POLITECHNIKA WARSZAWSKA
Wydział Elektroniki i Technik Informacyjnych
Instytut Informatyki

Rok akademicki 2013/2014

PRACA DYPLOMOWA MAGISTERSKA

Jakub Turek

Mechanizm modelowania danych i mapowania obiektowego dla Apache Cassandry

Praca wykonana pod kierunkiem
dra inż. Jakuba Koperwasa

Ocena: .....

.....

Podpis Przewodniczącego Komisji
Egzaminu Dyplomowego

Kierunek: Informatyka
Specjalność: Inżynieria Systemów Informatycznych
Data urodzenia: 1990.01.09
Data rozpoczęcia studiów: 2013.02.20

Życiorys

Urodziłem się 9 stycznia 1990 roku w Łodzi. W 1997 roku rozpocząłem edukację w Szkole Podstawowej nr 7 w Łodzi. W latach 2003-2006 kontynuowałem naukę w Gimnazjum nr 42 im. Władysława Stanisława Reymonta w Łodzi. Od 2006 roku uczyłem się w Liceum Ogólnokształcącym nr 31 im. Ludwika Zamenhofa w Łodzi. W 2009 roku zdałem egzaminy maturalne i ukończyłem szkołę licealną z wyróżnieniem. W latach 2009-2013 studiowałem dziennie informatykę na Wydziale Elektroniki i Technik Informacyjnych Politechniki Warszawskiej. Ukończyłem studia z wynikiem celującym i odebrałem tytuł zawodowy inżyniera. Obecnie kończę pracę dyplomową magisterską pod kierownictwem Instytutu Informatyki. We wrześniu 2012 roku rozpocząłem pracę zawodową jako programista aplikacji do zarządzania procesami biznesowymi oraz aplikacji mobilnych w firmie Xentivo, gdzie pracuję do dziś. Moją pasją jest tworzenie aplikacji mobilnych oraz internetowych, które uruchamiane są w środowisku iOS.

.....

Podpis studenta

Egzamin dyplomowy:
Złożył egzamin dyplomowy w dniu: .....
z wynikiem: .....
Ogólny wynik studiów: .....
Dodatkowe uwagi i wnioski Komisji: .....
.....

Streszczenie

Celem pracy dyplomowej jest stworzenie mechanizmu do modelowania i mapowania obiektowego dla aplikacji wykorzystujących bazę danych Apache Cassandra. System powinien cechować się wysoką wydajnością i wsparciem dla wzorców modelowania. Dodatkowym celem jest zachowanie zgodności z istniejącymi mechanizmami mapowania dla baz relacyjnych. Nie wszystkie cele udało się zrealizować. Stworzenie wydajnego mapowania wymagało odejścia od relacyjnego modelu danych. Praca dyplomowa zawiera zbiór informacji teoretycznych, badań oraz studia przypadków, które posłużyły do nakreślenia pryncypiów systemu modelowania obiektowego. Obejmuje również implementację omawianych mechanizmów.

Data modeling and object mapping mechanism for Apache Cassandra

Summary

The goal of this thesis is to create a modeling and object mapping mechanism for applications which use Apache Cassandra database. The mechanism should provide good performance and be capable of making use of Cassandra design patterns. An additional goal is to explore possibility of using existing interfaces for object-relational mapping. Not all goals could be achieved. Creating a fast system required abandoning object-relational mapping compatibility. The thesis contains of theoretical information, research results and case studies which contributed to specify principles of the mechanism. It also provides functional implementation of described ideas.

Spis treści

Rozdział 1

Wstęp

Tematem pracy jest stworzenie systemu mapowania obiektowego dla bazy danych Apache Cassandra. Celem pracy jest zweryfikowanie możliwości wykorzystania interfejsów analogicznych mechanizmów stworzonych dla relacyjnych baz danych. Podstawowym wymaganiem jest zachowanie wysokiej wydajności zapisu, która wyróżnia Cassandrę na tle innych systemów bazodanowych, a także możliwość czerpania z praktyk zwiększających efektywność modelowania opisanych przez użytkowników Cassandry. Niniejsza praca prezentuje zbiór rozważań teoretycznych, badań oraz doświadczeń z użytkowania, które przyczyniły się do powstania takiego mechanizmu. Częścią pracy jest również implementacja mechanizmu.

1.1 Zakres pracy

Zakres pracy obejmuje następujące elementy:

1. Badanie możliwości wykorzystania interfejsów mapowania obiektowo-relacyjnego dla bazy danych Cassandra:
    - Wyszukanie istniejących implementacji.
    - Pomiary wydajności ze szczególnym naciskiem na różnice w stosunku do relacyjnych baz danych oraz modelowania i komunikacji

z wykorzystaniem języka zapytań Apache Cassandry.

2. Zaproponowanie własnego interfejsu zorientowanego na wysoką wydajność oraz wsparcie w modelowaniu danych z wykorzystaniem najlepszych praktyk:

   - Zebranie i opisanie sposobów modelowania zależności między danymi oraz porównanie ich wydajności w różnych przypadkach użycia.
   - Zebranie i opisanie zbioru najlepszych wzorców modelowania oraz możliwości wspierania ich w mechanizmie.
   - Wykonanie referencyjnej implementacji zaproponowanego interfejsu.
   - Przeprowadzenie badań wydajnościowych implementacji.

1.2 Motywacja

W dobie szybkiego postępu technologicznego i intensywnego rozwoju informatyki, a zwłaszcza powszechnego dostępu do sieci Internet oraz niskich kosztów składowania danych, zaczął rozwijać się trend nazywany przez specjalistów od marketingu *big data*[1]. *Big data* to termin używany do określania dużych rozmiarów różnorodnych i często zmieniających się zbiorów danych. [1] Efektywne przetwarzanie takich danych wymaga stosowania innowacyjnych, stale ulepszanych rozwiązań technologicznych.

Powstanie ogromnych portali społecznościowych obsługiwanych przez dziesiątki tysięcy fizycznych maszyn spowodowało pojawienie się wyzwań, które wcześniej nie były brane pod uwagę. Awarie maszyn, dotychczas traktowane jako sytuacje wyjątkowe, przy tej skali użytkownia stały się regułą. W połączeniu z wymaganiami na krótki czas obsługi setek tysięcy jednoczesnych żądań doprowadziło to do osiągnięcia kresu możliwości wykorzystywanych od

[1] Z angielskiego oznacza to dosłownie „wielkie dane”.

wielu lat relacyjnych baz danych. Aby móc sprostać tym warunkom zaczęły powstawać nowe silniki bazodanowe, które odchodziły od postulatów transakcyjności i klasycznej, tabelarycznej reprezentacji danych. Jednym z najlepszych rozwiązań do obsługi wysoce rozproszonych środowisk jest Apache Cassandra.

Negatywną stroną wprowadzenia nowych rozwiązań z zakresu przechowywania danych było odcięcie się od wielu mechanizmów, które bardzo ułatwiały pracę z aplikacjami. Przede wszystkim nie było możliwe użycie istniejących systemów mapowania obiektowo-relacyjnego. Ich wykorzystanie stało się na tyle powszechne, że w dniu dzisiejszym mogą być uznawane za standard w tworzeniu aplikacji bazodanowych.

Wraz z rozwojem Cassandry, a zwłaszcza wprowadzeniem języka zapytań o składni przypominającej ten z relacyjnych baz danych, pojawiła się możliwość ponownego wykorzystania istniejących interfejsów. Nawet jeżeli zachowanie wysokiej wydajności wymaga wprowadzenia pewnych modyfikacji, to warto to uczynić. Dzięki temu możliwe jest zapewnienie narzędzia do szybszego rozwoju aplikacji oraz zmniejszenie bariery wejścia związanej z opanowaniem podstaw nowego systemu.

Autor ma nadzieję, że przeprowadzone przez niego badania przyczynią się do rozwoju prac nad efektywnym modelowaniem danych w Apache Cassandra oraz pomogą wypełnić lukę w oprogramowaniu wspomagającemu pracę z nierelacyjnymi bazami danych. Ponadto autor wyraża opinię, że rozwój dedykowanych systemów mapowania przyczyni się do szerszego wykorzystywania przedmiotowego systemu bazodanowego.

1.3 Zawartość pracy

Rozdział 2 opisuje podstawy teoretyczne związane z silnikiem Apache Cassandra, które są niezbędne do zrozumienia całej pracy. W sekcji 2.1 Autor przedstawia stosowany w silniku model danych. Sekcja 2.2 opisuje sposób dystrybucji wierszy w klastrze. W punkcie 2.5 zostały omówione, na przykła-

dzie, różnice pomiędzy relacyjnym modelem danych a schematem Cassandry. Sekcja 2.6 opowiada o języku zapytań CQL, natomiast w punkcie 2.8 zostało wprowadzone pojęcie wzorców i antywzorców modelowania.

W rozdziale 3 zostały przedstawione informacje teoretyczne na temat mapowania obiektowo-relacyjnego. W punkcie 3.1 omówione są zasady działania mechanizmu na przykładzie *persistence API* języka Java. Kolejna sekcja (3.2) przedstawia wyniki badania wydajności istniejącej implementacji JPA dla Apache Cassandry. Na podstawie uzyskanych wyników Autor formuuje wnioski zaprezentowane w punkcie 3.3, które są przyczynkiem do przedstawienia autorskiej koncepcji mapowania opisanej w punkcie 3.5.

Rozdział 4 zawiera opis wszystkich unikalnych cech, które wyróżnia zaproponowane przez Autora narzędzie do modelowania dziedziny w Cassandrze. W sekcji 4.1 wypisane są mechanizmy, dla których wsparcie posiada przygotowany interfejs. Punkt 4.3 wprowadza w podstawowe pojęcia związane z biblioteką. Sekcja 4.4 przedstawia sposób definiowania modelu. Punkt 4.5 to obszerne rozważania na temat modelowania zależności pomiędzy danymi, wraz z kompleksowymi wynikami badań, które pomagają wybrać poprawny model w zależności od scenariusza użycia. W sekcji 4.6 zaprezentowane zostały wszystkie wzorce projektowania schematu, dla których wsparcie udostępnia interfejs modelowania. Punkty 4.7 oraz 4.8 omawiają zakres obsługi specyficznych mechanizmów silnika bazodanowego. W sekcji 4.9 autor przedstawia mechanizm migracji danych dedykowany dla Apache Cassandry. Ostatni punkt rozdziału (4.10) przedstawia narzędzia do profilowania aplikacji udostępnione przez system modelowania obiektowego.

