla Strozziego w 1998 roku, lecz dopiero ponowne wprowadzenie tego terminu w 2009 roku spowodowało, że zaczęto go używać do kategoryzowania nierelacyjnych baz danych [14].

Bazy NoSQL zaczęły zyskiwać dużą popularność dopiero około roku 2009. Mimo, że przez wielu uważane za nową technologię, to w przeszłości powstało wiele baz, które można skategoryzować jako NoSQL. Najstarszy z przykładów, MultiValue, powstał już w 1965 roku. Od tamtego czasu rozwój baz noSQL toczył się powolnym tempem. Rosnące zainteresowanie alternatywami dla SQL pod koniec lat 90tych zaowocowało utworzeniem grafowej bazy Neo4j w 2000 roku. Zachęciło to programistów do poświęcenia większej uwagi tym technologiom i zwiastowało powstaniom nowych baz, część z których jest popularna do dzisiaj takich jak: Memached powstały w 2003 roku, CouchDB z roku 2005, Cassandra z 2008 oraz Redis i MongoDB z 2009 [1].

Liczba ciągle powstających baz NoSQL i ich różne zastosowania powodują, że tego typu technologie dopiero starają się odnaleźć swoje miejsce w środowisku baz danych. W przeciwieństwie do baz relacyjnych, bazy NoSQL nie były przedmiotem wielu badań i eksperymentów, dlatego ważne jest by poświęcić temu zagadnieniu czas i postarać się dogłębnie je poznać.

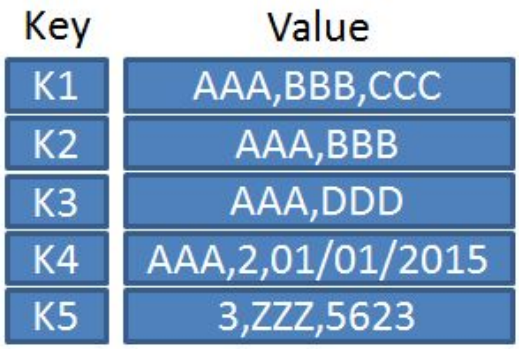
1.5 Typy baz NoSQL

Bazy NoSQL mogą zostać skategoryzowane z użyciem różnych podejść i różnorodnych kryteriów. Część ekspertów klasyfikuje je ze względu na model danych jaki

reprezentują, a większość z nich wymienia cztery główne typy [15]: bazy danych klucz-wartość (np. Redis, Riak, Voldemort), dokumentowe bazy danych (np. CouchDB, MongoDB, Simple DB), bazy zorientowane kolumnowo (np. Cassandra, HBase, BigTables), i grafowe bazy danych (np. Neo4j, OrientDB). W następnych częściach opiszę te cztery typy.

1.5.1 Bazy klucz-wartość

Ten model został zaimplementowany z użyciem tablicy z haszowaniem, w której przechowywane są unikatowe id i wskaźniki do konkretnych elementów zawierających dane, co tworzy pary klucz-wartość. Tablice z haszowaniem są dobrym narzędziem do wyszukiwania prostych lub skomplikowanych danych w ekstremalnie dużym zbiorze danych. Wartością dla klucza może być liczba, ciąg znaków, tablica, czy też tablica z haszowaniem. Ze względu na użycie takiego modelu danych bazy klucz-wartość całkowicie odrzucają idee schematów. Model danych jest listą par klucz-wartość:



Rysunek 1.2 Model klucz-wartość [16]

Bazy klucz-wartość potrafią sprawnie zarządzać bardzo dużymi ilościami danych, które mogą zostać rozproszone między węzły (skalowalność). Mogą przeprowadzać wielką liczbę zmian na sekundę z milionami użytkowników pracującymi w bazie jednocześnie dzięki rozproszonym obliczeniom i rozproszonemu przechowywaniu danych. Bazy klucz-wartość opierają się na redundancji, by chronić dane przed utratą w przypadku awarii węzła.

Tego typu bazy NoSQL dobrze sprawdzają się w przechowywaniu sesji, rezultatów analitycznych algorytmów i cache'owaniu, dziedziczą jednak jedną wadę wielu baz NoSQL, mianowicie mogą nie dostarczać możliwości przeprowadzania transakcji. Zapewnienie

atomowości transakcji może być niewydajne i może istnieć konieczność jej implementacji z poziomu aplikacji.

Kolejnym problemem jest brak możliwości dostępu do danych przez wartość. Niemożliwa jest próba wyszukania wszystkich rekordów zawierających podaną wartość. Jedynym sposobem by to zrobić jest podanie zakresu kluczy, z których będziemy mogli uzyskać odpowiednie rekordy.

1.5.2 Dokumentowe bazy danych

Dokumentowe bazy danych są podobne do typu klucz-wartość w tym, że przechowywane obiekty są powiązane (i przez to również odzyskiwane) poprzez klucze składające się z ciągów znaków. Różnica jest taka, że przechowywane wartości, które nazywane są "dokumentami", zapewniają pewną strukturę i kodowanie zarządzanych danych. Dokumenty mogą być przechowywane w standardowych formatach wymiany danych takich jak XML, JSON (Javascript Option Notation) lub BSON (Binary JSON). Dokumentowe bazy danych uważa się za potężne i elastyczne narzędzie do przechowywania Big Data. Ich model danych można przedstawić tak:

Rysunek 1.3 Dokumentowy model danych

Dokumentowy model danych zezwala aplikacjom na przechowywanie powiązanych ze sobą informacji w tym samym rekordzie. W ten sposób aplikacje mogą wykonywać mniej zapytań i aktualizacji, by ukończyć proste operacje [17].

Dokumentowe bazy danych różnią się od tych bazujących na modelu klucz-wartość w kilku aspektach. Po pierwsze, jak omówiłem wcześniej, w bazach typu klucz-wartość nie ma

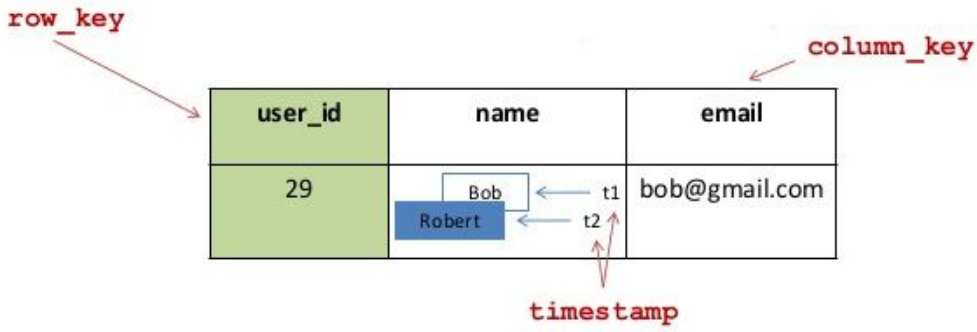
możliwości wyszukiwania względem wartości, za to bazy dokumentowe pozwalają użytkownikom na znajdywanie rekordów względem kluczy oraz zawartości dokumentów.

Po drugie w przeciwieństwie do baz klucz-wartość, dokumenty, oprócz prostych wartości, mogą zawierać usktrukturyzowane dane, takie jak tablice, zagnieżdżone dokumenty, czy nawet tablice dokumentów. Oznacza to, że dokumenty mają elastyczne schematy, co jest wymagane przez niektóre przypadki użycia. Oznacza to, że nic nie stoi na przeszkodzie, by w pewnej kolekcji dokumentów znajdowały się takie, których liczba i istota pól się różni. Skutkiem tego jest fakt, że zapytania mogą zwrócić dokumenty z różnymi atrybutami.

Dokumentowe bazy danych obsługują transakcje tylko częściowo. Dla przykładu MongoDB oraz CouchDB wspierają atomowość operacji jedynie dla pojedynczych dokumentów.

1.5.3 Bazy zorientowane kolumnowo

Bazy zorientowane kolumnowo w dużej części są spadkobiercami bazy utworzonej przez Google o nazwie Bigtable i są przystosowane do przechowywania petabajtów danych pomiędzy setkami lub tysiącami maszyn. Architektura baz zorientowanych kolumnowo łączy ze sobą cechy baz klucz-wartość oraz baz relacyjnch. Modele danych baz kolumnowych mogą różnić się w zależności od ich implementacji. Dla przykładu, baza BigTable przechowuje informacje w trójwymiarowych tablicach, które indeksowane są przez: klucz wiersza (który używa się podobnie do tych w modelach klucz-wartość oraz bazach dokumentowych), klucz kolumny wskazujący na konkretny atrybut, w którym wartość jest przechowywana oraz znacznik czasu (ang. *timestamp)*, który może odnosić się do momentu w czasie, w którym wartość pola została zapisana [Rysunek 1.4].



Rysunek 1.4 Model bazy BigTable

Bazy zorientowane kolumnowo zostały zaprojektowane tak, by rozpraszały dane do wielu węzłów. Charakteryzuje je duża przepustowość operacji read/write. Działają opierając się na założeniu, że pamięć jest najtańszym z 3 najważniejszych zasobów (CPU, pamięć, szybkość połączenia sieciowego) mających wpływ na wydajność działania rozproszonego systemu komputerowego. W związku z tym często korzystają z redundancji danych, co pozwala przyspieszyć wykonywane zapytania.

Ten typ baz wydaje się idealnym wyborem dla dużych aplikacji operujących na ogromnych zestawach danych. Ich największym atutem jest skalowalność. Przykładem tutaj może posłużyć baza Cassandra, w której, przy odpowiednim zaprojektowaniu modelu bazy danych, dodawanie kolejnych węzłów do sieci rozproszonej powoduje zwiększenie wydajności bazy w sposób liniowy [18].

Bazy zorientowane kolumnowo są przeznaczone do obsługi Big data, więc w kontekście twierdzenia CAP poświęcają spójność na rzecz dostępności i rozproszenia danych. W związku z tym, właściwości ACID mogą być kosztowne do implementacji. Kolumnowe bazy danych nie są więc dobrym wyborem w przypadku konieczności wykonywania transakcji.

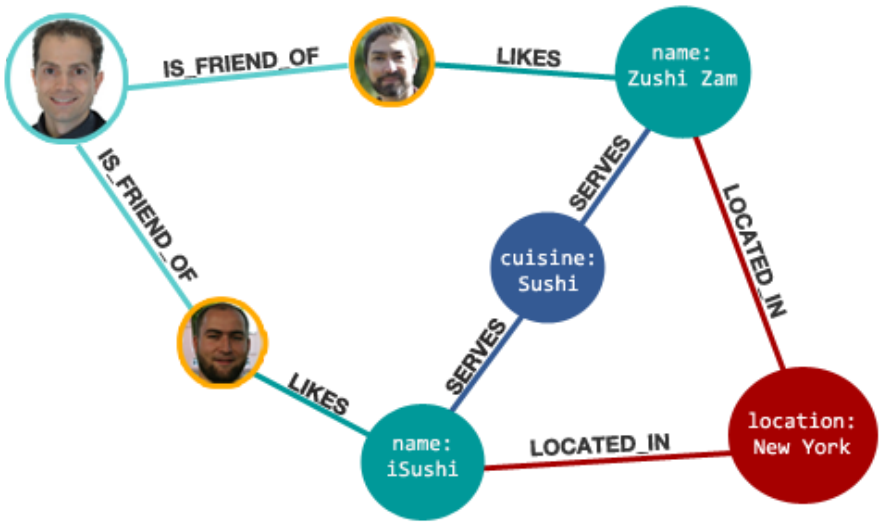
1.5.4 Grafowe bazy danych

Grafowe bazy danych są przystosowane do przechowywania informacji nie tylko o obiektach, lecz również o wszystkich związkach, które między nimi istnieją. Model takiej bazy składa się z wierzchołków oraz z krawędzi, które odpowiadają połączeniom między danymi.

Prosty, nieskierowany graf, w którym krawędzie pomiędzy wierzchołkami nie odwzorowują natury związku ani nie wskazują ich kierunków ma ograniczoną użyteczność. Dopiero dodając kontekst do wierzchołków oraz krawędzi poprzez dodanie do nich etykiet wzmacnia ich znaczenie w grafie i reprezentację całego modelu [Rysunek 1.5].

Ponadto, w modelach grafowych istnieją inne metody wzmacniania znaczenia węzłów i krawędzi jak na przykład:

- krawędziom mogą być nadane kierunki i wagi
- wierzchołkom i krawędziom mogą być przypisane dowolne, nieograniczone cechy
- wielokrotność krawędzi może odwzorowywać związki między parami wierzchołków



Rysunek 1.5 Grafowy model danych

Bazy typu grafowego nie używają schematów, co ułatwia reprezentację danych. Dodatkową elastyczność zapewnia fakt, że graf jest strukturą łatwą do wyobrażenia, co niesie ze sobą szereg zalet, na przykład pozwala w łatwy sposób przekształcić model biznesowy w model bazodanowy.

Relacyjne bazy danych nie są wydajne w przypadku istnienia wielu połączeń między danymi [19]. Brakuje im szybkości i elastyczności potrzebnej do przetworzenia i wyszukiwania wielokrotnych związków w środku dużego zestawu danych [20]. Udowodniono też, że niektóre bazy NoSQL np. typu klucz-wartość nie sprawdzają się w obsłudze silnie połączonych danych i dużych grafów [21]. W przeciwieństwie do nich, grafowe bazy danych bardzo dobrze radzą sobie z przechowywaniem, wyszukiwaniem i analizowaniem siły i natury związków pomiędzy wierzchołkami. Możliwość sprawnego wykonywania operacji na grafach (np. odpowiedzi na pytanie “Jak blisko powiązanych jest dwoje ludzi?”) daje programistom większą swobodę projektowania aplikacji.

Grafowe bazy, mimo natury reprezentacji danych, prezentują bardzo dobry profil wydajnościowy dla operacji read/write [22]. Szczególnie uwydatnia się to w przypadku skomplikowanej sieci wierzchołków i relacji.

Bazy NoSQL typu grafowego są jedynymi, które w pełni obsługują transakcje w ramach właściwości ACID [23].

2. Sposób przeprowadzania badań

W kontekście przedstawionych baz SQL oraz NoSQL widać, że ich liczba i ich różne rodzaje komplikują wybór odpowiedniej bazy dla konkretnego przypadku użycia. Z użyciem konkretnych przykładów użycia baz NoSQL postaram się sformułować poradnik, w którym poprzez porównanie cech różnych baz wskażę, na co architekt aplikacji webowej powinien zwrócić uwagę przy wyborze bazy danych.

Do zbadania wybrałem nastepujące bazy NoSQL

a. Baza oparta na modelu klucz-wartość: Redis (v. 3.0)
b. Baza oparta na modelu kolumnowym: Cassandra (v. 3.0.5)
c. Baza oparta na dokumentach: MongoDB (v. 3.2.6)
d. Baza oparta na modelu grafowym: Neo4j (v. 2.3.3)

W niektórych badaniach użyłem bazy MySQL jako reprezentanta baz relacyjnych. Bazy danych wybierałem na podstawie ich popularności oraz zaawansowania technicznego. Brałem również pod uwagę fakt, czy baza danych jest rozwijana i czy systematycznie pojawiają się jej nowe wersje.

2.1 Technologie

W tym rozdziale opiszę sposób, którym będę dokonywał analizy baz. Badania porównawcze będę przeprowadzał poprzez zbadanie 4 problemów, które oparte są na najczęściej spotykanych przeszkodach napotykanych przy używaniu baz SQL, których rozwiązaniem może być użycie odpowiedniej bazy NoSQL.

Bazę NoSQL do rozwiązania danego problemu wybiorę na podstawie podobieństwa jednego z jej możliwych przypadków użycia podanych przez producenta bazy do przedstawionego problemu oraz na podstawie możliwości technicznych bazy.

Dla każdego problemu zbuduję odrębną aplikację webową napisaną z użyciem frameworku Ruby on Rails, która zaproponuje rozwiązanie z użyciem jednej z baz NoSQL.

Ruby on Rails to framework open source do tworzenia aplikacji webowych. RoR został napisany w języku Ruby z użyciem architektury MVC (*Model View Controller*). RoR jest elastycznym narzędziem, które pozwala szybko budować aplikacje, między innymi dzięki używanym konwencjom:

Convention over Configuration zmniejsza ilość decyzji jaką programista musi podjąć zyskując prostotę i tym samym nie tracąc elastyczności. Zakłada wykorzystanie ustalonych zasad, które regulują skomplikowane kwestie działania aplikacji, co w praktyce redukuje to potrzeby konfiguracji oraz pisania kodu do minimum.

Don't Repeat Yourself (DRY) to zasada, której celem jest redukcja powtórzeń jakichkolwiek informacji w aplikacji, takich jak wklejanie tego samego kawałka kodu źródłowego w różnych miejscach. Przykładem wykorzystania zasady DRY może być odseparowanie danego kodu i odwoływanie się do niego kiedy jest to konieczne.

RoR jest popularnym narzędziem do budowania aplikacji webowych. Korzystają z niego między innymi: Github, Shopify i Basecamp. Do stycznia 2016 roku szacuje się, że na RoR opiera się 1,2 miliona stron internetowych.

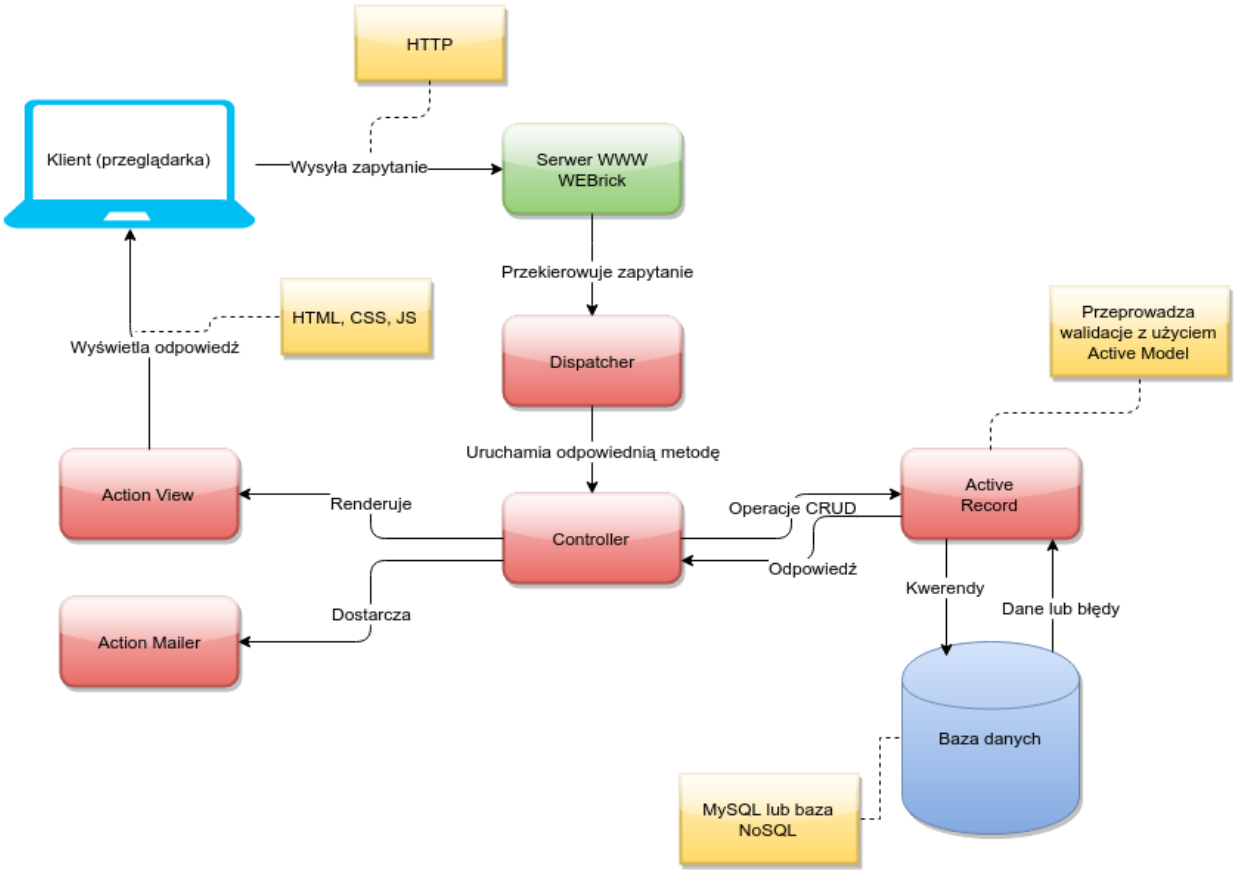
Architektura aplikacji jest przedstawiona na Rysunku 2.1. By zrozumieć zasadę działa aplikacji opartych na MVC konieczne jest wyjaśnienie zasad tej architektury:

Active Record - odpowiada za literę M - czyli Model - w architekturze MVC. Jest to warstwa systemu odpowiedzialna ze reprezentację obiektów i ich logiki, która bezpośrednio komunikuje się z bazą danych.

Controller - litera C w MVC. Służy jako pośrednik między Modelem i View. W controllerach znajduje się logika obsługująca zapytania.

Action View - odpowiada za literę V w MVC (View). Renderuje odpowiedź skierowaną do klienta. Generuje HTML, CSS, JS itp.

Na diagramie umieszczonych jest również kilka innych modułów, takich jak Action Mailer, który zezwala na wysyłanie maili przez aplikację oraz serwer WEBrick, który będzie obsługiwał zapytania HTTP aplikacji.



Rysunek 2.1 Architektura aplikacji zbudowanej na Ruby on Rails

W napisanych przeze mnie aplikacjach używam Ruby on Rails w wersji 4.2.5.2. Korzystam również z bibliotek:

- Faker (v. 1.6.3) - w celu generowania losowych danych.
- Mysql2 (v. 0.3.20) - sterownik MySQL do Ruby on Rails
- Cequel (v. 1.7.0) - sterownik Cassandry do RoR
- Mongoid (v. 5.1.1) - sterownik MongoDB do RoR
- Neo4j (v. 7.0.1) - sterownik Neo4j do RoR
- Redis-rails (v. 4.0.0) - sterownik Redis do RoR
- Actionpack-action_caching (v. 1.1.1) - udostępnia możliwość przeprowadzenia Action Caching'u

2.2 Pomiar wydajności

Mierzę wydajność aplikacji w następujący sposób:

1. W katalogu aplikacji uruchamiam konsolę Rails poleceniem ‘rails console’. Pozwala ona na interakcję z aplikacją z poziomu linii poleceń.
2. Do pomiaru wydajności zapytań do baz używam modułu Benchmark, który zawiera zestaw metod do mierzenia i zwracania czasu, który był potrzebny do wykonania danego zestawu poleceń. Ten moduł jest oficjalną częścią języka Ruby.
3. Pomiar powtarzam 10 razy i liczę z nich średnią.

   Przykład pomiaru może wyglądać następująco:

```
2.3.0 :001 > timing = []
 => []
2.3.0 :002 > 10.times do
2.3.0 :003 >     timing << Benchmark.measure { User.get_anniversary_date(user) }
2.3.0 :004?>   end
2.3.0 :005 > timing.reduce(:+)/timing.count
```

Rysunek 2.2 Przykład pomiaru z użyciem modułu Benchmark

Należy zaznaczyć, że pomiary wydajności prowadzone są na domyślnych konfiguracjach baz danych, czyli bez prób ich optymalizacji.

Pierwszym pomiarem w moich badaniach zawsze jest generowanie danych. Każda aplikacja posiada klasę o nazwie Janitor, której używam by w kontrolowany sposób generować rekordy do bazy. Tej klasy nie umieszczam w diagramach klas aplikacji. Generuję dane z użyciem tych samych narzędzi jak w przypadku mierzenia wydajności, to jest konsoli rails oraz modułu Benchmark:

```
2.3.0 :007 > timing = []
 => []
2.3.0 :008 > 10.times do
2.3.0 :009 >     Janitor.delete_all_records
2.3.0 :010?>     records = Janitor.generate_records
2.3.0 :011?>     timing << Benchmark.measure { Janitor.save_records records }
2.3.0 :012?>   end
```

Rysunek 2.3 Generowanie nowych rekordów z użyciem klasy Janitor

Kolejne testy są przeprowadzane na tych wygenerowanych danych.