Rozdział 2

Apache Cassandra

Apache Cassandra jest bazą danych NoSQL[1], która powstała w wyniku połączenia rozwiązań wykorzystywanych w Dynamo[2] oraz BigTable[3]. Cassandra początkowo była rozwijana dla potrzeb portalu społecznościowego Facebook. Baza danych powstała z myślą o rozwiązaniu problemu pełnotekstowego przeszukiwania skrzynek odbiorczych użytkowników, w których dziennie zapisywane były miliardy wiadomości. Głównym celem, do którego dążyli twórcy Cassandry, była możliwość wykorzystania jej do przechowywania ogromnych ilości danych w bardzo rozproszonym środowisku, gdzie awarie pojedynczych węzłów zdarzają się na porządku dziennym. W tych warunkach baza danych musi zapewniać szybki i niezawodny dostęp do danych. [2]

Apache Cassandra wykorzystywana jest w wielu serwisach na całym świecie. Najbardziej znaczące przykłady użycia produkcyjnego to eBay, Instagram oraz Github[4]. Największa światowa instalacja Cassandry obejmuje oko-

[1]NoSQL (ang. Not Only SQL) - podzbiór baz danych, które zapewniają inne sposoby modelowania dziedziny niż tradycyjny model oparty na tabelach i relacjach.

[2]Amazon DynamoDB - zdecentralizowana, wysoce skalowalna baza typu klucz-wartość.

[3]Google BigTable - rozproszony system bazodanowy, który dobrze skaluje się dla ogromnych ilości danych.

[4]eBay, Instagram, Github - przykłady dużych portali internetowych. eBay to serwis aukcyjny, Instagram to portal społecznościowy oparty o publikację zdjęć wykonanych telefonami komórkowymi, a Github to usługa pozwalająca na przechowywanie i wersjonowanie kodu źródłowego aplikacji.

ło 15000 węzłów, na których przechowywane jest łącznie ponad 4 petabajty danych. [3]

W przeciwieństwie do relacyjnych baz danych, Apache Cassandra nie zapewnia wsparcia dla reguły ACID[5]. Zamiast tego zostały zrealizowane postulaty twierdzenia CAP: „we współdzielonym systemie plików można zachować maksymalnie dwie z trzech właściwości: spójności, dostępności oraz podatności na partycjonowanie". [4] Apache Cassandra priorytetyzuje właściwości dostępności oraz partycjonowania. Spójność danych jest odwrotnie proporcjonalna i może być regulowana w zależności od czasu odpowiedzi. Wysoka spójność danych oznacza wolniejszą odpowiedź bazy.

2.1 Model danych

Model danych Apache Cassandra jest analogiczny do BigTable. [5] Można przedstawić go jako dwuwymiarowa mapa trójek wartości:

```
Map<RowKey, Map<ColumnKey, Triple<Value, Timestamp, TTL>>>
```

gdzie `RowKey` to identyfikator wiersza, `ColumnKey` to identyfikator kolumny, `Value` to wartość komórki, `Timestamp` to czas aktualizacji komórki, a `TTL` to czas życia danej wartości. [6] Na rysunku 2.1 przedstawiona jest schematyczna ilustracja wiersza danych. Pogrubiona wartość w lewej komórce to klucz wiersza, natomiast wyróżnione nagłówki oznaczają klucze poszczególnych kolumn. Każda komórka składa się z trzech elementów: wartości, czasu życia oraz „odcisku czasu".

2.2 Dystrybucja danych

Do dystrybucji danych wykorzystywana jest funkcja skrótu, która zachowuje kolejność elementów. Węzły są rozmieszczone w topologii pierścienia. Algo-

[5]ACID (ang. Atomic, Consistency, Isolation, Durability) - zasada atomowości, spójności, izolacji i trwałości. Wymienione cechy gwarantują poprawne przetwarzanie transakcji w bazach danych.

	ABC	**DEF**	···	**XYZ**
123	test value	another test value	···	not a test value
456	20	∞	···	∞
	1291987837942000	1291980736812000	···	1291980736212000

Rysunek 2.1: Przykładowy wiersz modelu danych o identyfikatorze 123456. Wartość komórki (123456, DEF) to „another test value".

rytm dystrybucji zostanie omówiony na przykładzie ze schematu 2.2:

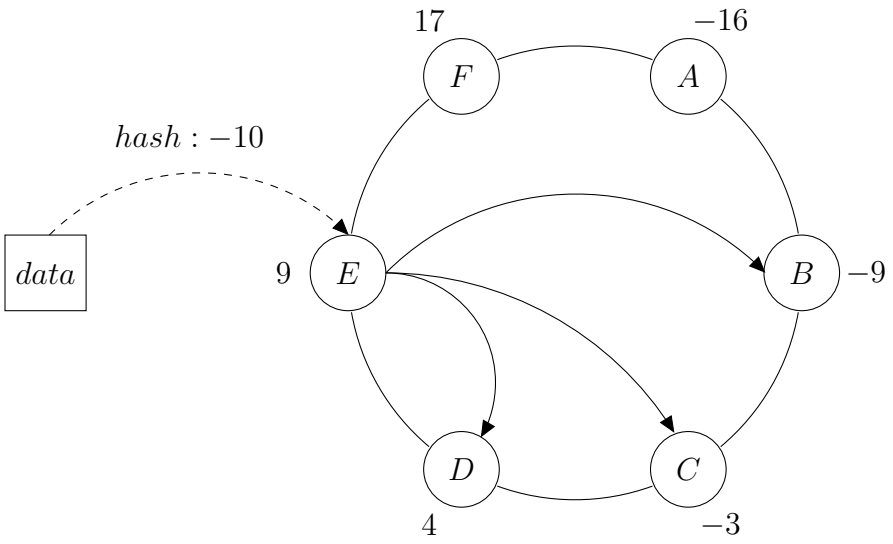
1. Każdemu z węzłów $\{A, B, C, D, E, F\}$ przypisywany jest token, który zawiera się w zakresie wartości przyjmowanych przez funkcję skrótu. Strategię wyboru tokena można konfigurować. Przykładową strategią jest wybór losowy. W omawianym przykładzie węzłom zostały przypisane tokeny o wartościach $\{-16, -9, -3, 4, 9, 17\}$.

2. Użytkownik bazy danych przesyła żądanie do dowolnego węzła, który pełni funkcję koordynatora dla danej operacji. Koordynator nadzoruje wpisanie danych do odpowiednich węzłów. W omawianym przykładzie rolę koordynatora pełni węzeł E.

3. Każdy węzeł przechowuje dane, których funkcja skrótu zawiera się w przedziale $(token_{n-1}, token_n]$, gdzie n to numer kolejny węzła rosnący zgodnie z ruchem wskazówek zegara. W przykładzie węzeł C przechowuje wiersze o wartościach funkcji skrótu z przedziału $(-9, -3]$, natomiast węzeł D z przedziału $(-3, 4]$. Wartości funkcji obliczane są cyklicznie, stąd węzeł A przechowuje wiersze o skrócie z przedziału $(-\infty, -16] \cup (17, \infty)$. W przykładzie wiersz o kluczu z funkcją skrótu wartości -10 zostanie utrwalony na węźle B.

4. Dane replikowane są na n węzłach, gdzie n to wartość konfigurowalnego współczynnika replikacji. Poza węzłem macierzystym (wyznaczanym w punkcie 3 algorytmu) dane są replikowane na $n-1$ kolejnych (zgodnie

z ruchem wskazówek zegara) węzłach. W omawianym przykładzie dane zostaną zreplikowane na węzłach C i D.

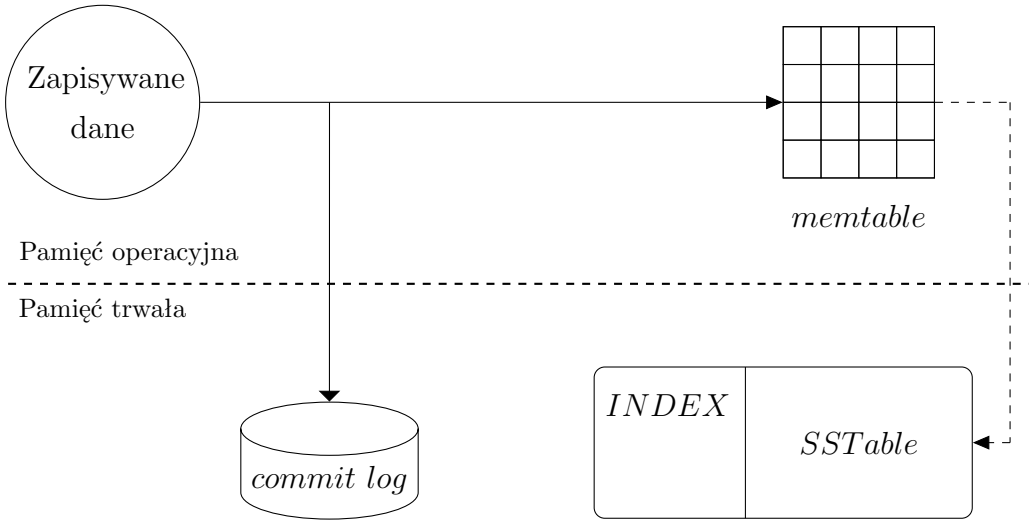


Rysunek 2.2: Schematyczna ilustracja dystrybucji danych w bazie danych Apache Cassandra.

2.3 Algorytmy zapisu/usuwania danych

Algorytm zapisu danych w Apache Cassandra został schematycznie przedstawiony na diagramie 2.3. [7] Składa się on z następujących kroków:

1. Do węzła przesyłane są dane, które mają zostać na nim zapisane.
2. Dane kopiowane są do dwóch struktur:
   - *memtable* - przechowuje wiersze w pamięci operacyjnej,
   - *commit log* - przechowuje informacje o kolejnych zapisach wykonywanych do bazy danych.



Rysunek 2.3: Schematyczna ilustracja algorytmu zapisu danych dla Cassandry.

3. Podstawową strukturą do pobierania danych jest *memtable*. Jest ona umieszczona w szybkiej pamięci, zapewnia więc krótki czas dostępu do danych. Wykorzystanie trwałej struktury *commit log* pozwala na odtworzenie zawartości *memtable* w przypadku awarii wymagającej nagłego restartu maszyny, na przykład braku zasilania. Przy ponownym uruchomieniu węzła Cassandry odtwarza on kolejno wszystkie zapisy, które znajdują się w *commit logu*.

4. W przypadku przepełnienia *memtable* wykonywane jest „spłukiwanie" zawartości pamięci na dysk do struktury nazywanej *SSTable*. Poza danymi zawiera ona indeks, który pozwala na szybki dostęp do danych wierszy. Jej charakterystyczną cechą jest niezmienność. Po przepisaniu danych z *memtable* do *SSTable* jej zawartość nie jest już modyfikowana. Z tego względu wiersze o tym samym kluczu są często podzielone pomiędzy wiele takich struktur. „Spłukiwanie" może zostać uruchomione manualnie z wykorzystaniem polecenia `nodetool flush`.

5. Ostatnim etapem zapisu danych jest kompresja. Aktualizacja danych

nie może nadpisywać istniejących wierszy, bo struktura *SSTable* jest niezmienna. Zamiast tego wstawia ona rekordy z późniejszym „odciskiem czasu". Powoduje to nadmiarowe zużycie przestrzeni dyskowej. Kompresja pozwala usunąć nieaktualne wierwsze poprzez przepisanie istniejących *SSTable* na nowe, uporządkowane struktury. Kompresja może być wywoływana automatycznie w zależności od wybranej strategii lub manualnie, z wykorzystaniem polecenia `nodetool compact`.

Niemodyfikowalność struktur *SSTable* ma swoje konsekwencje także dla operacji usuwania danych. Wiersz nie może zostać fizycznie skasowany. Zamiast tego oznacza się go specjalnym znacznikiem *tombstone*. Wiersze oznaczone tą flagą są usuwane na etapie kompresji danych.

2.4 Obszary zastosowania Cassandry

Specyficzny schemat danych Apache Cassandry i brak wielu mechanizmów znanych z relacyjnych systemów bazodanowych powodują, że nie jest ona najlepszym wyborem do przechowywania uniwersalnych danych:

- Brak wsparcia dla transakcji znacząco utrudnia wykorzystanie Cassandry w dziedzinach, dla których spójność danych jest kwestią kluczową. Przykładem może być obszar finansowy. W systemach relacyjnych spójność jest zapewniona poprzez mechanizmy bazy danych. W przypadku Apache Cassandry odpowiedzialność zostaje przeniesiona na aplikacje dostępowe, co znacząco zwiększa ilość pracy koncepcyjnej i zwiększa ryzyko błędów.

- Brak możliwości złączania powoduje, że model danych musi być projektowany w oparciu o wykonywane do niego odwołania. Utrudnia to rozbudowę aplikacji. Przykładowo dla relacji jeden do wielu modelowanej poprzez tabelę, w której identyfikatorem wiersza jest identyfikator obiektu nadrzędnego, a w kolumnach wpisywane są identyfikatory obiektów podrzędnych, nie jest możliwe zwrócenie wszystkich obiektów

nadrzędnych wskazujących na dany obiekt podrzędny bez modyfikacji schematu.

- Brak złączeń implikuje denormalizację modelu danych. Wprowadza to problemy z zachowaniem spójności. Aby zaktualizować adres, który jest przechowywany w formie zdenormalizowanej w tabeli użytkownika oraz zamówienia, należy znaleźć wszystkie rekordy, które wskazują na ten adres. Odpowiedzialność za spójność danych ponownie zostaje przeniesiona na aplikację, która wykorzystuje bazę.

W tabeli 2.4 przedstawiono porównanie różnych aspektów Apache Cassandry i relacyjnych systemów bazodanowych. Analizując informacje zebrane w tabeli i przedstawione wcześniej w rozdziale można wyciągnąć następujące wnioski:

- Modelowanie dziedziny danych jest znacznie prostsze w przypadku systemów relacyjnych. Na etapie projektowania nie trzeba brać pod uwagę wykonywanych odwołań, potencjalnych problemów niespójności, a także możliwych kierunków rozbudowy aplikacji.
- Apache Cassandra najlepiej sprawdza się w zastosowaniu do *big data*. Do poszczególnych fragmentów definicji wielkich zbiorów danych, która została przytoczona w sekcji 1.2, można przyporządkować cechy charakteryzujące Cassandrę:
  - „dużych rozmiarów" - liniowa skalowalność horyzontalna pozwala na obsługę informacji o dowolnej wielkości, pod warunkiem zapewnienia odpowiedniej liczby węzłów,
  - „różnorodnych" - model danych nie jest sztywny, może być dostosowany do zmieniających się danych w trakcie działania aplikacji,
  - „często zmieniających się" - wbudowana obsługa szeregu chronologicznego oraz optymalizacja czasu zapisu zostały zaprojektowane pod kątem dynamicznie zmieniających się danych.

	Apache Cassandra	**RDBMS**
Skalowalność horyzontalna	Liniowa skalowalność horyzontalna.	Pod warunkiem wykorzystania specjalnych narzędzi i/lub technik, na przykład *shardingu*[6].
Mechanizmy zachowywania spójności	Brak mechanizmów zachowywania spójności danych.	Mechanizmy i model zapewniające wysoką spójność danych. Wsparcie dla transakcyjności; znormalizowany model.
Odporność na awarie	Wysoka odporność na awarie zapewniona między innymi przez replikację danych pomiędzy węzłami.	Niska odporność na defekty. Pojedynczy punkt awarii.
Optymalizacja operacji	Optymalizacja pod kątem szybkości zapisu.	Optymalizacja pod kątem szybkości wykonywania zapytań.
Koncepcja	Projekt z myślą o opisywaniu szeregu chronologicznego danych. [8]	Projekt do opisu uniwersalnych, ustrukturyzowanych danych.
Przystosowanie modelu danych	Model odpowiedni dla dynamicznych struktur danych zmieniających się w trakcie działania systemu. [9]	Model odpowiedni dla niezmiennych struktur danych.

Rysunek 2.4: Porównanie Apache Cassandry z relacyjnymi bazami danych.

[6]Sharding (od *shard* - ang. kawałek) - technika polegająca na podziale zbioru danych na niezależne partycje w zależności od ich cech, na przykład grupując użytkowników według lokalizacji geograficznej. [10]

Według twórców Cassandry [11] typowymi obszarami jej zastosowań są:

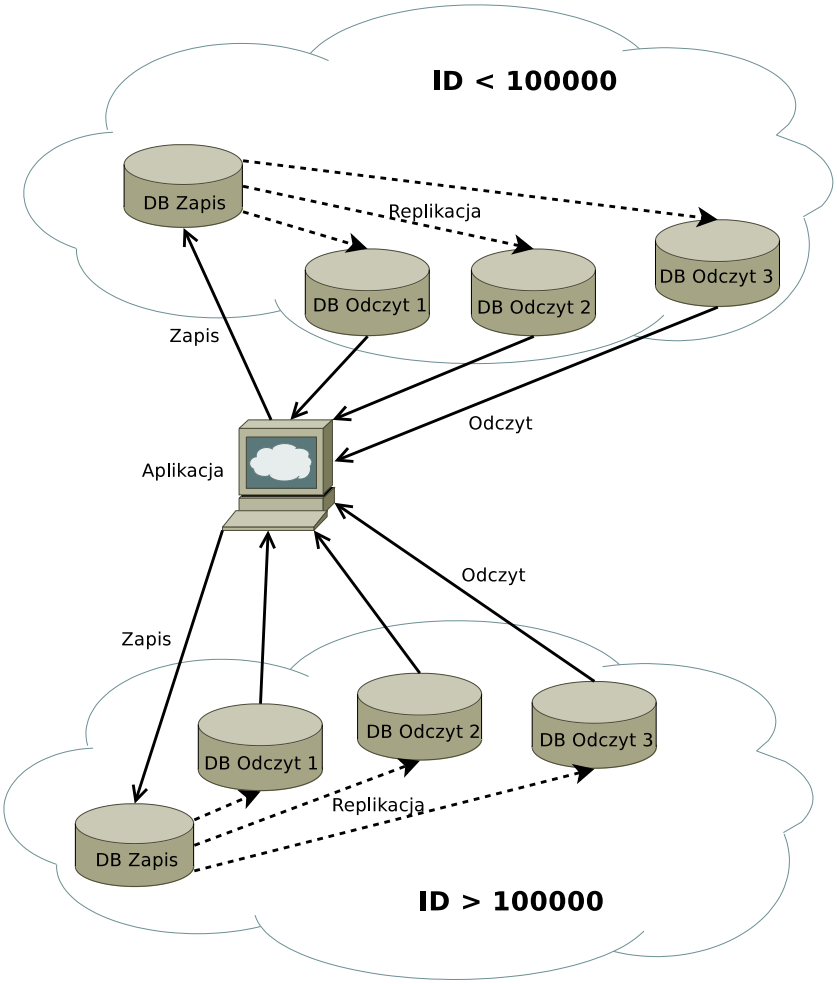
- Aplikacje wykonujące OLTP[7], a więc transakcje charakteryzujące się wysoką współbieżnością, krótkim czasem odpowiedzi, małymi zapytaniami oraz pracą na dużych zbiorach danych. [12]
- Zarządzanie danymi uszeregowanymi w czasie.
- Pobieranie i analiza danych z urządzeń o wysokiej częstotliwości odświeżania informacji.
- Aplikacje typu SaaS[8] oparte o intensywne wykorzystanie usług sieciowych.
- Zarządzanie mediami strumieniowymi (muzyka, filmy) oraz nieustrukturyzowanymi danymi (dobrym przykładem są portale społecznościowe).
- **Systemy „zapisochłonne”**.

Z przedstawionych scenariuszy Autor zdecydował się wyróżnić ostatni. Istotnym zagadnieniem w przypadku wielkich danych jest operacja zapisu, która musi być wykonywana w czasie rzeczywistym. Aby omówić problem należy przedstawić typową architekturę wykorzystywaną w relacyjnych bazach danych do przechowywania wielkich danych. [13] Prezentuje ją diagram 2.5.

Na diagramie przedstawiono aplikację, która odwołuje się do relacyjnej bazy danych. Została ona podzielona na dwie partycje względem identyfikatora wiersza (mniejszy lub większy od 100000). Każda partycja zawiera jeden węzeł, który wykonuje operacje zapisu (`DB Zapis`). Do niego odwołuje się

[7] On-Line Transaction Processing - przetwarzanie transakcji sieciowych.

[8] Software as a Service (ang. oprogramowanie jako usługa sieciowa) - aplikacje, które mogą być wykorzystywane przez sieć bez pobierania i instalacji na stacji roboczej. [14] Przykładem takiej aplikacji jest Google Docs (`https://docs.google.com/`), który umożliwia edycję dokumentów, arkuszy kalkulacyjnych oraz prezentacji z wykorzystaniem przeglądarki internetowej.



Rysunek 2.5: Schematyczna ilustracja partycjonowania relacyjnej bazy danych (określanego angielskim terminem *sharding*).

aplikacja z żądaniem zapisu danych. W każdej partycji znajdują się również trzy węzły, które pozwalają na odczyt danych (`DB Odczyt`). Aplikacja zwraca się o odczyt danych kolejno do różnych węzłów. Pozwala to zbalansować obciążenie. Należy zauważyć, że w przedstawionej architekturze aplikacja musi wiedzieć, do której partycji należy się odwołać aby pobrać odpowiednie dane.

Proces skalowania systemu o przedstawionej architekturze jest dwustopniowy:

1. Skalowanie operacji odczytu. Może zostać przeprowadzone dla każdej

partycji osobno. Polega na dodaniu nowego węzła, który służy do odczytu danych. Jest to stosunkowo proste, należy jednak uwzględnić dodany punkt dostępu w puli serwerów, do których odwołuje się aplikacja.

2. Skalowanie operacji zapisu. W tym przypadku nie wystarczy dodanie drugiego węzła odpowiedzialnego za zapis, gdyż dane i tak muszą zostać zreplikowane pomiędzy węzłami. Z tego względu obciążenie obu serwerów pozostałoby bez zmian. Skalowanie operacji zapisu to skomplikowany proces wymagający stworzenia nowej partycji i modyfikacji reguł przydzielania rekordów do poszczególnych fragmentów.