2.3 Sprzęt

Wszystkie aplikacje przedstawione w tej pracy uruchamiane są na maszynie Lenovo Y580, której specyfikacje to:

a. Procesor Intel Core i5-3210M @ 2.50GHz × 4

b. Pamięć RAM 8 GB (DDR3, 1600 MHz)
c. Dysk SSD 128GB
d. Grafika NVIDIA GeForce GTX 660M + Intel HD Graphics 4000
e. Używany system operacyjny: Ubuntu 16.04

2.4 Kryteria

Kryteria które będę porównywał przy opisywaniu każdego problemu to:

a. Wydajność - zmierzę operacje zapisu danych oraz ich odczytu z uwzględnieniem skomplikowanych, wymagających obliczeniowo zapytań
b. Skalowalność - ocenię możliwość ekspansji bazy na więcej niż 1 maszynę
c. Zastosowanie - udzielę odpowiedzi na pytania: czy przypadki użycia bazy NoSQL zbiegają się z problemami, z którymi może się spotkać programista oraz jakie korzyści daje użycie tej badanej bazy nad innymi.
d. ACID - zbadam obsługę transakcji w bazie danych ze względu na właściwości atomowości, spójności, izolacji i trwałości.
e. Trudność implementacji - ocenię poziom trudności projektowania schematów, tworzenia kwerend i łatwość obsługi bazy.
f. Kompatybilność z różnymi językami programowania
g. Dostępność licencji - sprawdzę dostępność darmowych oraz płatnych wersji bazy
h. Pomoc techniczna
i. Opiszę dodatkowe właściwości specyficzne dla opisywanej bazy NoSQL

3. Problemy

3.1 Problem dużych danych

3.1.1 Wstęp, opis problemu

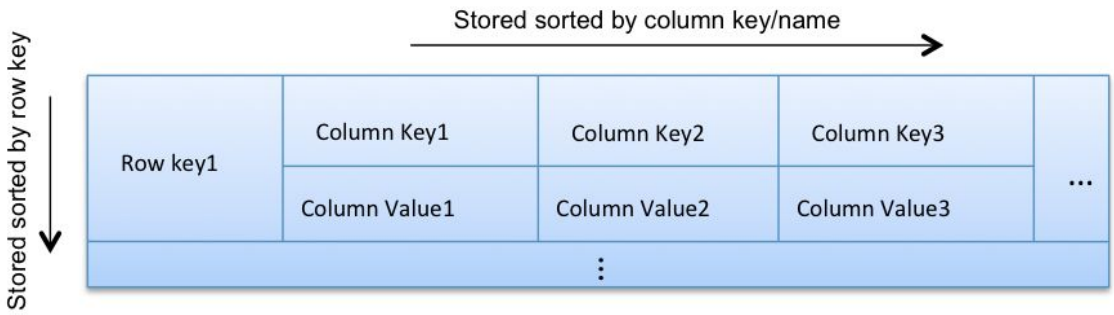
Obsługa bardzo dużych ilości danych (Big data) od zawsze stanowiła duże wyzwanie dla rozwijających się aplikacji. Bazy SQL nie są przystosowane do obsługi Big data z powodów takich jak: słaba skalowalność do dużych rozmiarów, brak wsparcia dla niestandardowych formatów danych oraz trudna implementacja niektórych, prostych z pozoru, zapytań jak na przykład znalezienie najkrótszej ścieżki między punktami. Facebook, Twitter i Google jako pierwsze zaczęły aplikować rozwiązania NoSQL w kontekście Big data na dużą skalę [24].

Aplikacja zbada problem Big data w kontekście baz SQL oraz NoSQL. Przy ocenianiu użytej bazy zwrócę szczególną uwagę na wydajność oraz skalowalność użytej bazy danych. Aplikacja zbada optymalizację zapisu i odczytu bardzo dużej liczby postów, głosów i komentarzy w bazie.

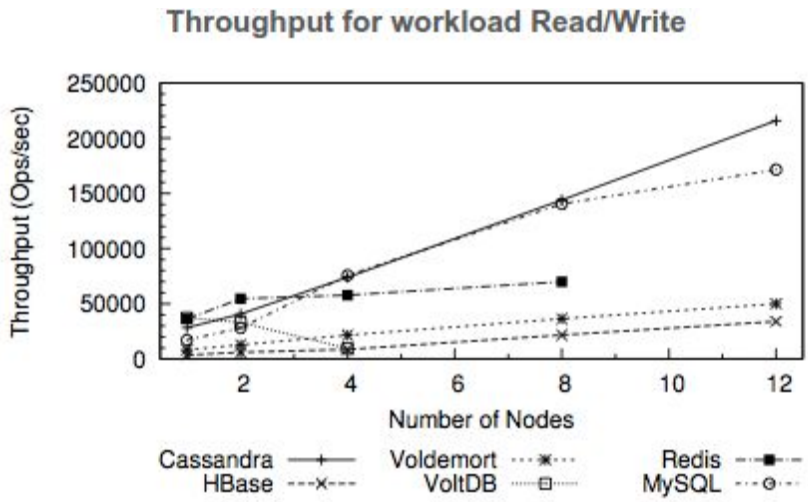
Dla tego przypadku wybrałem bazę Cassandra, która została zaprojektowana w celu zarządzania dużymi ilościami danych rozproszonych na wielu maszynach, zapewniając dużą dostępność wraz z odpornością na partycjonowanie. Mimo, że należy ona do rodziny baz zorientowanych kolumnowo, to model danych tej bazy przez jej twórców nazywany jest *partycjonowaną bazą wierszy*. Wiersze są zorganizowane w tabele. Pierwszym elementem klucza głównego tabeli jest *klucz partycji*. W partycji, wiersze są sortowane przez pozostałą część klucza głównego, czyli przez *klucz klastrujący* [25]. Do tego pozostałe kolumny mogą być indeksowane niezależnie od klucza głównego [26][Rysunek 3.1].

Cassandra nie jest w stanie wykonywać operacji JOIN ani zagnieżdżonych zapytań. Zamiast tego, wymaga wykorzystania denormalizacji danych [27].

W kontekście skalowalności Cassandra zawsze przewodzi badaniom, ponieważ jej wydajność rośnie liniowo wraz ze wzrostem liczby węzłów w sieci. Przykładem może być badanie 2012 roku przeprowadzone przez naukowców z uniwersytetu z w Toronto [Rysunek 3.2]. Na wykresie widać, że żadna inna baza nie osiągnęła tak dobrego wyniku.



Rysunek 3.1 Model danych Cassandry



Rysunek 3.2 Benchmark baz NoSQL [18]

Najważniejszymi cechami Cassandry są [28]:

- Decentralizacja - każdy węzeł w klastrze ma to samo zadanie, co oznacza, że awaria jednego z nich nie powoduje zatrzymania pracy całego systemu
- Skalowalność - dodawanie nowych węzłów odbywa się bez żadnych opóźnień w działaniu aplikacji. Służy do tego narzędzie *nodetool*.
- Replikacja danych - dane są automatycznie replikowane do wielu węzłów, by zapewnić dostępność danych w razie awarii.
- Dostosowanie spójności - Cassandra pozwala dostosować poziom spójności dla odczytów i zapisów danych. W przypadku zapisu, najszybszy i najmniej spójny poziom wymaga potwierdzenia poprawnego zapisu na jednym węźle, a najwolniejszy, ale za to najspójniejszy wymaga potwierdzenia zapisu od wszystkich docelowych węzłów. W przypadku odczytu zasada działania jest podobna, z tym, że najspójniejszy

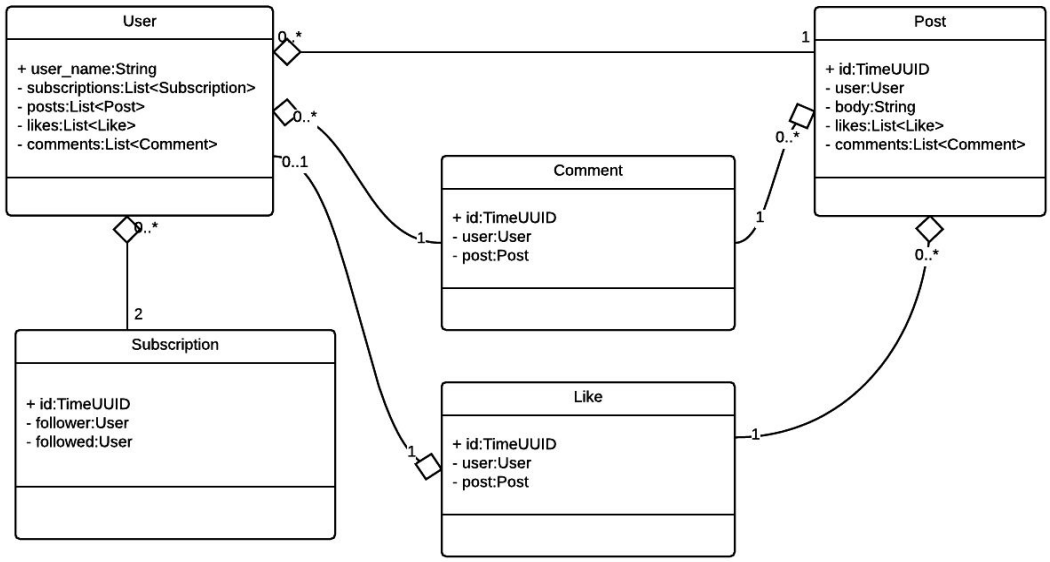
poziom wymaga zwrócenia danej od wszystkich węzłów, na której ta dana się znajduje [29].

- CQL - czyli Cassandra Query Language, to język zapytań Cassandry. Ma budowę podobną do SQL, w myśl znajomości tego języka przez programistów.
- Wsparcie dla języków programowania - dla Cassandry dostępnych jest wiele sterowników dla języków programowania takich jak: Python, C#, C++, Ruby, Java, Go i inne.

Wybrałem Cassandrę jako potencjalne rozwiązanie tego problemu, ponieważ jest ona z założenia przystosowana do Big data. Charakteryzuje się łatwą skalowalnością i wydajnością, która rośnie proporcjonalnie do liczby użytych maszyn. Ponadto uszkodzenie jednego węzła w klastrze nie powoduje zatrzymania pracy całego systemu. Spodziewam się również lepszej wydajności niż w przypadku użycia bazy SQL.

3.1.2 Opis aplikacji

Jest to aplikacja mikroblogowania na wzór Twittera, którą nazwałem *Squeaker*. Jej diagram klas przestawia się następująco:



Rysunek 3.3 Diagram klas aplikacji *Squeaker*

Klasa User reprezentuje użytkownika aplikacji. Użytkownicy mogą subskrybować siebie nawzajem, z czego każda subskrypcja tworzy nowy obiekt klasy Subscription. Ponadto mogą tworzyć posty i dodawać do nich komentarze oraz polubić je.

W klasie User znajdują się 2 metody, których używam do przeprowadzenia skomplikowanych zapytań do bazy danych by zbadać jej przepustowość. Te metody to:

- `like_every_post_of_user(user1, user2)` - dla każdego posta utworzonego przez user1, tworzy się nowe polubienie od user2.

```
 8
 9   def self.like_every_post_of_user(user1, user2)
10     Post[user1].each do |post|
11       Like.create(post: post, user_name: user2)
12     end
13     true
14   end
```

Rysunek 3.4 Metoda `like_every_post_of_user`

- `get_new_posts_from_subscriptions(user_name)` - pobiera posty utworzone dzień wcześniej przez użytkowników, których użytkownik o imieniu *user_name* subskrybuje.

```
15
16   def self.get_new_posts_from_subscriptions(user_name)
17     posts = []
18     Subscription[user_name].each do |subscription|
19       posts << Post[subscription.followed_name].from(1.day.ago)
20     end
21     posts
22   end
23 end
```

Rysunek 3.5 Metoda `get_new_posts_from_subscriptions`

By móc porównać wydajność Cassandry zdecydowałem się przeprowadzić te same eksperymenty dla bazy MySQL. Generowanie rekordów oraz testowane metody dla MySQL wyglądają niemal identycznie, więc nie będę umieszczał tutaj ich kodu źródłowego.

3.1.3 Wyniki

3.1.3.1 Generowanie nowych rekordów

Wygenerowałem:

- 500 użytkowników
- 10000 postów
- 3000 subskrypcji
- 5000 polubień
- 3000 komentarzy

	Cassandra	MySQL
Średni czas	37,8 s	41,5 s

3.1.3.2 Metoda like_every_post_of_user

	Cassandra	MySQL
Średni czas	49 ms	58 ms

3.1.3.3 Metoda get_new_posts_from_subscriptions

	Cassandra	MySQL
Średni czas	3 ms	18 ms

Należy zauważyć, że badania przeprowadzam dla jednego węzła. Według przedstawionych wcześniej badań, wyniki te powinny osiągać większą rozbieżność w miarę zwiększania liczby węzłów w sieci z korzyścią dla Cassandry. Zaznaczam też, że dla wszystkich operacji Cassandra ma ustawiony poziom spójności na domyślny, czyli QUORUM.

3.1.4 Dyskusja, interpretacja wyników

Dla wykonanych przeze mnie badań Cassandra jest minimum o 10% szybsza od bazy MySQL, co może być dużą różnicą dla aplikacji obsługujących miliony rekordów. Przyspieszenie działania aplikacji o 10% to również o 10% mniejsze wymagania procesora, co potencjalnie może pozwolić firmom zaoszczędzić na sprzęcie komputerowym. Należy zauważyć, że różnica szybkości powinna rosnąć w miarę dodawania liczby węzłów do klastra.

Pod względem skalowalności Cassandra jest zdecydowanym liderem wobec baz SQL oraz NoSQL. Po pierwsze, wydajność bazy rośnie liniowo wraz ze wzrostem liczby węzłów w klastrze [18]. Takiego wyniku nie zaobserwowano w żadnej innej bazie danych. Po drugie, dodawanie nowych węzłów za pomocą narzędzia *nodetool* odbywa się bez żadnych opóźnień w działaniu aplikacji. Po trzecie, węzły są zdecentralizowane, więc awaria jednego nie wypływa znacząco na pracę całego systemu. Dostępność danych w razie awarii również jest zachowana, ponieważ dane są replikowane na kilka węzłów.

Cassandra jest dobrym wyborem, kiedy architektowi aplikacji zależy na szybkości działania bazy danych, na dostępności danych mimo awarii systemu, łatwym skalowaniu bazy danych i przede wszystkim, zachowaniu wszystkich tych cech w kontekście Big data. Te cechy są jednak zyskiwane kosztem spójności, więc Cassandra nie jest przeznaczona dla systemów wymagających pełnej obsługi transakcji, jak na przykład systemów bankowych. Facebook używał Cassandry w postaci 200 węzłów do przeszukiwania odebranych wiadomości, jednak zrezygnował z tego rozwiązania, ponieważ twórcy Facebooka mieli problem z dostosowaniem spójności Cassandry [30]. Transakcje w Cassandrze są obsługiwane częściowo, w innym wymiarze niż zakładają to właściwości ACID. Cassandra zapewnia atomowość i izolację na poziomie wiersza, ale wymienia izolację i atomowość transakcji na dostępność danych i szybkie operacje zapisu. Daje też możliwość dostosowania poziomu spójności transakcji [31].

Ciężko jest ocenić poziom trudności implementacji aplikacji z użyciem Cassandry. Z jednej strony, programista ma do dyspozycji CQL, czyli język zapytań, który jest podobny w budowie do SQL, co może pomóc programistom zastępującym Cassandrą systemy relacyjne. Do tego Cassandra wspiera wiele języków programowania. Z drugiej strony jednak, sposób budowy bazy może przysporzyć problemów programiście. Po pierwsze, sam koncept bazy kolumnowej jest dość złożony. Dane są przechowywane w wierszach, które są podzielone na tabele ze względu na klucz partycji, a te są posortowane według klucza klastrowego. Po drugie, Cassandra wprowadza wiele nowych pojęć jak *Quorum* do dostosowywania spójności bazy. Może zdarzyć się również, że nieznajomość procesu dodawania nowych węzłów i narzędzia *nodetool* doprowadzi do zepsucia klastra [32].

Plusem jest obecność szczegółowej dokumentacji od twórców bazy, czyli firmy DataStax, która omawia każdy jej aspekt, włącznie z instrukcją instalacji bazy. Zawiera też poradnik do poprawnego modelowania danych [33]. Cassandra jest bazą open source, więc

jest darmowa, ale istnieje możliwość zakupu wsparcia technicznego od firm trzecich, jak Datastax [34]. Cassandra jest najpopularniejszą bazą kolumnową wg db-rankings.com [1]. Na portalu stackoverflow.com na chwilę obecną istnieje 9138 pytań oznaczonych etykietą ‘cassandra’. Z tych powodów programista nie powinien mieć problemu ze znalezieniem pomocy technicznej.

3.2 Problem częstych zmian

3.2.1 Wstęp, opis problemu

Kolejny problem, który zamierzam zbadać to szybki dostęp do małych, nietrwałych danych z często zmieniającymi się rekordami. Wymagany tutaj jest szybki zapis i odczyt niekoniecznie dużych danych na jednej maszynie, co znaczy, że nie będę zwracał uwagi na skalowalność. Bazy SQL są przystosowane do wykonywania skomplikowanych operacji, co może spowalniać działanie aplikacji w przypadku kiedy baza często aktualizuje małe ilości danych [35].

Dla tego problemu wybrałem bazę Redis. Redis to baza typu klucz-wartość, która domyślnie przechowuje dane w pamięci RAM. Firma G2 Crowd przedstawiła artykuł, w którym według opinii użytkowników, Redis jest najlepszą bazą NoSQL pod względem satysfakcji użytkowania. Baza ta wspiera takie struktury danych jak ciągi znaków, tablice z haszowaniem, listy, zbiory i posortowane zbiory. Jej największymi atutami są szybkość wykonywanych operacji odczytu oraz zapisu i prostota użycia.

Różnicą między bazą Redis a innymi typu klucz-wartość jest to, że jest bazą rozbudowaną i posiada wiele dodatkowych możliwości, jedną z których jest trwały zapis danych na dysku. Domyślnie Redis kopiuje dane na dysk co dwie sekundy. W ten sposób w razie awarii jedynie kilka sekund zmian zostałoby utraconych. Dodatkowo Redis wspiera transakcje, jednak działają one inaczej niż w bazach relacyjnych. Zapewniają one atomowość, ale posiadają pewne ograniczenia, na przykład nie obsługują cofania transakcji.

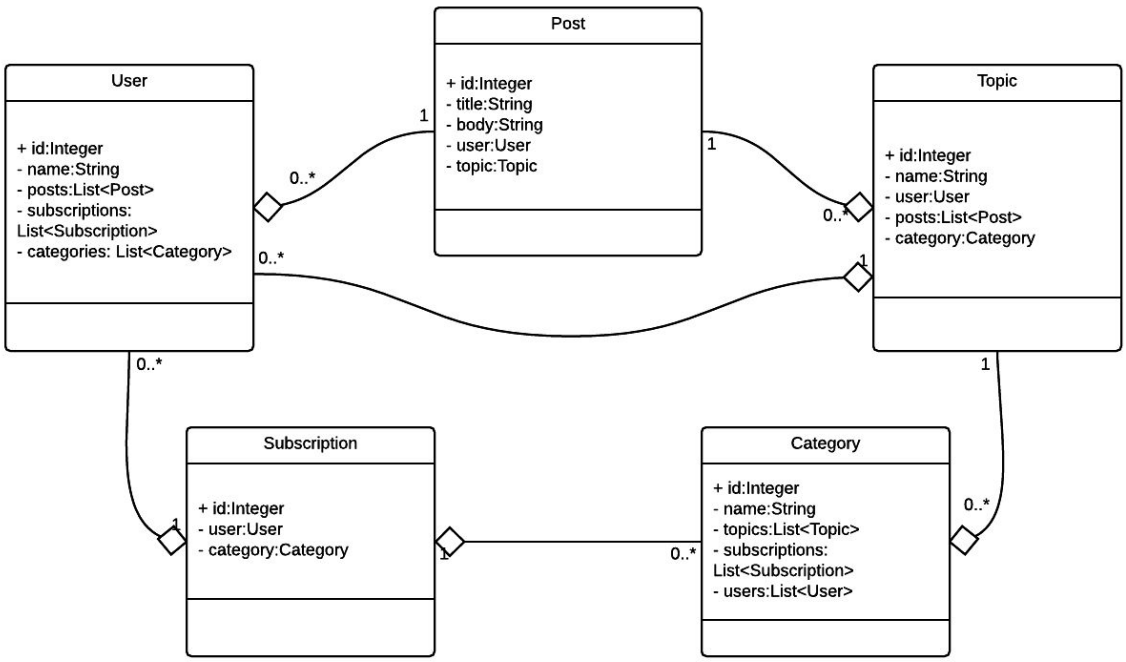
Redis jest również dość dobrze skalowalny. Zezwala na łatwe partycjonowanie danych, czyli dzielenie ich pomiędzy węzły. Redis obsługuje replikację danych stosując architekturę master-slave, co znaczy, że dane z węzła ‘master’ mogą się replikować do

połączonych z nim węzłów ‘slave’ oraz że każdy ‘slave’ może pełnić funkcję ‘master’ dla innych węzłów.

Uważam, że Redis sprawdzi się jako rozwiązanie tego problemu. Brak schematów i model przechowywania danych typu klucz-wartość idealnie przystosowują ją do przechowywania prostych danych. Baza Redis ma zdolność przechowywania danych w pamięci maszyny, tj. w pamięci RAM, dzięki czemu spodziewany jest dużo szybszy odczyt / zapis danych niewielkich rozmiarów w porównaniu do baz SQL.

3.2.2 Opis aplikacji

Utworzyłem stronę internetową, na której użytkownicy mogą tworzyć nowe tematy oraz wiadomości, gdzie każdy temat będzie należał do jednej z kategorii. Do aplikacji będę się odnosił nazwą RedisBoard.



Rysunek 3.6 Diagram klas aplikacji RedisBoard

RedisBoard oprócz modeli ma zdefiniowane widok i kontroler po to, by zwrócić wszystkie posty przez zapytanie HTTP. Aplikacja używa dwóch baz danych: MySQL jako baza główna oraz Redis do przechowywania pamięci podręcznej. Część treści zwracanej do klienta będę zapisywał do pamięci podręcznej, która posłuży jako sprawdzenia badanego

problemu. Rails wspiera kilka metod cache'owania. W tej aplikacji użyłem Action Caching, który zapisze do pamięci podręcznej widok wygenerowany przez zdefiniowaną przeze mnie metodę controllera.

Pamięć podręczną wykorzystam do wyświetlania postów użytkownika. Za pierwszym razem, kiedy użytkownik wejdzie na podstronę z postami, rekordy zwrócą się z bazy MySQL i zapiszą do pamięci podręcznej obsługiwanej przez Redis. Za drugim, zwróci się to co jest zapisane w pamięci podręcznej z pominięciem MySQL.

Szybkość działania tej aplikacji będę mierzył narzędziem UNIXowym curl, które służy do transportowania danych z lub do serwera. Po uruchomieniu serwera aplikacji wykonam więc dwukrotnie polecenie:

```
arclite@Harvey:~$ curl -so response -w '%{time_total}\n' localhost:3000/categories/1
```

Rysunek 3.7 Polecenie curl

Odpowiedź serwera będzie zawierała wszystkie tematy kategorii o id 1 oraz wszystkie posty wraz z autorami.

3.2.3 Wyniki

Najpierw wygenerowałem przykładowe dane:

- 50 kategorii
- 30 użytkowników
- 2000 postów
- 400 tematów
- 300 subskrypcji

Następnie wykonałem polecenia *curl*. W tabeli poniżej umieściłem wyniki dla Redis oraz dla innych sposobów przechowywania pamięci podręcznej. Każdy wynik to średnia z 10 pomiarów.

	Redis	MemoryStore	FileStore	MemCacheStore
Pierwsze zapytanie	175ms	188ms	277ms	178ms
Drugie zapytanie	16ms	15ms	18ms	17ms

3.2.4 Dyskusja, interpretacja wyników

O tym, że cache działa nie trzeba nikogo przekonywać. Jak widać, istnieją pewne różnice w wydajności poszczególnych metod przechowywania pamięci podręcznej. MemoryStore, czyli przechowywanie cache'u w pamięci RAM jest metodą najszybszą, ale nie nadającą się dla dużych danych. Nie może przechować większej ilości informacji niż zezwala na to ilość dostępnej pamięci RAM komputera. Najwolniejszą metodą jest FileStore, która polega na przechowywaniu pamięci podręcznej w plikach. Redis jest kombinacją tych dwóch podejść, ponieważ przechowuje dane w pamięci RAM i zapisuje je również na dysku, domyślnie w odstępie dwóch sekund. Zapewnia trwałość danych w przeciwieństwie do bazy Memcached, która jest podstawą metody MemCacheStore. Ta aplikacja pokazuje, że Redis, jako baza danych, jest przystosowana do tego by użyć ją jako cache w aplikacji webowej. Funkcjonalności bazy, jak na przykład możliwość zarządzania danymi bazy z linii poleceń, dają większą kontrolę nad pamięcią podręczną.