Podobna architektura wykorzystywana jest w komercyjnych rozwiązaniach, takich jak `MySQL Cluster CGE`[9]. Występują tam dodatkowe warstwy pośredniczące, które ukrywają szczegóły wykonywania zapytań przed aplikacją. Sama zasada działania pozostaje jednak analogiczna.

Problemem w tej architekturze pozostaje skalowanie operacji zapisu. Może ono zostać wykonane poprawnie tylko pod warunkiem, że istnieje funkcja partycjonująca dla danego natężenia operacji zapisu, która dzieli całość danych na dostatecznie wydajne (a więc małe) fragmenty. Okazuje się, że w praktyce nie zawsze jest to możliwe. Właśnie ten problem przyczynił się do powstania projektu Apache Cassandra w Facebooku.

Wszystkie przypadki użycia wymienione przez twórców Cassandry pokrywają scenariusze, w których ich system bazodanowy sprawdza się lepiej niż podejście relacyjne w subiektywnym odczuciu. Omówiony szerzej przypadek z zapisem jest jednak jedynym, który z przyczyn technologicznych może być nierealizowalny w świecie SQL-owych baz danych.

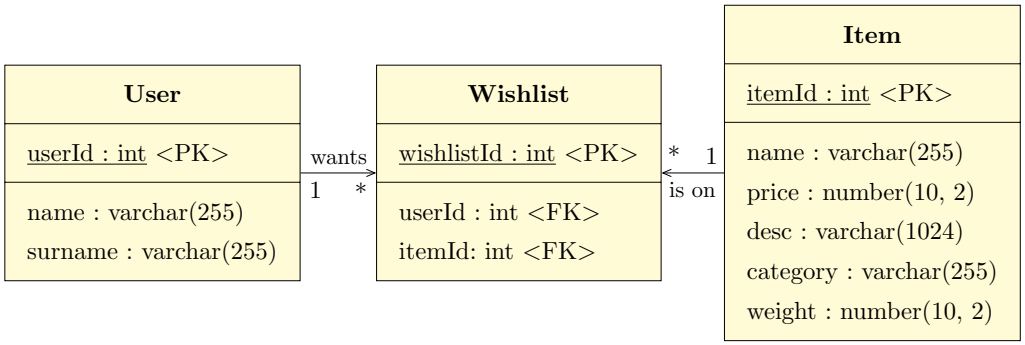
2.5 Struktura danych a modelowanie

Struktura i mechanizm dystrybucji danych wykorzystywany w Apache Cassandrze zmieniają podejście do modelowania dziedziny znane z relacyjnych

[9]Strona domowa projektu - `http://www.mysql.com/products/cluster/`.

baz danych. Zbudowanie efektywnego modelu danych Cassandry wymaga skupienia się w podobnym stopniu na zdefiniowaniu encji z modelowanego świata, jak również na analizie odwołań, które będą wykonywane do obiektów z tego świata. [15]

Załóżmy, że celem jest modelowanie danych dla sklepu internetowego. Zakupów dokonują użytkownicy, którzy mogą wstawić wiele przedmiotów z oferty sklepu na listę życzeń. W przypadku baz opartych o język SQL jest to klasyczny problem relacji typu wiele-do-wielu, do modelowania których wykorzystywana jest najczęściej tabela pośrednia.



Rysunek 2.6: Modelowanie listy życzeń w relacyjnej bazie danych.

Diagram prezentujący zamodelowaną relację dla listy życzeń jest przedstawiony na rysunku 2.6. W tabeli Użytkownik (*User*) przechowywane są imię, nazwisko oraz identyfikator. W tabeli Przedmiot (*Item*) znajduje się nazwa, cena, a także inne właściwości: opis, kategoria oraz waga. Tabela Lista życzeń (*Wishlist*) łączy ze sobą użytkownika i przedmiot poprzez wykorzystanie kluczy obcych.

Powyższy model jest wykorzystywany w widoku listy życzeń na profilu użytkownika. Na liście życzeń prezentowane są informacje o nazwie przedmiotu oraz jego cenie. Po kliknięciu nazwy użytkownik przenoszony jest do strony przedmiotu. Na listingu 2.7 zaprezentowano zapytanie, które wyświetla listę życzeń.

Cassandra umożliwia utworzenie dokładnej repliki relacyjnego modelu danych. Zostało to przedstawione na rysunku 2.8.

```
SELECT item.name, item.price
FROM Item item, Wishlist wishlist
WHERE wishlist.userId = 202;
```

Rysunek 2.7: Zapytanie, które pobiera wszystkie przedmioty na liście życzeń użytkownika o identyfikatorze 202.

User

	name	**surname**
123	Janusz	Kowalski

	name	**surname**
124	Marcin	Nowak

Wishlist

	userId	**itemId**
51	123	579

	userId	**itemId**
52	124	232

Item

	name	**price**	**desc**	**category**	**weight**
232	Master Chef	20.34	Recipes	BOOKS	0.2

	name	**price**	**desc**	**category**	**weight**
579	Seat Hit	159.99	Armchair	FURNITURE	10.8

Rysunek 2.8: Wynik błędnego przeniesienia relacyjnego modelu danych do Cassandry.

Taki model jest jednak niepoprawny. Nie umożliwia on filtrowania zawartości listy życzeń po identyfikatorze użytkownika. Wynika to z faktu, że pobranie odpowiednich wierszy listy życzeń wymaga znajomości ich identyfikatorów, podczas gdy widok dysponuje wyłącznie odniesieniem do użytkownika. Błąd ten można łatwo naprawić zastępując encję *Wishlist* encją *WishlistByUser*, co przedstawia diagram 2.9.

Poprawiony model można poddać dalszej optymalizacji. Wyświetlenie listy życzeń użytkownika wymaga odwołania do encji *Item*, w której znajdują się informacje o nazwie i cenie przedmiotu. Ponieważ przedmioty mogą być rozmieszczone na różnych węzłach, silnik Cassandry nie może wykonać złączenia w sposób optymalny - zapytanie o każdą pozycję listy życzeń jest wykonywane osobno. W celu przyspieszenia wykonywania operacji należy wykonać denormalizację. Dołączając do encji *WishlistByUser* informacje o nazwie

WishlistByUser

	579
123	-
	232
124	-

Rysunek 2.9: Definicja encji listy życzeń umożliwiająca filtrowanie względem użytkownika.

i cenie produktu można uniknąć wykonywania kosztownych złączeń. Pozostałe dane przedmiotu zostaną pobrane dopiero po przejściu na jego stronę. Efekt denormalizacji jest przedstawiony na diagramie 2.10.

WishlistByUser

	579
123	(„Master Chef”, 20.43)
	232
124	(„Seat Hit”, 159.99)

Rysunek 2.10: Zdenormalizowana postać listy życzeń.

2.6 CQL

Efektywne modelowanie i obsługa danych w Apache Cassandra wymaga dobrej znajomości wewnętrznej struktury bazy danych. Dodatkowym utrudnieniem przy korzystaniu z początkowych wersji Cassandry była konieczność wykorzystania skomplikowanego interfejsu programistycznego opartego o wywoływanie zdalnych procedur Thrift[10]. Thrift jest platformą pozwalającą budować aplikacje przenośne między różnymi językami programowania. Dzięki temu rozwiązaniu baza danych dostępna była dla wszystkich platform. Niestety, skutkiem ubocznym było skomplikowanie interfejsu dostępowego.

[10]Dokumentacja interfejsu dostępna jest pod adresem `https://wiki.apache.org/cassandra/API10`.

Wraz z wydaniem 1.2 Apache Cassandry wprowadzony został nowy interfejs dostępu do tej bazy danych. Interfejs ten nosi nazwę CQL[11] i jest językiem zapytań, którego składnia wzorowana jest na SQL. Poza podobieństwami składniowymi języki te nie mają cech wspólnych. Nie są wzajemnie zgodne. W chwili obecnej CQL jest rekomendowanym standardem komunikacji z Apache Cassandra. [16] Na listingu 2.11 przedstawiono zapytanie w języku CQL, które opisuje omawianą wcześniej encję *User*. Wynikiem wykonania tego zapytania jest prosty schemat modelu danych zaprezentowany na diagramie 2.12.

```
CREATE TABLE User (
    userId uuid PRIMARY KEY,
    name text,
    surname text);
```

Rysunek 2.11: Zapytanie CQL, które tworzy encję *User*.

User		**name**	**surname**
	userId	null	null

Rysunek 2.12: Wynik wykonania zapytania 2.11.

2.7 Przykład modelowania dziedziny danych

W Internecie znajdują się przykładowe modele danych stworzone przez użytkowników Apache Cassandry. Zapewniają one materiał do nauki dla nowych użytkowników bazy. Prezentują sprawdzone sposoby rozwiązania zadanego problemu koncentrując się na omówieniu decyzji projektowych, które należy podjąć aby zbudować efektywny schemat. Proces tworzenia wydajnych modeli danych zostanie przez Autora przedstawiony z wykorzystaniem takiego przykładu. Do modelowania zostanie wykorzystany język CQL, gdyż

[11]CQL (ang. Cassandra Query Language) - język zapytań Cassandry.

począwszy od wersji CQL3 jest on oficjalnym standardem zalecanym przez autorów Cassandry.

2.7.1 Twissandra

Projekt Twissandra został stworzony przez Tylera Hobbsa i opublikowany pod adresem `https://github.com/twissandra/twissandra`. Według informacji na stronie „Twissandra jest przykładowym projektem stworzonym do nauki i zaprezentowania jak używać Cassandry. Po uruchomieniu projektu zaprezentowana zostanie strona internetowa, która posiada funkcjonalność podobną do serwisu Twitter". [17]

Serwis Twitter[12] pozwala na publikację krótkich wiadomości tekstowych o długości maksymalnie 160 znaków. Jego funkcjonalność jest wzorowana na SMS[13]. Różnica polega na tym, że wiadomość nie jest kierowana do konkretnego adresata (adresatów), ale publicznie dostępna dla wszystkich odwiedzających profil danego użytkownika.

Głównym widokiem w Twitterze jest oś czasu. Znajdują się na niej wpisy ułożone w odwrotnym porządku chronologicznym. Istnieją trzy typy osi czasu:

Publiczna przedstawia wpisy wszystkich użytkowników.

Użytkownika przedstawia wpisy danego użytkownika oraz wszystkich śledzonych przez niego osób.

Profilowa przedstawia wyłącznie wpisy danego użytkownika.

Pierwszym krokiem jest zdefiniowanie modelu użytkownika. Dla uproszczenia encja posiadać będzie wyłącznie dwa parametry - nazwę oraz hasło - oba zapisywane jako tekst. Istotną kwestią jest wybór klucza głównego. W przypadku relacyjnych baz danych identyfikatory najczęściej tworzone są

[12]Dostępny pod adresem `https://twitter.com`.