Redis jest bazą skalowalną z możliwością klastrowania węzłów. Zezwala na partycjonowanie danych i stosuje replikację danych z użyciem architektury master-slave. W ten sposób gwarantuje dużą dostępność danych i odporność na partycjonowanie. Ponadto twórcy Redis uważają, że wraz ze wzrostem węzłów w klastrze wydajność bazy rośnie liniowo do 1000 węzłów, jednak nie znalazłem badań potwierdzających tej tezy. Z tych względów nie ma przeciwskazań, by stosować bazę Redis dla Big data. Jedynym problemem jest słaba spójność, przez którą niektóre operacje zapisu, które zostały potwierdzone przez serwer, mogą być utracone. Powodem tego jest asynchroniczna replikacja danych [36].

Nie ma jednak wątpliwości, że jeśli architektowi aplikacji zależy na szybkości działania bazy danych, Redis jest wyborem do rozważenia. Dla jednego węzła wykazuje najlepszą wydajność spośród baz: MySQL, Cassandra, HBase, Voldemort i VoltDB [18]. Jedną z wad Redisa jest fakt, że jego serwer jest jedno-rdzeniowy, to znaczy, że nie korzysta z możliwości jaką dają procesory wielordzeniowe. Problem ten da się rozwiązać uruchamiając kilka instancji serwera, każdy dla jednego rdzenia, jednak komplikuje to implementację bazy.

Transakcyjność jest przez Redis wspierana, ale nie w tak zaawansowanym stopniu jak w bazach relacyjnych. Pod względem atomowości Redis pozostawia wiele do życzenia. Redis wykryje błędy syntaktyczne lub pewne krytyczne warunki, jak na przykład problemy z

dostępną pamięcią i nie wykona żadnych instrukcji z transakcji. Kiedy jednak transakcja zacznie się wykonywać, błędy takie jak wykonanie operacji niepasującej do danego klucza (na przykład operacji dla list LPOP na ciągu znaków) spowodują, że instrukcje poprawne się wykonają (bez możliwości ich automatycznego cofnięcia), a te niepoprawne nie [37]. Według twórców bazy, takie rozwiązanie ma swoje powody. Twórcy Redisa uważają, że instrukcje mogą się nie wykonać jedynie przez błędy programistyczne, które można wyeliminować w procesie tworzenia aplikacji. Do tego brak 'rollbacków' przyspiesza działanie bazy, które jest priorytetem [37]. Brak pełnej transakcyjności może więc okazać się problemem dla systemów, które wymagają całkowitej bezawaryjności wykonanych operacji, jak często przytaczany przeze mnie system bankowy. Jednak według mnie, taka transakcyjność jest wystarczająca dla większości możliwych przypadków użycia i nie powinna stanowić żadnego problemu.

Baza Redis jest bardzo prosta w użyciu. Według opinii użytkowników, jest ona najlepszą bazą NoSQL pod względem satysfakcji użytkowania. Posiada elastyczny, łatwy do zrozumienia schemat danych. Ilość obsługiwanych struktur danych, jak listy, zbiory i tablice z haszowaniem sprawiają, że łatwo jest przekształcić zmienne aplikacji na obiekty bazodanowe. Redis posiada bardzo dużą ilość sterowników dla różnych języków programowania, takich jak Haskell, Lua, C, C++, Java, Ruby i wiele innych. Na oficjalnej stronie bazy Redis znajduje się szczegółowa dokumentacja, która tłumaczy i uzasadnia każdy aspekt działania bazy, w tym partycjonowanie, transakcje i optymalizację pamięci. Zawiera szereg poradników dotyczących administracji, dział z najczęściej zadawanymi pytaniami oraz przykład aplikacji z użyciem bazy Redis.

Redis jest bazą open source, więc jest całkowicie darmowa w użyciu. W razie istnienia potrzeby, istnieje możliwość wykupu pomocy technicznej od twórców bazy. Redis oferuje też wsparcie społeczności poprzez listę mailingową. Dodatkowo na portalu StackOverflow na chwilę obecną znajdują się 8853 pytania oznaczone tagiem 'Redis' [38]. Z tych względów programista powinien z łatwością znaleźć rozwiązania ewentualnych problemów z bazą.

Szybkość i łatwość w użyciu bazy Redis czynią ją bardzo popularnym i przydatnym narzędziem. W mojej ocenie, jeśli architekt aplikacji znajdzie przypadek użycia odpowiedni dla bazy Redis, to wówczas powinien z tej bazy skorzystać, ze względu na wydajność i łatwość użycia.

3.3 Problem sztywnego schematu danych

3.3.1 Wstęp, opis problemu

W tej części opiszę problem zarządzania danymi katalogu produktów. Aplikacja musi być w stanie przechowywać wiele typów obiektów z podobnymi zbiorami atrybutów. Najlepsza wydajność zostanie tutaj osiągnięta, gdy baza będzie obsługiwać elastyczny schemat danych. Baza SQL może nie być tutaj dobrym rozwiązaniem, o czym pisałem w rozdziale o SQL.

Kolekcje tego typu wydają się być kompatybilne z dokumentową bazą MongoDB, dlatego w oparciu o nią zbudowałem następną aplikację. MongoDB przechowuje dane w postaci dokumentów BSON, który bazuje na popularnym JSONie z dodatkiem możliwości przechowywania kilku nowych obiektów takich jak zagnieżdżone dokumenty oraz tablice dokumentów. Pozwala to uprościć model danych i pozbyć się kosztownych operacji JOIN z modelu relacyjnego. Według rankingu db-engines.com MongoDB jest czwartą najpopularniejszą bazą danych, w tym najpopularniejszą bazą NoSQL [1].

MongoDB obsługuje klastry. Zapewnia wysoką wydajność i dużą dostępność, poprzez replikację danych z użyciem architektury master-slave, czyli w podobny sposób jak w bazie Redis. By efektywnie dystrybuować dane pomiędzy węzły stosuje skalowanie horyzontalne, czyli *sharding*.

Inną ważną cechą tej bazy jest możliwość indeksowania znana z modelu relacyjnego. Indeks to struktura danych, która poprawia czas pozyskiwania informacji z bazy danych kosztem dodatkowych operacji zapisu i zajmowanego miejsca, które potrzebne jest, by tę strukturę przechować. Indeksy są używane do szybkiego lokalizowania danych bez potrzeby przeszukiwania każdego wiersza w bazie danych.

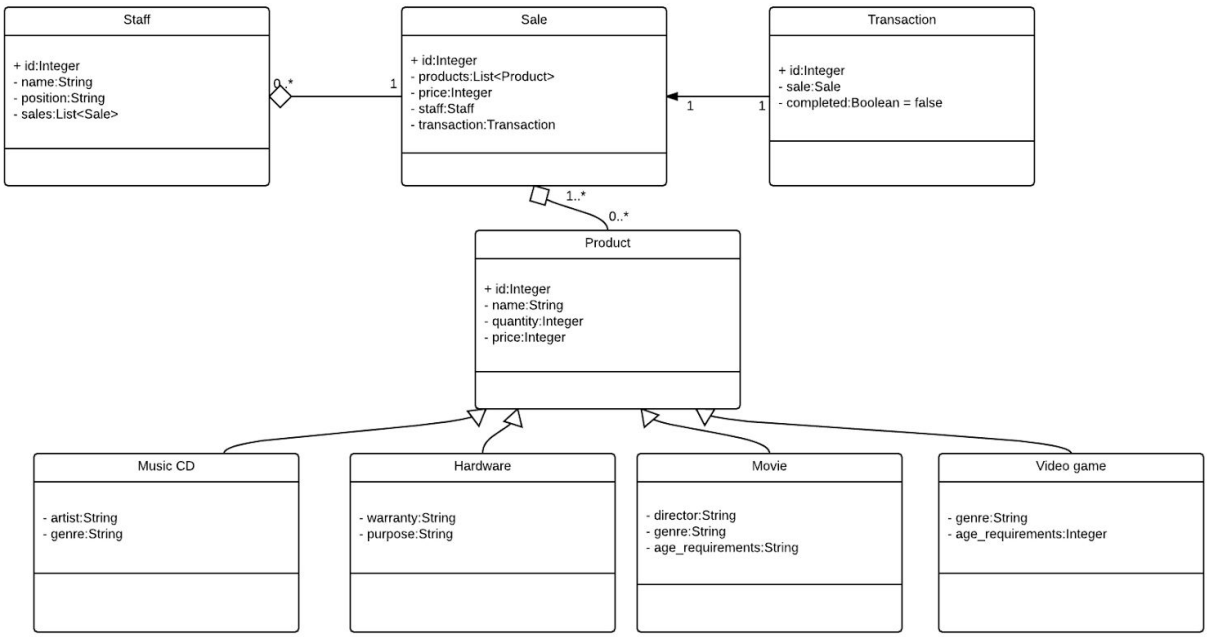
MongoDB jak każda baza danych posiada pewne wady. W niektórych przypadkach, kiedy aplikacja ma dostęp do dwóch procesów MongoDB, które nie komunikują się ze sobą, to możliwe będzie, że baza zwróci nieaktualne dane [39]. Ponadto, mogą wystąpić problemy związane z niewystarczającą ilością pamięci RAM dla systemów 32-bitowych [40].

Wybrałem tą bazę do rozwiązania tego problemu, ponieważ charakteryzuje się ona brakiem ściśle zdefiniowanej struktury obsługiwanych danych. Zamiast tego dane

składowane są jako dokumenty w stylu BSON, co zezwala na elastyczne zarządzanie typami danych wraz ze zmiennymi atrybutami.

3.3.2 Opis aplikacji

Aplikacja przechowuje dane na temat produktów znajdujących się w sklepie elektronicznym, takich jak filmy, muzyka, gry, sprzęt rtv itp. Nazwałem ją MongoShop.



Rysunek 3.8 Diagram klas aplikacji MongoShop

Aplikacja rozróżnia 4 różne typy produktów, a są to płyta CD, sprzęt, film lub gra komputerowa. Dziedziczą one z klasy Product, która zawiera wspólne pola dla produktów. Zakup, reprezentowany przez klasę Sale, może się składać z wielu produktów. Obiekt klasy Staff wyraża członka obsługi sklepu. Za każdy zakup trzeba oczywiście zapłacić. Informacje o płatności dla każdego zakupu reprezentowane są przez klasę Transaction.

Utworzyłem kilka metod, dzięki którym chciałem pokazać część funkcjonalności języka zapytań MongoDB. Pierwsza to `get_jazz_albums`, która zwraca wszystkie produkty typu album w gatunku Jazz:

```
13  def self.get_jazz_albums
14    Product.where(_type: "MusicCd", g: "Jazz").order_by(quantity: "desc")
15  end
```

Rysunek 3.9 Metoda `get_jazz_albums` (MongoDB)

```
26   def self.get_jazz_albums
27     Product.joins("INNER JOIN music_cds ON (product_info_type
28                    = 'MusicCd' AND music_cds.id = product_info_id)
29                    AND music_cds.genre = 'Jazz'").order(quantity: "DESC")
30   end
```

Rysunek 3.10 Metoda `get_jazz_albums` (MySQL)

Druga to `get_cheapest_game`, której zadaniem jest wybranie najtańszego produktu typu gra komputerowa o cenie maksymalnej 50:

```
17   def self.get_cheapest_game
18     VideoGame.where(:price.lte => 50).order_by(price: "asc").limit(1)
19   end
```

Rysunek 3.11 Metoda `get_cheapest_game` (MongoDB)

```
32   def self.get_cheapest_game
33     Product.where(product_info_type: "VideoGame").where('price >= 50').order(price: 'ASC').limit(1)
34   end
35 end
```

Rysunek 3.12 Metoda `get_cheapest_game` (MySQL)

Ostatnia to `finish_sale`, która oznacza transakcję powiązaną ze sprzedażą jako ukończoną oraz zmniejsza ilość sprzedanych produktów o 1.

```
13   def finish_sale
14     self.transaction.complete_transaction
15     self.products.each do |p|
16       p.quantity -= 1
17     end
18   end
19 end
```

Rysunek 3.13 Metoda `finish_sale` (MongoDB)

```
19   def finish_sale
20     self.transaction do
21       self.payment.complete_transaction
22       self.products.each do |p|
23         p.update(quantity: p.quantity - 1)
24       end
25     end
26   end
27 end
```

Rysunek 3.14 Metoda `finish_sale` (MySQL)

3.3.3 Wyniki

3.3.3.1 Wygenerowałem następujące dane:

- 50 pracowników
- 100 płyt CD
- 100 gier wideo
- 100 sztuk sprzętu elektronicznego
- 100 filmów
- 300 sprzedaży
- 300 płatności

	MongoDB	MySQL
Średni wynik	1,6 s	4,36 s

3.3.3.2 Metoda get_jazz_albums

	MongoDB	MySQL
Średni wynik	3,3 ms	3 ms

3.3.3.3 Metoda get_cheapest_game

	MongoDB	MySQL
Średni wynik	3,2 ms	2,3 ms

3.3.3.4 Metoda finish_sale

	MongoDB (bez transakcji)	MySQL (z transakcją)
Średni wynik	0,5 ms	7,5 ms

3.3.4 Dyskusja, interpretacja wyników

Na temat wydajności bazy MongoDB można znaleźć sprzeczne informacje. Jedno badanie wykonane przez firmę End Point opublikowane na stronie bazy Cassandra pokazuje, że dla operacji read / write w proporcji 1 do 1 Cassandra jest w stanie wykonać 13 razy więcej operacji niż MongoDB dla jednego węzła [41]. Dla większej liczby węzłów ta różnica rośnie. Według drugiego badania opublikowanego na stronie MongoDB dla tych samych operacji MongoDB jest o 50% szybsza od bazy Cassandra [42]. Minusem tego badania jest brak wyników dla większej liczby węzłów niż 1. Co ciekawe, obydwa badania korzystają z tego samego narzędzia YSCB utworzonego przez Yahoo!, które jest standardem do przeprowadzania pomiarów benchmarkowych.

Pomijając te dwa podane przeze mnie doświadczenia, istnieje wiele innych, mniejszych i gorzej udokumentowanych pomiarów wydajności bazy MongoDB, tym razem w porównaniu do MySQL [43][44]. Wyniki obu baz są w większości przypadków podobne, z małą przewagą dla MongoDB. Z tego względu uważam, że wydajność w przypadku MongoDB nie powinna być cechą decydującą przy wybieraniu odpowiedniej bazy danych dla aplikacji. Istnieje wiele sprzecznych ze sobą badań, w których wyniki zależą od producenta bazy, który te badania publikuje. W moich pomiarach również nie ma jednoznacznego zwycięzcy. Jeśli architektowi serwera aplikacji zależy na wydajności, powinien on wówczas przeprowadzić własne badania dla swojego systemu i interpretować wyniki w charakterze indywidualnym.

MongoDB posiada bogaty język zapytań, który pozwala przeprowadzać szereg kwerend, głównie za pomocą metod *find*, *limit* oraz *sort*. Ich wykorzystanie przedstawiłem w metodach `get_jazz_albums` oraz `get_cheapest_game`.

Metoda `finish_sale` klasy Sale modifykuje atrybuty z obiektu klasy Transaction oraz obiektów klas Product i jej klas potomnych. W związku z tym, że chcemy przeprowdzić operacje, które są ze sobą ściśle związane, warto byłoby tutaj użyć transakcji, by zapewnić atomowość tych operacji. MongoDB wspiera transakcje jedynie dla pojedynczych dokumentów, więc w moim przypadku nie jestem w stanie zapewnić atomowości wykonywanym operacjom.

Z badanych przeze mnie 4 baz poprawna konfiguracja MongoDB sprawiła mi najwięcej kłopotów. Instalacja biblioteki MongoDB do Ruby on Rails była prosta, lecz

musiałem ręcznie utworzyć bazę danych i użytkownika dla aplikacji, o czym nie napisano w dokumentacji. Zwróciłem też uwagę na to, że niektóre komendy linii poleceń MongoDB są nieintuicyjne, jak na przykład komenda *use*, której trzeba użyć by utworzyć nową bazę danych. Użytkownicy, w kwestii wad tej bazy, wskazują na problemy ze spójnością zapisów w przypadku rozproszenia bazy na kilka maszyn [39].

Modelowanie danych w MongoDB jest łatwe. Dzięki możliwości zagnieżdżania dokumentów, wszystkie klasy reprezentujące produkt przechowywane są w jednej kolekcji dokumentów o nazwie *products*. Dzięki temu uniknąłem łączenia tabel za pomocą operacji JOIN co miałoby miejsce w przypadku baz relacyjnych.

Pod względem skalowalności MongoDB prezentuje się dość dobrze. Węzły są grupowane w klastry, a odpowiednia dystrubucja danych w klastrze jest zapewniana przez skalowanie horyzontalne, czyli *Sharding*. MongoDB zapewnia duża dostępność danych poprzez replikację, gdzie węzły są podzielone na podstawowe i drugorzędne. Z tego powodu ta baza NoSQL może być dobrym wyborem do obsługi Big data.

Architekt aplikacji webowej powinien zwrócić uwagę na MongoDB ze względu na elastyczny schemat danych, który zezwala na przechowywanie ustrukturyzowanych zmiennych, jak tablice, tablice z haszowaniem, zagnieżdżone dokumenty oraz tablice dokumentów. Pozwala to uprościć model danych i pozbyć się kosztownych operacji JOIN.

MongoDB posiada wiele sterowników dla wszystkich bardziej popularnych języków programowania i dość rozbudowaną dokumentację, której w mojej ocenie brakuje intuicyjności. Szukanie odpowiedzi na jedno konkretne pytanie zajmowało mi więcej czasu niż w przypadku pozostałych baz.

MongoDB jest dostępna pod warunkami ‘GNU Affero General Public License’. Istnieje też możliwość wykupu licencji komercyjnej z dostępną pomocą techniczną. Producenci bazy oferują darmowe kursy online w postaci filmów edukacyjnych. Do tego ze względu na popularność bazy można łatwo uzyskać odpowiedzi na pytania od społeczności MongoDB na popularnych portalach informatycznych.

3.4 Problem wielu relacji

3.4.1 Wstęp, opis problemu

Ostatni problem to baza danych z dużą liczbą relacji między rekordami. Relacje w SQL są bardzo niewydajne, ponieważ by przeprowadzać kwerendy potrzebnych jest dużo kosztownych operacji JOIN.

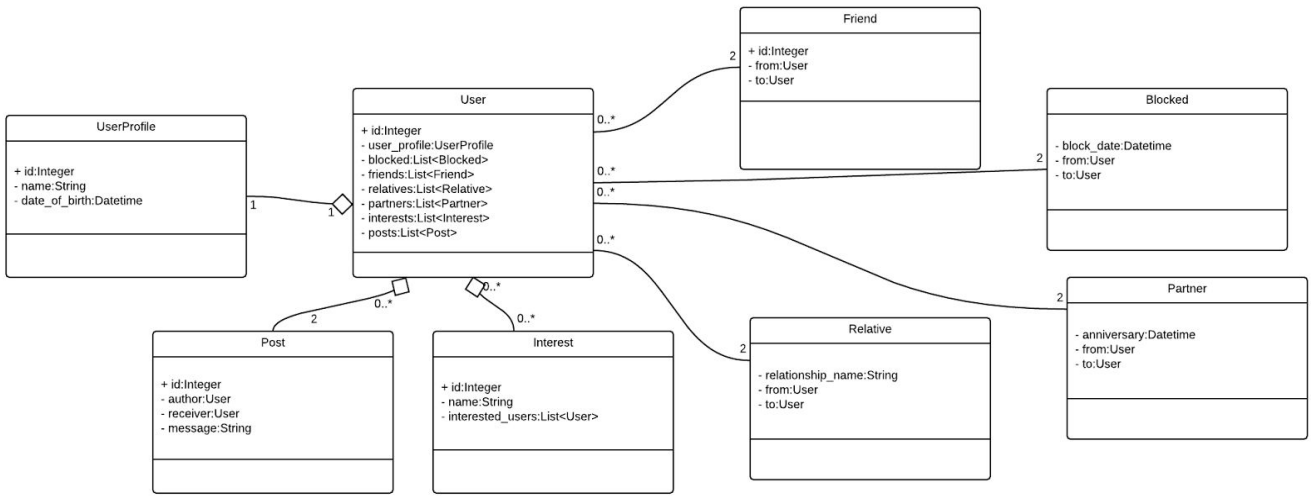
Zdecydowałem się na użycie bazy Neo4j. Jest to najpopularniejsza grafowa baza danych według db-engines.com [1]. Charakteryzują ją:

- Elastyczny schemat danych, który zezwala na tworzenie i usuwanie cech węzła w trakcie działania aplikacji.
- Obsługa właściwości ACID - każda aktualizacja grafu musi być wykonana w transakcji. Uzyskuje się to poprzez automatyczne blokowanie. Neo4j blokuje odczyt węzła, jeśli transakcja z niego czyta, oraz blokuje zapis węzła, jeśli transakcja go zmienia. Blokada zapisu nie może być uzyskana, gdy węzeł posiada blokadę odczytu oraz vice versa.
- Skalowalność - baza obsługuje podział na klastry tylko dla płatnej wersji Enterprise. Zapewnia narzędzie Neo4j Metrics do monitorowania klastra i wspiera replikację danych w architekturze master-slave. Dla darmowej wersji Community Neo4j może działać tylko na jednym węźle.
- Cypher - język zapytań bazy Neo4j. Ze względu na grafowy model danych nie przypomina on języka SQL. Twórcy bazy zapewniają, że Cypher wymaga pisania od 10 do 100 razy mniej kodu od SQL [45].
- Wsparcie dla języków programowania, między innymi dla: Javy, C#, Pythona, Javascriptu, Rubiego, PHP i innych.

Neo4j to grafowa baza danych, co oznacza, że idealnie pasuje do tego modelu danych. W związku z dużą liczbą relacji, w przypadku użycia bazy SQL musiałbym użyć dużej liczby operacji JOIN, które znacząco spowolniłyby pracę aplikacji. Według twórców bazy, Neo4j przeprowadza optymalne obliczenia w przypadku istnienia wielu relacji [46].

3.4.2 Opis aplikacji

Dla tego przypadku napisałem aplikację typu portal społecznościowy, do której będę odnosił się nazwą Neo4jNetwork. Przedstawiam jej diagram klas:



Rysunek 3.15 Diagram klas aplikacji Neo4jNetwork

W bazie istnieje wielu użytkowników z wieloma typami powiązań między nimi, takimi jak znajomość, pokrewieństwo, partner. Każdy użytkownik posiada informacje o swoim profilu w klasie UserProfile, może posiadać zainteresowania i pisać posty adresowane do innych użytkowników.