[13]Short Message Service (ang. usługa krótkich wiadomości) - usługa pozwalająca na przesyłanie pomiędzy telefonami komórkowymi krótkich (do 160 znaków) wiadomości. [18]

w sposób sztuczny, a odpowiednie wiersze wybierane są z zastosowaniem zapytań. W przypadku Cassandry wybór klucza głównego jest dużo bardziej istotny:

- Cassandra nie posiada mechanizmu generacji unikatowego klucza głównego. Zamiast tego wykorzystywane są typy `uuid` lub `timeuuid`, [19] które są obliczane przez aplikację kliencką.
- Ze względu na architekturę Cassandra nie potrafi efektywnie pobrać wszystkich wierszy należących do danej tabeli. Oznacza to, że bez używania indeksów (co wiąże się z tworzeniem kolejnych tabel) **jedyną możliwością odwołania się do wiersza jest podanie jego identyfikatora**. Wybór losowo/sekwencyjnie generowanego klucza może sprawić, że pobieranie danego wiersza będzie bardzo utrudnione.

Dla modelu użytkownika dobrym identyfikatorem jest nazwa (zakładając jej unikatowość). Jest to dana, która będzie znana w dowolnym kontekście odwołania. Na listingu 2.13 przedstawiono zapytanie, które pozwala utworzyć i wstawić użytkownika do bazy danych. W wyniku tego zapytania zostanie utworzona tabela o strukturze przedstawionej na diagramie 2.14.

```
CREATE TABLE users (
    username text PRIMARY KEY,
    password text)
INSERT INTO users (username, password) VALUES (
    'jturek',
    'UnsafePassword')
```

Rysunek 2.13: Tworzenie i wstawianie użytkownika w Twissandrze.

Kolejnym krokiem jest umożliwienie śledzenia użytkowników. W relacyjnym modelu danych do przechowywania takiej informacji można użyć tabeli `followers`, która posiada trzy kolumny: sztuczny klucz główny, identyfikator śledzonego użytkownika oraz identyfikator śledzącego. Ta sama architektura

	password
jturek	UnsafePassword

Rysunek 2.14: Poglądowy wynik struktury uzyskanej po wykonaniu zapytań 2.13.

w Cassandrze będzie błędna, gdyż nie da się zadać zapytania, które zwróci wszystkich użytkowników śledzących profil o zadanej nazwie:

- Nie ma możliwości efektywnego pobrania wszystkich wpisów z tabeli `followers`, a więc nie da się ich filtrować po nazwie śledzonego użytkownika.
- Istnieje możliwość stworzenia indeksu na kolumnie z identyfikatorem śledzonego użytkownika. Dzięki temu możliwe będzie wybranie zadanych wpisów w tabeli `followers`.
- Indeks tworzy niewidoczną tabelę, w której kluczem głównym wiersza jest jedna z unikalnych wartości komórki (w podanym przypadku identyfikator śledzonego użytkownika), a w kolumnach przechowywane są identyfikatory wszystkich wierwszy, które posiadają taką wartość komórki.

Analizując opis metody tworzenia indeksów łatwo dojść do wniosku, że można przekształcić model tabeli `followers` w taki sposób, aby ich używanie nie było potrzebne. Przyjmując identyfikator śledzonego użytkownika jako klucz wiersza, a informację o nazwach profili śledzących przechowując w kolejnych kolumnach, możliwe jest pobranie kompletu informacji w jednym zapytaniu.

CQL3 umożliwia wykorzystanie dwóch typów kluczy do dwuwymiarowego podziału wierszy [20]:

Partition key określa, na którym węźle znajduje się dany wiersz. W niskopoziomowym schemacie danych jest implementowany na kluczu wiersza.

Clustering key określa kolumny, względem których porządkowane są wiersze dla danej partycji. W niskopoziomowym schemacie danych jest implementowany na kolumnach.

Dzięki wykorzystaniu dwóch rodzajów kluczy można zdefiniować efektywny schemat dla tabeli `followers`. Odpowiednie zapytanie zostało przedstawione na listingu 2.15. Poglądowy wygląd fizycznego schematu danych uzyskanego dla omawianego zapytania został zaprezentowany na diagramie 2.16.

```
CREATE TABLE followers (
    username text,
    follower text,
    since timestamp,
    PRIMARY KEY (username, follower))
INSERT INTO followers (username, follower, since) VALUES (
    'jturek',
    'jkowalski',
    1412020800)
INSERT INTO followers (username, follower, since) VALUES (
    'jturek',
    'mnowak',
    1409601600)
```

Rysunek 2.15: Śledzenie użytkowników w Twissandrze.

	follower:jkowalski	**follower:mnowak**
jturek	1412020800	1409601600

Rysunek 2.16: Poglądowy wynik struktury uzyskanej po wykonaniu zapytań 2.15.

W zaprezentowanym przykładzie kolumna `username` pełni rolę klucza `partition`, natomiast `follower` to klucz typu `clustering`. Dla powyższego

przypadku można wypisać wszystkie wiersze tabeli `followers` dla użytkownika o kluczu `jturek`. Odpowiednie zapytanie zostało zaprezentowane na listingu 2.17. Wynik tego zapytania przedstawia tabela 2.18.

```
SELECT * FROM followers WHERE username = 'jturek'
```

Rysunek 2.17: Zapytanie, które wybiera wszystkie elementy tabeli *follower* dla użytkownika *jturek*.

username	**follower**	**since**
jturek	jkowalski	1412020800
jturek	mnowak	1409601600

Rysunek 2.18: Wybór wszystkich wpisów tabeli *follower* dla użytkownika *jturek*.

Usunięcie kolumny `follower` z klucza utworzonej tabeli spowodowałby, że dany użytkownik mógłby być śledzony tylko przez jedną osobę. Z kolei przesunięcie tej kolumny do klucza typu `partition` pozwoliłoby pobierać wiersze wyłącznie przy podaniu kombinacji wartości `username` oraz `follower`, co nie jest intencją omawianego modelu.

Tabela `followers` umożliwia reprezentację relacji śledzenia tylko w jedną stronę. Do realizacji widoku, na którym będą wyświetlone wszystkie wpisy osób śledzonych przez użytkownika, wymagana jest informacja odwrotna. Autor Twissandry sugeruje stworzenie komplementarnej tabeli `friends`, w której zapisane będą nazwy osób śledzonych przez dany profil. W omawianym przypadku równie wydajnym rozwiązaniem jest wykorzystanie indeksów zaimplementowanych w CQL. Stworzenie indeksu na kolumnie `follower` w tabeli `followers` przy pomocy zapytania przedstawionego na listingu 2.19 umożliwia wyszukanie wszystkich osób śledzonych przez danego użytkownika. Przykładowe zapytanie realizujące takie wyszukanie zostało zaprezentowane na listingu 2.20.

Kolejnym elementem modelu jest definicja tabeli, która umożliwia przechowywanie wpisów. Pełni ona rolę normalizacyjną i nie będzie wykorzysty-

```
CREATE INDEX ON followers (follower)
```

Rysunek 2.19: Zapytanie, które tworzy indeks na kolumnie *follower* w tabeli *followers*.

```
SELECT * FROM followers WHERE follower = 'jkowalski'
```

Rysunek 2.20: Zapytanie, które wybiera wszystkie osoby śledzone przez użytkownika *jkowalski* dzięki indeksowi.

wana w widokach osi czasu. W tym przypadku można posłużyć się sztucznym identyfikatorem, gdyż pobieranie wierszy następować będzie wyłącznie w kontekście znanego klucza wpisu. Zapytania pozwalające stworzyć strukturę tabeli oraz wstawić do niej przykładowy wpis zostały zaprezentowane na listingu 2.21. Poglądową strukturę fizyczną uzyskaną w wyniku ich wykonania prezentuje natomiast diagram 2.22.

```
CREATE TABLE tweets (
    tweet_id uuid PRIMARY KEY,
    user text,
    body text)
INSERT INTO tweets (tweet_id, user, body) VALUES (
    f8122d00-2f99-11e4-babf-0002a5d5c51b,
    'jturek',
    'Sample')
```

Rysunek 2.21: Tabela wpisów w Twissandrze.

	user	**body**
f8122d00-2f99-11e4-babf-0002a5d5c51b	'jturek'	'Sample'

Rysunek 2.22: Poglądowy wynik struktury uzyskanej po wykonaniu zapytań 2.21.

Tabela `tweets` nie może być wykorzystania do konstrukcji widoków osi czasu. Aby wyświetlić wszystkie wpisy użytkownika i śledzonych przez niego

osób należałoby złączyć trzy tabele - `tweets`, `users` oraz `followers`. Apache Cassandra nie umożliwia jednak wykonywania złączeń. Możliwe jest manualne scalanie tabel w aplikacji, ale podobnie jak dla relacyjnych baz danych jest to rozwiązanie skrajnie niewydajne. W celu uniknięcia złączeń należy zbudować widok osi czasu w osobnej tabeli. Jej zawartość będzie uzupełniana podczas dodawania nowych wpisów przez użytkowników. Na osi czasu wyświetlane będą dwie informacje - nazwa użytkownika publikującego oraz treść wpisu.

Autor Twissandry sugeruje, aby w tabeli osi czasu umieścić trzy informacje - nazwę użytkownika, a także czas publikacji oraz identyfikator wpisu. Przy tak zdefiniowanym schemacie na etapie budowy widoku dla każdego wpisu osobno pobierana jest jego treść z tabeli `tweets`. Nie jest to kluczowy problem wydajnościowy, jednakże dzięki denormalizacji dodatkowe pobranie może być łatwo wyeliminowane. Wystarczy uwzględnić treść wpisu w tabeli osi czasu. Zapytanie, które tworzy schemat `timeline` zostało zaprezentowane na listingu 2.23. Poglądowy kształt wynikowej struktury danych przedstwia diagram 2.24.

W zapytaniu tworzącym tabelę `timeline` wykorzystano klauzulę sortowania `WITH CLUSTERING ORDER BY`. Pozwala ona na układanie kolejności wierszy względem wartości klucza typu `clustering`. W przykładzie 2.23 sortowanie wykonywane jest względem wartości kolumny `time`, w porządku malejącym[14]. Oznacza to, że wiersze pobrane dla danej nazwy użytkownika będą posortowane w odwrotnym porządku chronologicznym (od najnowszych).

Należy zauważyć, że w Cassandrze możliwe jest sortowanie wierszy tylko i wyłącznie po wartości klucza typu `clustering`. Wynika to z reprezentacji wewnętrznej danych. Klucz typu `partition` jest tłumaczony na identyfikator wiersza w Cassandrze. Wiersze są rozdzielane w architekturze pierścienia, w zależności od wartości skrótu identyfikatora. Zmiana ich sortowania nie jest możliwa, gdyż zaprzecza to pryncypiom architektonicznym Cassandry. Sortowanie klucza typu `clustering` jest możliwe i polega na szeregowaniu

[14]Świadczy o tym słowo kluczowe *DESC* - od angielskiego *descending* - malejąco.

```
CREATE TABLE timeline (
    username text,
    time timeuuid,
    tweet_id uuid,
    body text,
    PRIMARY KEY (username, time)
) WITH CLUSTERING ORDER BY (time DESC)
INSERT INTO timeline (username, time, tweet_id, body) VALUES (
    'jturek',
    915d6810-308c-11e4-95de-f9043d8d556b,
    f8122d00-2f99-11e4-babf-0002a5d5c51b,
    'Sample')
```

Rysunek 2.23: Oś czasu w Twissandrze z modyfikacją wprowadzającą denormalizację treści wpisu.

	time: 915d6810-308c-11e4-95de-f9043d8d556b	
	tweet_id	**body**
'jturek'	f8122d00-2f99-11e4-babf-0002a5d5c51b	'Sample'

Rysunek 2.24: Poglądowy wynik struktury uzyskanej po wykonaniu zapytań 2.23.

kolumn w wewnętrznej reprezentacji danych.