W klasie User znajduje się kilka metod, których używam do przeprowadzenia skomplikowanych zapytań do bazy danych by dodatkowo zbadać jej przepustowość. Te metody to:

1. `distant_friends, distant_friends2, distant_friends3`

Zwraca znajomych drugiego, trzeciego i czwartego stopnia dla podanego użytkownika. Zapytanie dla `distant_friends` wygląda tak:

```
17  def distant_friends
18    self.query_as(:user).match('user-[:knows*2]-(friend:User)').return('DISTINCT friend').to_a
19  end
```

Rysunek 3.16 Metoda `distant_friends` (Neo4j)

```
42  def distant_friends3
43    User.joins("INNER JOIN friendships f1 ON users.id = f1.friend_id").
44      joins("INNER JOIN friendships f2 ON f1.person_id = f2.friend_id").
45      joins("INNER JOIN friendships f3 ON f2.person_id = f3.friend_id").
46      joins("INNER JOIN friendships f4 ON f3.person_id = f4.friend_id").
47      where('f4.person_id = ?', self.id).distinct
48  end
49
```

Rysunek 3.17 Metoda `distant_friends3` (MySQL)

2. `get_friends_with_interest(name)`

Wyszukuje znajomych, którzy mają przynajmniej jednego krewnego interesującego się tematem podanym w parametrze metody. W kodzie wygląda to tak:

```
29  # return friends whos at least one relative have interest given as a parameter
30  def get_friends_with_interest(name)
31    self.friends(:f).relatives.interests.where(name: name).pluck(:f)
32  end
```

Rysunek 3.18 Metoda `get_friends_with_interest` (Neo4j)

```
50  def get_friends_with_interest(name)
51    User.joins("INNER JOIN friendships ON users.id = friendships.friend_id").
52      joins("INNER JOIN kinships ON kinships.person_id = friendships.friend_id").
53      joins("INNER JOIN user_interests ON user_interests.user_id = kinships.relative_id").
54      joins("INNER JOIN interests ON user_interests.interest_id = interests.id").
55      where("interests.name LIKE '%#{name}%'").
56      where("friendships.person_id = ?", self.id)
57  end
```

Rysunek 3.19 Metoda `get_friends_with_interest` (MySQL)

3. `get_anniversary_date`

Ta metoda powstała, ponieważ starałem się wymyślić jak najbardziej skomplikowane zapytanie, żeby sprawdzić jak ciężko będzie je sformułować oraz to, jak wydajny będzie Neo4j. Zapytanie zwraca daty rocznic związków znajomych użytkownika, których partner napisał wiadomość na profilu użytkownika zawierającą słowo ‘of’.

```
34  # return an anniversary date of my friends relationship whos partner posted
35  # on my profile a post containing the word "of"
36  def get_anniversary_date
37    self.friends.partners(:p, :rel).posts(:post).where(receiver: self)
38      .where("post.message CONTAINS 'of'").pluck(:rel).map(&:anniversary_date)
39  end
```

Rysunek 3.20 Metoda `get_anniversary_date` (Neo4j)

```
59  def get_anniversary_date
60    Partnership.select(:anniversary_date, :partner_id).
61      joins('INNER JOIN users ON person_id = users.id').
62      joins('INNER JOIN friendships ON users.id = friendships.friend_id').
63      where('friendships.person_id = ?', self.id).
64      joins('INNER JOIN posts ON users.id = posts.receiver_id').
65      where("posts.message LIKE '%of%'").distinct.map(&:anniversary_date)
66  end
67
```

Rysunek 3.21 Metoda `get_anniversary_date` (MySQL)

3.4.3 Wyniki

3.4.3.1 Generowanie rekordów:

- 100 użytkowników
- 100 zainteresowań
- 200 związków użytkownik-zainteresowanie
- 1000 postów
- 1500 związków typu znajomy
- 50 związków typu zablokowany
- 100 związków typu partner
- 100 związków typu krewny

Co w sumie daje 1200 obiektów i 1950 relacji.

	Neo4j	MySQL
Średni czas wygenerowania obiektów	26,09 s	3,21 s
Średni czas wygenerowania relacji	11,22 s	4,7 s
W sumie	37,31 s	7,91 s

3.4.3.2 Metoda distant_friends

	Neo4j	MySQL
Średni czas	26 ms	47 ms

3.4.3.3 Metoda distant_friends2

	Neo4j	MySQL
Średni czas	129 ms	2800 ms

3.4.3.4 Metoda distant_friends3

	Neo4j	MySQL
Średni czas	3146 ms	58594 ms

3.4.3.5 Metoda get_friends_with_interest

	Neo4j	MySQL
Średni czas	102 ms	37 ms

3.4.3.6 Metoda get_anniversary_date

	Neo4j	MySQL
Średni czas	108 ms	20 ms

3.4.4 Dyskusja, interpretacja wyników

Czas wygenerowania danych jest dosyć wysoki, co zgadza się z badaniami przeprowadzonymi między innymi przez projekt *graphdb-benchmarks* [22] oraz przez producentów bazy ArangoDB [47]. Grafowe bazy danych są mało wydajne w operacjach zapisu spośród baz NoSQL. Siłę szybkości Neo4j można dopiero zauważyć w kwerendach opierających się na grafie, jak znajdowanie najkrótszej ścieżki, znajdowanie sąsiadów, czy jak w moim przypadku znalezienie sąsiadów n-tego stopnia. Te zapytania są wykonywane szybciej niż w innych bazach NoSQL i SQL, a ich pisanie jest znacznie łatwiejsze niż w bazach SQL, co widać porównując ilości napisanego kodu. Używając Neo4j skomplikowane metody `get_friends_with_interest` oraz `get_anniversary_date` sformułowałem bez większych problemów. Zrobienie tego dla MySQL stanowiło duże

wyzwanie i wymagało dogłębnego przemyślenia problemu. Konieczne było wykonanie wielu kosztownych operacji JOIN.

Zdecydowanym plusem bazy Neo4j, który różni ją od pozostałych baz NoSQL jest zdolność przeprowadzania transakcji. Grafowe bazy danych są jedynym typem, który całkowicie obsługuje ten element baz danych, więc Neo4j jest dobrym wyborem, kiedy architekt aplikacji potrzebuje spójnej bazy z dużą dostępnością danych.

Neo4j to projekt open source. Jest on wydany w 2 edycjach: darmowej Community Edition, która może działać tylko na 1 węźle oraz płatnej Neo4j Enterprise, która nie ma ograniczeń w liczby węzłów. Płatna licencja jest używana między innymi przez firmy Adobe oraz Cisco. Neo4j posiada największą społeczność z grafowych baz danych. Na portalu StackOverflow istnieje ponad 10000 pytań oznaczonych tagiem neo4j [38]. Z tych powodów programista nie powinien mieć problemu z uzyskaniem pomocy technicznej.

Jeśli rozważany przypadek użycia pasuje do grafowego modelu danych, to Neo4j sprawdzi się jako substytut bazy relacyjnej. Jego największą siłą jest zdolność przeprowadzania skomplikowanych zapytań. Pisanie kwerend jest proste i daje więcej możliwości niż język SQL. Ponadto, pod względem wydajności Neo4j radzi sobie tak samo lub lepiej niż bazy relacyjne, co zostało zbadane i potwierdzone dla bazy MySQL [48], a spośród baz grafowych, Neo4j jest jedną z najszybszych [22].

4. Wnioski

Różnorodność baz danych jest ogromna. Każda z nich rozwiązuje inne problemy i pasuje do innych przypadków użycia. Zaczynając od podziału baz NoSQL na bazy grafowe, dokumentowe, klucz-wartość oraz zorientowane kolumnowo, które starają się odzworować najczęstsze modele biznesowe aplikacji, a kończąc na pojęciach transakcji, właściwościach BASE, czy wydajności, można zauważyć, że kryteriów ocen baz danych jest wiele.

Bazy relacyjne są powszechnie używane od lat 80tych, a bazy NoSQL zyskały popularność dopiero na początku tego tysiąclecia. O ile bazy SQL są od długiego czasu standardem w sposobie przechowywania danych i sprawdziły się jako rozwiązanie mało awaryjne i skuteczne, to bazy NoSQL wciąż są technologią nową (w popularnym użyciu od około 10 lat) i niezbadaną. Z tego względu mała liczba dużych firm używa rozwiązań NoSQL jako głównej bazy danych. Częściej korzysta się z nich równolegle z bazami SQL. Nie oznacza to, że programista powinien całkowicie wykluczyć bazy NoSQL z użycia w swoich aplikacjach. Bazy nierelacyjne mają wiele zastosowań w aplikacjach webowych i w niektórych sytuacjach radzą sobie dużo lepiej niż ich relacyjne odpowiedniki.

Bazy NoSQL zapewniają wiele korzyści w przechowywaniu, przetwarzaniu i tworzeniu zapytań w kontekście Big data. Są przystosowane do obsługi milionów klientów jednocześnie i obsługiwaniu wprowadzanych przez nich zmian. W przeciwieństwie do tradycyjnych, relacyjnych baz, NoSQL zapewniają lepszą skalowalność. Najlepszą pod tym względem okazuje się zorientowana kolumnowo Cassandra, która prezentuje najlepszy profil wydajnościowy pod względem liczby węzłów. Jej wydajność rośnie liniowo wraz ze wzrostem wielkości klastra, a większa wydajność oznacza mniejsze zużycie zasobów sprzętowych. Węzły są zdecentralizowane, obsługują replikację danych i można je dodać do istniejącego klastra bez opóźnień w działaniu aplikacji.

Relacyjne bazy danych są lepsze do ustrukturyzowanych danych, takich, które naturalnie da się przedstawić w tabelach. Często model biznesowy aplikacji nie pasuje do architektury tabele+relacje i wymagany jest elastyczniejszy schemat danych. W tym wypadku architekci najczęściej korzystają z bazy MongoDB, która przechowuje dane w dokumentach w formacie BSON. Programiści mogą w nim umieszczać struktury danych kompatybilne z formatem JSON plus kilka innych.

Jeśli aplikacja wymaga elastycznego modelu danych, ale chce zachować transakcje ACID z baz relacyjnych, wtedy odpowiednim wyborem może być grafowa baza Neo4j. Podobnie do baz SQL, jej zaletą jest możliwość formułowania i sprawnego wykonywania skomplikowanych zapytań.

Bazy SQL wspierają transakcje i dobrze radzą sobie z trudnymi wyzwaniami jakie stawia zachowanie spójności podczas aktualizacji danych, kiedy jedna transakcja wpływa na wiele tabel i te aktualizacje muszą się odbyć jednocześnie. Niestety, kosztowne pod względem wydajności operacje i cechy, które są potrzebne do wykonania skomplikowanych operacji, mogą spowalniać działanie aplikacji w przypadku, kiedy baza obsługuje małe, ale często zmieniające się dane [35]. Wówczas odpowiednim narzędziem do wykorzystania byłaby baza klucz-wartość Redis, która specjalizuje się w zarządzaniu małymi danymi, ze względu na bardzo niskie opóźnienia operacji zapisu i odczytu oraz ze względu na trwałość danych.

Do badań wybrałem popularne bazy NoSQL, każda z których posiada i rozbudowaną dokumentację, i sposoby na uzyskanie pomocy technicznej. Ponadto, bazy te są albo całkowicie darmowe, albo gwarantują nowe funkcjonalności w przypadku kupna wersji komercyjnej. Te kryteria nie muszą więc być brane pod uwagę przy wyborze odpowiedniej bazy NoSQL do aplikacji. Wyjątkiem tutaj może być Neo4j, w której ograniczenie darmowej wersji do możliwości konfiguracji tylko jednego węzła może zmusić aplikacje wymagające większej liczby węzłów do zakupu płatnej licencji.

Poniżej porównałem badane przeze mnie bazy NoSQL pod względem modelu danych, transakcyjności, skalowalności, wydajności, twierdzenia CAP oraz elastyczności danych. Powstała tabela może służyć jako uogólnienie najważniejszych cech baz danych, które może być przydatne podczas wyboru odpowiedniej bazy dla aplikacji.

	Cassandra	**Redis**	**MongoDB**	**Neo4j**	**MySQL**
Model danych	Kolumnowy, partycjonowana baza wierszy	Klucz-wartość	Dokumentowy	Grafowy	Relacyjny (tabele + relacje)
Transakcje	Niepełne. Spójność operacji na poziomie wiersza. Stawia nacisk na dostępność danych i	Transakcje bez możliwości ich cofania. Brak atomowości transakcji - część	Tylko dla pojedynczych dokumentów	Pełna obsługa transakcji spełniających właściwości ACID	Pełna obsługa transakcji spełniających właściwości ACID

	przepustowość operacji zapisu kosztem spójności transakcji	operacji w transakcji może się wykonać, część nie			
Skalowalność / obsługa big data	Obsługa klastrów. Dodawanie nowych węzłów bez żadnych opóźnień w działaniu aplikacji. Wbudowane narzędzie *nodetool* do obsługi węzłów	Obsługa klastrów. Jednordzeniowość bazy wymusza utworzenie nowej instancji dla każdego dodatkowego rdzenia, co komplikuje skalowalność	Obsługa klastrów. Dobry wybór dla big data. Możliwość dynamicznej zmiany atrybutów dokumentów	Obsługa klastrów. Nie ma przeciwskazań dla Big data	Obsługa klastrów, Problemy z wydajnością jeśli tabela ma więcej niż milion rekordów [24]
Wydajność	Bezkonkurencyjna przepustowość w kontekście Big data, która rośnie liniowo wraz ze wzrostem liczby węzłów w klastrze. Bardzo szybkie operacje zapisu	Bardzo szybkie operacje odczytu / zapisu. Najlepsza wydajność dla jednego węzła	Lepsza od SQL	Słaba dla operacji zapisu w porównaniu do innych baz NoSQL. Znakomicie radzi sobie z zapytaniami związanymi z grafem, takimi jak znajdowanie najkrótszych ścieżek	Duża przepustowość operacji odczytu / zapisu, jednak gorszy wynik od Cassandry. Małe opóźnienia operacji odczytu, większe opóźnienia dla zapisu [16]
Spójność (Twierdzenie CAP)	Niepełna. Możliwość dostosowania poziomu spójności w zależności od wydajności	Słaba spójność. Niektóre potwierdzone zapisy mogą zostać utracone	Silna spójność. Jeśli zapis się powiedzie, to odczyt zawsze zwróci poprawnie zapisane dane	Możliwa pełna spójność. Istnieje możliwość dostosowania poziomu spójności.	Pełna spójność, która jest priorytetem w przypadku partycjonowania [24]
Dostępność (Twierdzenie CAP)	Replikacja danych do kilku węzłów	Replikacja danych w architekturze master-slave	Replikacja danych. Podział węzłów na podstawowe i drugorzędne	Replikacja danych w architekturze master-slave	Replikacja danych w architekturze master-slave. Poświęcona na rzecz spójności w przypadku partycjonowania [24]
Odporność na partycjonowanie (Twierdzenie CAP)	Decentralizacja węzłów	Awaria klastra, jeśli większość węzłów ‘master’ przestanie działać	Jeśli podstawowy węzeł ulegnie awarii, operacje wykonywane są na drugorzędnym	Dane są bezpieczne dopóki działa choć jeden węzeł	Odporny na partycjonowanie z użyciem arbitracji [24]
Elastyczny model danych	Możliwość dodawania nowych kolumn do wierszy	Elastyczny model ze względu na brak schematu danych. Rekordy	Elastyczny model, możliwość zagnieżdżania dokumentów	Elastyczny model. Możliwość dodawania	Sztywny model danych. Podział danych na tabele i relacje. Brak

		zapisywane jako klucz-wartość		właściwości do istniejących węzłow / relacji	możliwości dynamicznej zmiany kolumn w tabelach
Dostępne licencje	Baza open-source, z możliwością zakupu bazy od firmy trzeciej z pomocą techniczną	Open source (licencja BSD)	Licencja AGPL v3.0, która ogranicza pewne prawa. Możliwość zakupu licencji komercyjnej	Darmowa tylko dla jednego węzła. Płatna wersja jest koniecznością dla aplikacji średnich wielkości	Open source
Dodatkowe uwagi		Redis nie obsługuje procesorów wielordzeniowych. Niewystarczająca ilość RAM znacznie spowolni pracę systemu	Możliwość indeksowania. Prawdopodobne problemy dla systemów 32-bitowych oraz gdy aplikacja ma dostęp do więcej niż jednego procesu bazy	Ciekawy i potężny nowy język zapytań *Cypher*	Sprawdzone, popularne rozwiązanie

Rekomendacje

Przed wyborem odpowiedniej bazy danych dla swojej aplikacji architekt serwera aplikacji webowej powinien rozumieć jakie wymagania może mieć aplikacja pod względem:

- potrzeby wykonywania transakcji
- wielkości danych
- liczby operacji zapisu / odczytu
- tolerancji niespójnych danych w replikach
- natury relacji pomiędzy rekordami w bazie i jak wpływa to na wykonywane zapytania
- dostępności danych i odporności na awarie
- potrzeby elastycznego modelu danych

Następnie w świetle tych wymagań, architekt aplikacji powinien odpowiedzieć na pytanie: czy faktycznie potrzebuję bazy NoSQL? dlaczego tradycyjna baza relacyjna nie spełni wymagań postawionych przez moją aplikację?

Jeśli aplikacja zmusza architekta do wybrania bazy NoSQL, to dopiero wtedy architekt może zacząć szukać odpowiedniej dla jego przypadku bazy. Wówczas powinien on odpowiedzieć na pytania:

1. Jak przechowałbym dane do bazy typu klucz-wartość, bazy dokumentowej, kolumnowo zorientowanej bazy danych i bazy grafowej? Czy łatwo będzie mi przekształcić model biznesowy do danego modelu bazodanowego?
2. Czy zdecyduję się na wybór jednego z tych podejść i dlaczego?

Jak starałem się pokazać w tej pracy, przypadki użycia baz NoSQL są dość szczególne. Reklama nowych baz NoSQL jako szybsze i ‘lepsze’ od SQL może być kusząca dla architekta aplikacji. Programiści często ulegają nieprzemyślanej pokusie spróbowania nowego rozwiązania w nadziei, że przyniesie ono poprawę wydajności aplikacji i ułatwi proces jej tworzenia. Często lepszym wyborem dla aplikacji jest użycie sprawdzonej bazy typu SQL. Programista powinien sięgać po rozwiązania NoSQL dopiero, gdy upewni się, że jego aplikacja pasuje do jednego z możliwych przypadków użycia bazy. W przeciwnym wypadku, może to doprowadzić do spowolnienia pracy systemu, większych kosztów

utrzymania systemu, problemy z jego konserwacją i w konsekwencji konieczności migracji danych do bazy SQL.

Źródła

[1] Solid IT, “Knowledge Base of Relational and NoSQL Database Management Systems”, http://db-engines.com/en/ranking, Kwiecień, 2016

[2] Solid IT, “Knowledge Base of Relational and NoSQL Database Management Systems”, http://db-engines.com/en/ranking_definition, 2016

[3] K. Kaur and R. Rani, "Modeling and querying data in NoSQL databases," *Big Data, 2013 IEEE International Conference on*, Silicon Valley, CA, 2013, pp. 1-7.

[4] S.Gilbert, N.Lynch, “Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services”, 2002

[5] Eric Brewer, “CAP Twelve Years Later: How the “Rules” Have Changed”, http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed, 30 maja, 2012

[6] Apache HBase, “Apache HBase”, https://hbase.apache.org/index.html, 2016

[7] Google, “Google Cloud Platform”, https://cloud.google.com/storage/docs/consistency, 2016

[8] DataStax, “About data consistency”, https://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlAboutDataConsistency.html, 2016

[9] Eric Brewer, “CAP Twelve Years Later: How the “Rules” Have Changed”, http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed, 30 maja, 2012

[10] Dan Pritchett, “BASE: An Acid Alternative”, http://queue.acm.org/detail.cfm?id=1394128, 28 lipca, 2008

[11] ISO/IEC 9075-1:2008: Information technology - Database languages - SQL - Part 1: Framework (SQL/Framework)

[12] MongoDB, “Product Catalog”, https://docs.mongodb.org/ecosystem/use-cases/product-catalog/, 2016

[13] Microsoft, “SQL Injection”, http://technet.microsoft.com/en-us/library/ms161953%28v=SQL.105%29.aspx, Kwiecień, 2014

[14] Lith, Adam; Mattson, Jakob, “Investigating storage solutions for large data: A comparison of well performing and scalable data storage solutions for real time

extraction and batch insertion of data" (PDF). Goteborg: Department of Computer Science and Engineering, Chalmers University of Technology. p. 70, 2010

[15] Ahmed Oussous, Fatima-Zahra Benjelloun, Ayoub Ait Lahcen, Samir Belfkih, "Classification and Comparison of NoSQL Databases for Big Data", 22 czerwca, 2015, p. 2 https://www.researchgate.net/publication/278963532_Comparison_and_Classification_of_NoSQL_Databases_for_Big_Data

[16] Clescop, "Key Value Concept Ilustration", https://upload.wikimedia.org/wikipedia/commons/5/5b/KeyValue.PNG, 2016

[17] MongoDB, "Data model design", https://docs.mongodb.org/manual/core/data-model-design/, 2016

[18] Tilmann Rabl, Mohammad Sadoghi, Hans-Arno Jacobsen, Sergio Gomez-Villamor, Victor Muntes-Mulero, Serge Mankovskii, "Solving Big Data Challenges for Enterprise Application Performance Management", http://vldb.org/pvldb/vol5/p1724_tilmannrabl_vldb2012.pdf, 2012

[19] Oracle, "Optimization for Developers", http://oracle.readthedocs.io/en/latest/sql/joins/index.html, 2015

[20] I. Robinson, J. Webber, and E. Eifrem, "Graph databases". O'Reilly Media, Inc, 2013.

[21] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. ACM, 2010, pp. 135–146.

[22] Socialsensor, "Performance benchmark between popular graph databases", https://github.com/socialsensor/graphdb-benchmarks, 2016

[23] Bryce Kerki Sasaki, "Graph Databases for Beginners: ACID vs. BASE explained", http://neo4j.com/blog/acid-vs-base-consistency-models-explained/, 4 września, 2015

[24] MongoDB, "Data Model Design", https://docs.mongodb.org/manual/core/data-model-design/, 2016

[25] Ellis, Jonathan, "Schema in Cassandra 1.1", 2012

[26] Ellis, Jonathan, "What's new in Cassandra 0.7: Secondary indexes", 2012

[27] Lebresne, Sylvain, "Coming in 1.2: Collections support in CQL3", 2012

[28] The Apache Software Foundation, "Cassandra", http://cassandra.apache.org/, 2015

[29] DataStax, "Configuring data consistency", https://docs.datastax.com/en/cassandra/2.0/

cassandra/dml/dml_config_consistency_c.html, 2016

[30] Muthukkaruppan, Kannanm “The Underlying Technology of Messages”, 2010

[31] DataStax, “About transactions and concurrency control”, http://docs.datastax.com/en/cassandra/2.0/cassandra/dml/dml_about_transactions_c.html, 2016

[32] Michael Kjellman, “Distributed Computing is Hard: A Cassandra Migration Horror Story”, http://www.planetcassandra.org/blog/distributed-computing-is-hard-a-cassandra-migration-horror-story/, 4 maj, 2014

[33] DataStax, “Cassandra Documentation”, http://docs.datastax.com/en/cassandra/3.x/cassandra/cassandraAbout.html, 2016

[34] The Apache Software Foundation, “Cassandra Wiki”, http://wiki.apache.org/cassandra/ThirdPartySupport, 2015

[35] April Reeve, “Big Data and NoSQL: The problem with relational databases”, https://infocus.emc.com/april_reeve/big-data-and-nosql-the-problem-with-relational-databases/, 7 września, 2012

[36] RedisLabs, “Redis Cluster Tutorial”, http://redis.io/topics/cluster-tutorial, 2016

[37] RedisLabs, “Transactions” http://redis.io/topics/transactions, 2016

[38] Stack Exchange Inc., “Tags”, http://stackoverflow.com/tags (po wpisaniu nazwy bazy do pola oznaczonego etykietą ‘Type to find tags’), 30 kwietnia, 2016

[39] Kyle Kingsbury, Jepsen: “MongoDB stale reads”, https://aphyr.com/posts/322-jepsen-mongodb-stale-reads, 20 kwietnia, 2016

[40] MongoDB, “32-bit limitations”, http://blog.mongodb.org/post/137788967/32-bit-limitations, 8 lipca, 2008

[41] DataStax, “Benchmarking Top NoSQL Databases”, http://www.datastax.com/wp-content/themes/datastax-2014-08/files/NoSQL_Benchmarks_EndPoint.pdf, 13 kwietnia, 2015. p. 9

[42] United Software Associates, “HIGH PERFORMANCE BENCHMARKING: MongoDB and NoSQL Systems”, http://info-mongodb-com.s3.amazonaws.com/High%2BPerformance%2BBenchmark%2BWhite%2BPaper_final.pdf, czerwiec 2015, p. 6

[43] Imran Omar Bukhsh, “Mysql vs MongoDB 1000 reads”, http://stackoverflow.com/questions/9702643/mysql-vs-mongodb-1000-reads, 14 marca, 2012

[44] Gerhard, “Is MongoDB Always Faster Than MySQL?”, https://blog.tinned-software.net/is-mongodb-always-faster-then-mysql/, 27 lutego, 2013

[45] Neo Technology, “eBay Now Tabkles eCommerce Delivery Service Routing with Neo4j”, http://neo4j.com/case-studies/ebay/, 2016

[46] Neo Technology, “Get Started” http://neo4j.com/developer/get-started/, 2016

[47] ArangoDB, “Performance comparison between ArangoDB, MongoDB, Neo4j and OrientDB”, https:/www.arangodb.com/2015/06/performance-comparison-between-arangodb-mongodb-neo4j-and-orientdb/ 11 czerwiec, 2015

[48] Marko Rodriguez, “MySQL vs Neo4j on a Large-Scale Graph Traversal”, https://dzone.com/articles/mysql-vs-neo4j-large-scale, 5 grudnia, 2011