W powyższym schemacie kluczowym problemem jest reprezentacja publicznej osi czasu. Autor Twissandry wykorzystuje do tego celu zdefiniowaną wcześniej tabelę `timeline` używając specjalnego identyfikatora użytkownika (`!PUBLIC!`). Rozwiązanie to nie jest jednak poprawne:

- Cassandra może przechowywać maksymalnie miliard kolumn w jednym wierszu (przy założeniu, że cały węzeł wypełnia pojedynczy wiersz). [21] Oznacza to, że w tabeli `timeline` dla klucza `!PUBLIC!` można umieścić ograniczoną liczbę wpisów. Istnieje realne ryzyko, że liczba ta okaże się zbyt mała.

- Architektura w istotny sposób degraduje dobrą skalowalność horyzontalną, którą cechuje się Cassandra. Wiersz z publiczną osią czasu będzie znacznie dłuższy niż przeciętny wiersz z osią czasu użytkownika, gdyż znajdują się w nim wszystkie wpisy utworzone w serwisie. W zaprezentowanym schemacie publiczna oś czasu będzie umieszczona w jednym fizycznym wierszu Cassandry[15], a więc węzły przechowujące ją będą nieproporcjonalnie obciążone w porównaniu do pozostałych.

Rozwiązaniem tego problemu jest wprowadzenie większej granulacji publicznej osi czasu. Można to osiągnąć rozbudowując klucz typu `partition` o kolejną składową. Sugerowanym przez Autora rozbiciem osi czasu jest podział względem dnia publikacji wpisu. Przykładowo aplikacja może odwoływać się do klucza `!PUBLIC!2014-07-01` dla wpisów opublikowanych pierwszego lipca 2014 roku, natomiast do pobrania danych z kolejnego dnia wykorzystywać identyfikator `!PUBLIC!2014-07-02`.

Dużym ułatwieniem jest fakt, że aplikacja nie musi samodzielnie budować i zarządzać skomplikowanymi identyfikatorami. Język CQL umożliwia wykorzystanie złożonego klucza typu `partition`. [22] Listing 2.25 prezentuje w jaki sposób zdefiniować oś czasu z dodatkowym podziałem względem dnia publikacji. Zmiana obejmuje dodanie kolumny tekstowej `day` oraz uwzględnienie jej w klauzuli `PRIMARY KEY`. W fizycznym modelu danych wpisy o jednakowych wartościach nazwy użytkownika i dnia zostaną pogrupowane w wiersze, natomiast wpisy dla tej samej osoby, ale pochodzące z różnych dni, będą rozdzielone pomiędzy węzłami.

Wykorzystując klucze złożone należy pamiętać, że istotna jest kolejność nawiasów zastosowanych w klauzuli `PRIMARY KEY`. W przypadku pominięcia wewnętrznego nawiasu dla pary `(username, day)` nowa kolumna zostałaby częścią klucza typu `clustering`, który byłby opisany dwójką `(day, time)`.

[15] Należy pamiętać, że wiersze w fizycznym modelu danych są niepodzielne.

```
CREATE TABLE timeline (
    username text,
    day text,
    time timeuuid,
    tweet_id uuid,
    body text,
    PRIMARY KEY ((username, day), time)
) WITH CLUSTERING ORDER BY (time DESC)
INSERT INTO timeline (username, day, time,
        tweet_id, body) VALUES (
    'jturek',
    '2014-08-01',
    915d6810-308c-11e4-95de-f9043d8d556b,
    f8122d00-2f99-11e4-babf-0002a5d5c51b,
    'Sample')
```

Rysunek 2.25: Oś czasu w Twissandrze partycjonowana względem dnia utworzenia wpisu.

2.8 Modelowanie - wzorce i antywzorce

Pomimo że CQL znacząco upraszcza modelowanie w Cassandrze to stworzenie efektywnego schematu danych nie jest zadaniem prostym. Aby ułatwić ten proces twórcy i użytkownicy Cassandry zaczęli opisywać wzorce i antywzorce modelowania oraz dostępu do danych. Pełnią one rolę analogiczną do wzorców i antywzorców projektowych znanych z inżynierii oprogramowania. Na ich opis składa się możliwie ogólna definicja problemu, a także poprawny (lub niepoprawny) sposób jego rozwiązania.

Przykładem antywzorca modelowania jest kolejka, której elementy zapisywane są w kolumnach. [23] Kolejka to struktura danych, w której ilości wykonywanych operacji wstawiania, usuwania i odczytu są do siebie zbliżone. W przypadku Cassandry usuwanie kolumn z wiersza nie jest wykonywa-

ne natychmiast po odebraniu żądania. Zamiast tego usunięta kolumna jest oznaczana specjalnym znacznikiem *tombstone* i fizycznie usuwana dopiero po upłynięciu pewnego czasu. Takie działanie przyspiesza znacząco operację usuwania kosztem operacji odczytu. Kiedy w wierszu występuje wiele znaczników *tombstone* operacje filtrowania zakresu kolumn wykonywane są znacznie wolniej.

Przykładem wzorca dostępu do danych jest unikanie konfliktów synchronizacji poprzez uaktualnianie wyłącznie zmodyfikowanych wartości. [24] Encja *Item* z diagramu 2.8 ma 5 właściwości. Wyświetlenie ekranu aktualizacji przedmiotu wymaga pobrania zawartości całego wiersza z bazy danych. Wzorzec stanowi, że jeżeli na tym ekranie zostanie zmieniony wyłącznie opis to do Cassandry należy przesłać żądanie uaktualniające zawartość wyłącznie jednej komórki - *desc*. Pozostałe wartości z formularza mogą być już nieaktualne. Przesłanie kompletu informacji poskutkowałoby nadpisaniem aktualnych wartości. Postępowanie według wzorca pozwala wykorzystywać mechanizm rozwiązywania konfliktów wbudowany w Cassandrę.

Rozdział 3

Mapowanie obiektowo-relacyjne

Mapowanie obiektowo-relacyjne (zwyczajowo określane skrótem ORM od angielskiego terminu *object-relational mapping*) to technologia, która pozwala automatyzować połączenie pomiędzy relacyjnym modelem baz danych, a paradygmatem programowania zorientowanego obiektowo. [25] W szerokiej perspektywie mapowanie obiektowo-relacyjne może być postrzegane jako próba przełożenia tabelarycznej reprezentacji danych umieszczonej w pamięci masowej na dowolną reprezentację danych w pamięci operacyjnej. [26]

Potrzeba stworzenia mechanizmów ORM to naturalna konsekwencja powstawania aplikacji zorientowanych na przetwarzanie dużej ilości współdzielonych danych. Obsługa dostępu do bazy danych, czyli nieodłączna część współczesnych aplikacji biznesowych, wymaga napisania dużych ilości kodu źródłowego. Kod ten jest powtarzalny pomiędzy poszczególnymi rodzajami danych i aplikacjami. Jego stworzenie jest pracochłonne, a nie stanowi żadnej wartości funkcjonalnej dla aplikacji. Mechanizmy ORM w sposób znaczący redukują ilość kodu niezbędnego do komunikacji z systemami bazodanowymi. Ich główne zalety to:

- Możliwość definiowania modelu danych poprzez tworzenie klas reprezentujących obiekty wykorzystywane w aplikacji.
- Implementacja typowych operacji na danych: tworzenia, aktualizacji, usuwania i wyszukiwania.

- Eliminacja konieczności lub uproszczenie zarządzania połączeniami, sesjami i transakcjami bazodanowymi.
- Zwiększenie bezpieczeństwa aplikacji. Mechanizmy ORM dostarczają narzędzia do ochrony przed atakami typu *SQL Injection*[1]. [27]
- Uniwersalne wsparcie dla wielu silników bazodanowych, które mogą różnić się od siebie implementowanymi standardami języka SQL.
- Umożliwienie wymiany środowiska bazodanowego bez konieczności modyfikacji kodu aplikacji.

O powszechnym użyciu mechanizmów ORM w projektach informatycznych świadczą statystyki:

- W Internecie dostępne są tysiące implementacji mechanizmów ORM o otwartych źródłach dla dziesiątek różnych języków programowania. Wyszukiwanie frazy *object-relational mapping* w witrynie `http://github.com` zwraca około 45 tysięcy wyników.
- Popularnym narzędziem ORM jest biblioteka Hibernate dla języka Java. Wyszukiwanie frazy *<artifactId >hibernate-core </artifactId >*[2] w witrynie `http://github.com` zwraca około 151 tysięcy wyników.

O popularności mechanizmów ORM świadczy ponadto fakt powstawania oficjalnych standardów mapowania obiektowo-relacyjnego włączanych do specyfikacji języków programowania. Przykładem takiego standardu jest JPA[3] dla języka Java.

Mapowanie obiektowo-relacyjne nie jest pozbawione wad. Opanowanie podstaw mechanizmów ORM jest często trudniejsze niż nauka języka SQL.

[1]SQL Injection (ang. wstrzykiwanie SQL) - typ ataku polegający na wykorzystywaniu specjalnie spreparowanych wartości, które dołączone do szablonu zapytania SQL powodują szkodliwe działanie niezgodne z intencją programisty.

[2]Jest to fraza będąca częścią deklaracji biblioteki Hibernate w popularnym narzędziu do budowania projektów w języku Java - Maven.

[3]Java Persistence API - szerszy opis znajduje się w sekcji 3.1.

Wynika to ze złożoności takich mechanizmów. Przykładowo na projekt Hibernate ORM przypada ponad milion linii kodu źródłowego. Inną często przytaczaną wadą mapowania obiektowo-relacyjnego jest drastyczny spadek wydajności wynikający z natury mechanizmów ORM. Prowadzi to do sytuacji, w których organizacje decydują się na stworzenie własnego, dedykowanego dla danego problemu oprogramowania. Ilość projektów, które wykorzystują systemy mapowania obiektowo-relacyjnego pozwala jednak twierdzić, że wady te są akceptowalne w obliczu licznych zalet ORM.

3.1 Java Persistence API

Java Persistence API to interfejs programistyczny, który ma za zadanie uprościć tworzenie, zarządzanie i zapisywanie obiektów, które reprezentują dane z baz relacyjnych. [28] Interfejs JPA operuje na strukturach POJO[4], które dekorowane są adnotacjami umożliwiającymi konfigurację reguł mapowania. Przykład takiego obiektu prezentuje listing 3.1. Adnotacja `@Entity` określa klasę, która reprezentuje encję danych. Parametr `@Id` dekoruje identyfikator encji, natomiast `@GeneratedValue` oznacza, że będzie on ustawiany automatycznie. Adnotacje `@Column` informują, że dane zmienne będą mapowane na kolumny bazodanowe. Parametr `@ManyToMany` definiuje relację wiele-do-wielu.

Definicja encji 3.1 nie specyfikuje kompletnego przykładu. Brakuje przede wszystkim wartości konfiguracyjnych, takich jak nazwa tabeli, z której encja będzie mapowana, nazwy kolumn czy też sposób reprezentacji relacji wiele-do-wielu. W rzeczywistości JPA nie jest tak czytelene jak sugeruje przedstawiony przykład. Parametry konfiguracyjne zajmują większą część definicji klasy i zdarza się, że najważniejsza informacja - specyfikacja samego obiektu, który reprezentuje dane - jest mocno przysłonięta.

[4]Plain Old Java Object (ang. dosłownie „prosty, stary obiekt Java") - termin określający zwykły obiekt języka Java, używany jako przeciwieństwo Enterprise Java Bean, czyli specjalnego obiektu, który stosował się do wielu ściśle określonych reguł.

```
@Entity
public class User {
    @Id
    @GeneratedValue
    private Integer id;
    @Column
    private String name;
    @Column
    private String surname;
    @Column
    private String password;
    @ManyToMany
    private List<Item> wishlistItems;
    // getters & setters
}
```

Rysunek 3.1: Przykładowy obiekt specyfikujący użytkownika w standardzie JPA.

Na listingu 3.2 przedstawiono przykładowy kod, który służy do zapisywania użytkownika w bazie danych. Do obsługi operacji zapisu wykorzystywany jest zarządca encji (`EntityManager`). Za jego pomocą rozpoczynana jest transakcja (`beginTransaction()`), wywoływane jest żądanie utrwalenia obiektu (`persist()`), a następnie transakcja jest wykonywana (`commit()`).

```
User user = new User();
// setting field values
EntityManager manager = managerFactory.createEntityManager();
  manager.getTransaction().beginTransaction();
    manager.persist(user);
  manager.getTransaction().commit();
manager.close();
```

Rysunek 3.2: Przykładowy kod zapisujący użytkownika w bazie danych w standardzie JPA.

Do najbardziej popularnych implementacji Java Persistence API należą Hibernate ORM, OpenJPA oraz Toplink. Dużą przeszkodą jest fakt, że inter-

fejs JPA nie jest wyczerpująco określony. Opisuje on tylko najbardziej podstawowe operacje na obiektach bazodanowych. Z jednej strony pozostawia to twórcom bibliotek dużą elastyczność w kwestii implementacji. Z drugiej sprawia, że w przypadku wykonywania operacji bardziej skomplikowanych niż zapisanie i odczytanie mapowanego obiektu, programista często jest zmuszony sięgać do elementów specyficznych dla danej implementacji JPA. Sprawia to, że większość aplikacji wykorzystujących wymienione biblioteki jest nieprzenośna pomiędzy mechanizmami.

3.2 Kundera

Wraz z pojawieniem się języka CQL pojawiła się szansa wykorzystania istniejących implementacji mechanizmów ORM do przechowywania danych z użyciem silnika Cassandry. Motywacją do stworzenia takiego rozwiązania jest zwiększenie wydajności istniejących aplikacji bez modyfikacji ich kodu. Minimalna rekonfiguracja aplikacji i podłączenie jej pod klaster bazodanowy skutkowałyby zwiększeniem wydajności działania aplikacji. Ponadto wykorzystanie istniejących interfejsów mogłoby umożliwić obsługę różnych typów baz danych należących do ruchu NoSQL.

Przykładem projektu, który wykorzystuje mechanizm ORM do obsługi NoSQLowych baz danych jest Kundera.[29] Kundera zapewnia implementację mapowania obiektowego zgodną ze standardem JPA 2.0 między innymi dla Cassandry, HBase, MongoDB oraz Neo4j. Dodatkowo biblioteka wspiera obsługę wielu mechanizmów równocześnie.

3.2.1 Wydajność

Wyniki pomiarów na stronie domowej projektu Kundera świadczą o tym, że biblioteka nie wprowadza znacznych narzutów wydajnościowych względem bezpośredniego wykorzystania interfejsu bazy danych. Ważniejsze jest jednak sprawdzenie jak duży narzut wprowadza dostosowanie relacyjnego modelu danych do Apache Cassandry. W tym celu Autor przeprowadził te-

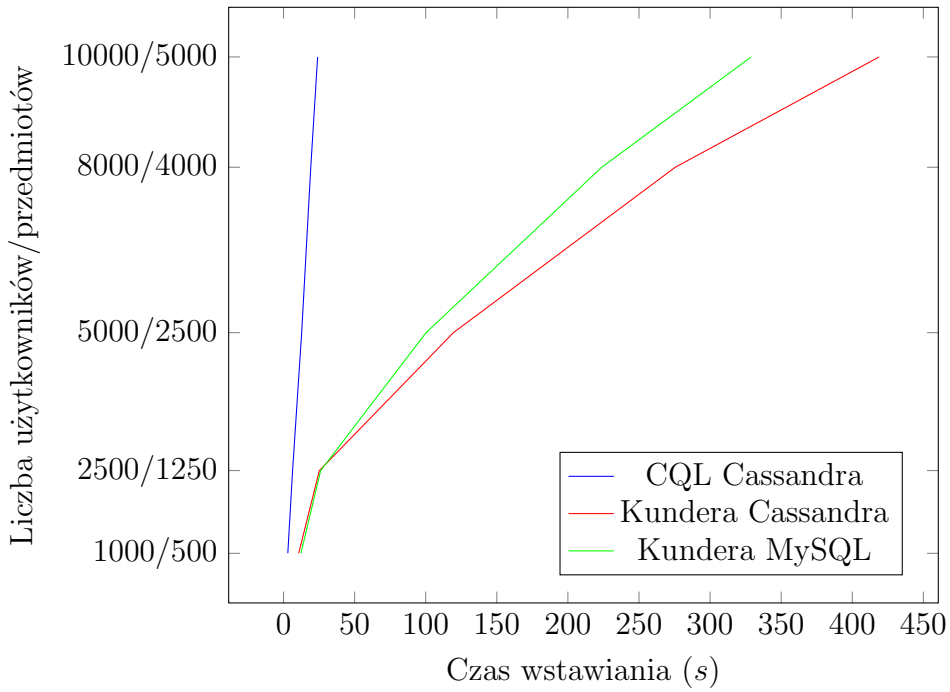
sty porównawcze masowego wstawiania i pobierania obiektów. Dla każdej z dwóch operacji zostały przeprowadzone trzy testy. Pierwszy test mierzy czas referencyjny. Jest wykonywany dla zdenormalizowanego modelu Cassandry przedstawionego na diagramie 2.10. Drugi test sprawdza czasy dla znormalizowanego modelu zaprezentowanego na diagramie 2.6, opisanego w JPA i uruchomiony na Cassandrze z wykorzystaniem biblioteki Kundera. Trzeci test wykorzystuje identyczny opis modelu jak drugi, jednakże wykonywany jest dla silnika MySQL. Wszystkie testy przeprowadzane były jednowątkowo. Czas opóźnień przesyłania danych jest pomijalny, gdyż testy były prowadzone na środowisku lokalnym.

W trakcie testów mierzony był całkowity czas wykonania danej operacji. Przyjęte zostały następujące założenia:

- Liczba użytkowników i przedmiotów są parametryzowane i skalowane liniowo.
- Liczba przedmiotów, które użytkownik może mieć na swojej liście życzeń zawiera się w przedziale $[0; 10]$.
- Odczytywana jest parametryzowalna liczba użytkowników.

Wyniki czasu wstawiania wielu rekordów zostały przedstawione na wykresie 3.3. Czasy pobierania wielu rekordów zostały zebrane na wykresie 3.4.

Wyniki zebrane na wykresie 3.3 pokazują, że wykorzystanie mechanizmów mapowania obiektowo-relacyjnego dla bazy danych Cassandra jest bardzo złym wyborem. Narzut związany z konwersją modelu danych do postaci relacyjnej jest tak duży, że w praktyce pojedynczy węzeł Cassandry osiąga gorsze wyniki zapisu niż baza danych MySQL. Zmiana silnika bazodanowego dla istniejącego kodu nie tylko nie poprawi wyników wydajnościowych, ale może wręcz spowodować spowolnienie działania aplikacji. Jedynym zyskiem z takiego rozwiązania będzie możliwość wykorzystania natywnego mechanizmu klastrowania węzłów Cassandry. Dzięki temu wyeliminowany zostanie



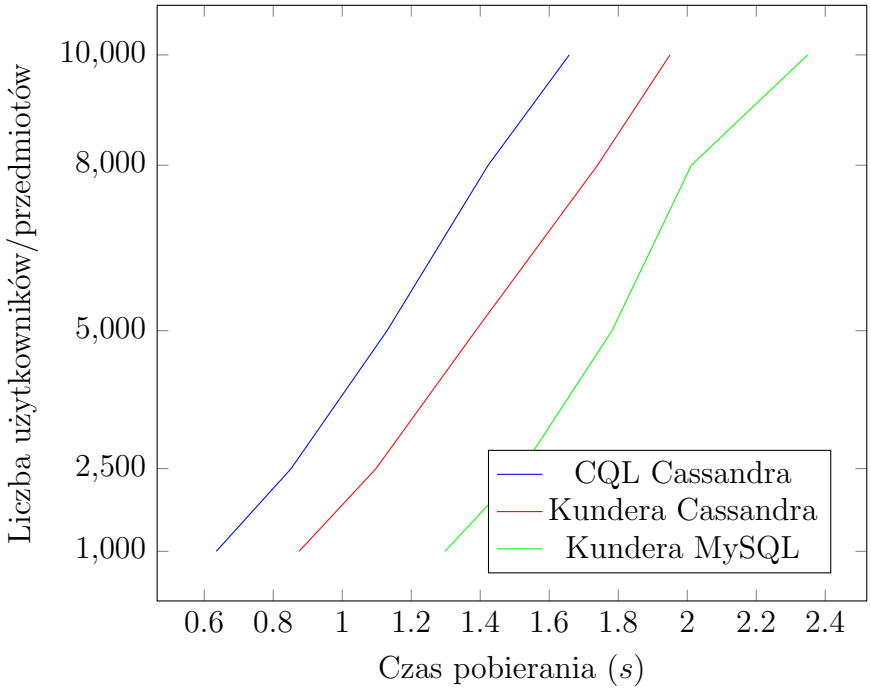
Rysunek 3.3: Porównanie czasu wstawiania wielu rekordów.

pojedynczy punkt awarii systemu. Ten sam efekt można jednak uzyskać stosując rozwiązania dedykowane dla relacyjnych baz danych. Przykładami takich systemów mogą być MySQL Cluster CGE dla bazy danych MySQL oraz Oracle RAC dla baz Oracle Database Enterprise Edition.

Potencjalnym zastosowaniem mapowania obiektowo-relacyjnego dla bazy danych Cassandra są środowiska integracyjne dla wielu aplikacji. Wykorzystując homogeniczny model danych można odwoływać się do różnych silników bazodanowych. W bibliotece Kundera istnieje takie rozwiązane. Zostało nazwane Polyglot Persistence[5]. W praktyce dostosowanie modelu danych ORM do istniejących schematów baz danych jest problematyczne.

Wykres 3.3 demonstruje jak ogromną przewagę szybkości przy zapisie posiada poprawnie zaprojektowany model danych Apache Cassandra. Przy-

[5]Polyglot Persistence (dosłowne tłumaczenie to „zapis poliglotyczny") - opis mechanizmu znajduje się na stronie `https://github.com/impetus-opensource/Kundera/wiki/Polyglot-Persistence`.



Rysunek 3.4: Porównanie czasu pobierania wielu rekordów.

puszczalnie osiągnięty czas mógłby być jeszcze lepszy, gdyż ograniczenie wydajności zapisu nastąpowało prawdopodobnie po stronie klienta testowego. Zgodnie ze specyfikacją Apache Cassandra jest w stanie obsługiwać bez opóźnień znacznie więcej jednoczesnych żądań zapisu.

Czasy pobierania listy życzeń użytkownika przedstawione na wykresie 3.4 nie różnią się znacząco od siebie. Przewaga szybkości pobierania danych z wykorzystaniem CQL ponownie wynika z zastosowania lepszego modelu danych. Denormalizacja pól encji *Item* pozwala pominąć pobieranie dla każdego użytkownika dodatkowego wiersza z bazy danych.

3.3 Wnioski z badania wydajności

Na podstawie wyników przeprowadzonych badań można wnioskować, że mechanizmy mapowania obiektowo-relacyjnego nie nadają się do wykorzystania przez bazę danych Apache Cassandra. Wynika to z faktu, że mechanizmy te

wspomagają normalizację i zachowywanie relacji między encjami, natomiast efektywne modelowanie dziedziny danych w Cassandrze dąży do denormalizacji i maksymalnego uproszczenia zależności pomiędzy obiektami.

Drastyczna różnica szybkości zapisu danych w modelach zoptymalizowanym i niezoptymalizowanym świadczy o tym, że do efektywnego modelowania dziedziny w Apache Cassandra niezbędne są: wiedza na temat sposobu przechowywania danych przez tę bazę oraz podejmowanie świadomych wyborów. Dostarczone przez ORMy mechanizmy mogą wyłącznie spowolnić korzystanie z Cassandry.

Podczas badania wydajności Autor natrafił również na problemy z przenośnością kodu źródłowego pomiędzy różnymi systemami bazodanowymi. Wykonanie operacji wstawiania partii danych w bazie relacyjnej oraz Cassandrze wymagało modyfikacji kodu źródłowego aplikacji testowej. W relacyjnej bazie danych nie można było ukończyć testu wstawiania i pobierania danych wykorzystując jedynie elementy zdefiniowane w JPA. Pomimo wielokrotnej zmiany konfiguracji w pewnym momencie testu dochodziło do całkowitego zablokowania puli połączeń. Dopiero przejście do transakcji bezstanowej zdefiniowanej w Hibernate ORM[6] umożliwiło zakończenie testu.

Opisane problemy z przenośnością kodu mogą być następstwem fundamentalnych różnic pomiędzy relacyjnymi systemami bazodanowymi a Apache Cassandrą. Przykładowo transakcja, czyli jedno z podstawowych pojęć związanych z zapytaniami w języku SQL, nie istnieje w systemie Cassandra. Dostosowanie implementacji do interfejsu mapowania obiektowo-relacyjnego wymaga sztucznego blokowania zapytań po stronie klienta, co prowadzi do wystąpienia niepotrzebnych opóźnień i nadmiernego wykorzystania mechanizmów synchronizacji wielowątkowej. Z drugiej strony brak lub nieodpowiednie wykorzystanie transakcji w przypadku bazy relacyjnej prowadzi do błędów zapytań.

[6]Hibernate ORM - środowisko aplikacyjne implementujące standard JPA dla baz relacyjnych. Zdefiniowana w ramach tego środowiska transakcja bezstanowa nie jest częścią interfejsu JPA, a więc nie może być wspierana przez Kunderę.

Wyniki badań nie świadczą o niemożliwości implementacji mapowania obiektowego dla Cassandry. Dla optymalnej wydajności należy jednak zastosować rozwiązane dedykowane:

- Definiowanie modelu powinno być skoncentrowane wokół wewnętrznej reprezentacji danych w Cassandrze. W stosunku do mapowania obiektowo-relacyjnego kluczowe są aspekty obniżenia istotności relacji i wsparcie dla operacji denormalizacji z poziomu interfejsu.

- Mapowanie powinno w prosty i transparentny sposób udostępniać realizację poprawnych wzorców modelowania. Przykładowo lista wartości powinna móc być zamodelowana przynajmniej na dwa sposoby: jako kolumna typu list lub jako lista kolumn o nazwach zawierających wartości listy. Mapowanie obiektowe dla Cassandry powinno umożliwiać wymienienie fizycznej struktury danych bez zmiany wykorzystującego ją kodu źródłowego.

- Mapowanie obiektowe dla Cassandry powinno być proste. Język CQL, w porównaniu do SQL, ma prostą składnię i nie udostępnia wielu operacji. Ponadto skomplikowany mechanizm mapowania mógłby niekorzystnie kontrastować z wydajną bazą danych. Przykładem nieczytelnego mapowania z dużą liczbą opcji konfiguracyjnych może być pole encji opisane w JPA przedstawione na listingu 3.5.

3.4 Hibernate OGM

Alternatywnym rozwiązaniem, które dostarcza implementacji Java Persistence API dla baz NoSQL jest Hibernate Object/Grid Mapper (OGM). Hibernate OGM wykorzystuje silnik projektu ORM o tej samej nazwie. Niestety, aktualne wydanie (wersja **4.1.0.Beta5**) wspiera wyłącznie silniki Infispan, Ehcache, MongoDB oraz Neo4j.

```
@ManyToMany
@JoinTable(name = "wishlist",
           joinColumns = {
               @JoinColumn(name = "userId",
                           referencedColumnName = "userId") },
           inverseJoinColumns = {
             { @JoinColumn(name = "itemId",
                           referencedColumnName = "itemId") },
           foreignKey = @ForeignKey(name = "userId_fk"),
           inverseForeignKey = @ForeignKey(name = "itemId_fk"))
private Set<Item> wishlistItems = new HashSet<Item>();
```

Rysunek 3.5: Przykład nieczytelnego mapowania obiektowo-relacyjnego.

Rozwiązanie to jest jednak interesujące z punktu widzenia dalszego rozwoju pracy. Według mapy przyszłych wydań Hibernate OGM w wersji 4.2 wprowadzi wsparcie dla silnika bazy danych Apache Cassandra. [30]

3.5 Koncepcja mapowania dla Cassandry

W większości rozwiązań ORM encja modelu danych jest opisywana jako definicja klasy, której obiekty reprezentować będą instancję tej encji. Rozważmy listing 3.6, który przedstawia hipotetyczne dostosowanie standardu JPA do denormalizacji dla modelu 2.10.

```
@Table(name = "wishlist")
class Wishlist {
    @Id(type = IdType.PARTITION_KEY)
    private String userId;
    @Denormalize(clustering_keys = { "itemId" },
                 fields = { "name", "price" })
    private Item item;
}
```

Rysunek 3.6: Hipotetyczny przykład dostosowania definicji JPA do denormalizacji.

W powyższym przykładzie denormalizacja jest modelowana jako zdegenerowany przypadek relacji jeden-do-jednego. Przypisanie zmiennej *item* do obiektu klasy *Wishlist* i utrwalenie tego obiektu w bazie danych spowoduje uzupełnienie wartości kolumn *itemId*, *name* oraz *price* w tabeli *wishlist*. Po pobraniu obiektu klasy *Wishlist* z bazy danych przechowywany reprezentant klasy *Item* posiada tylko część informacji. Takie rozwiązanie posiada istotną wadę. Podczas korzystania z mechanizmu nie można jednoznacznie stwierdzić czy brak uzupełnienia pola *desc* w obiekcie klasy *Item* jest spowodowany tym, że opisu faktycznie brakuje, czy też operacje dokonywane są na niekompletnym (zdenormalizowanym) obiekcie. Rozwiązaniem pozbawionym tej wady jest denormalizacja jawna przedstawiona na listingu 3.7.

```
@Table(name = "wishlist")
class Wishlist {
    @Id(type = IdType.PARTITION_KEY)
    private String userId;
    @Id(type = IdType.CLUSTERING_KEY)
    private String itemId;
    @Column(name = "name")
    private String name;
    @Column(name = "price")
    private String price;
}
```

Rysunek 3.7: Hipotetyczny przykład zastosowania denormalizacji jawnej w JPA.

Denormalizacja jawna posiada jednak inne wady. Przede wszystkim model nie podaje, że kolumny *itemId*, *name* oraz *price* powinny pochodzić od instancji klasy *Item*. Ponadto użytkownik mechanizmu musi pamiętać o ręcznym wypełnieniu pól, co jest z kolei narażone na błędy. Eleganckie połączenie tych dwóch opcji nie jest możliwe w języku Java.

Ze względu na mechanizm metaklas w Pythonie stworzono eleganckie i proste implementacje ORM dla relacyjnych baz danych. Dwa najpopular-

niejsze mechanizmy to Django ORM[7] oraz SQLAlchemy[8]. Wykorzystując metaklasy osiągalne jest stworzenie mapowania o interfejsie wymienionych mechanizmów, który rozwiązuje problem zasygnalizowany w przypadku języka Java. Ilustruje to listing 3.8.

```
class Wishlist(Model):
    userId = TextField(partition_key=True)
    item = DenormalizedField(relates=Item,
                             clustering_keys=['itemId'],
                             fields=['name', 'price'])
w = Wishlist()
w.item = Item(id='1', name='n', price='10.0')
id = w.item_itemId    # id equals '1'
name = w.item_name    # name equals 'n'
price = w.item_price  # price equals '10.0'
i = w.item            # exception raised
```

Rysunek 3.8: Hipotetyczny przykład denormalizacji w mapowaniu w języku Python.

Dzięki metaklasom można dokonać następujących modyfikacji modelu *Wishlist*:

- Dla każdej instancji *DenormalizedField* automatycznie dodać do klasy pola, które będą przechowywać zdenormalizowane wartości. W przykładzie definicja klasy została rozszerzona o pola *item_itemId*, *item_name*, *item_price*.
- Każdą instancję *DenormalizedField* zamienić na właściwość tylko do zapisu, która automatycznie ustawia wartości zdenormalizowanych pól.

Dzięki takim modyfikacjom możliwe jest ustawianie wartości zdenormalizowanych pól poprzez podawanie całej instancji obiektu *Item*, jak w listin-

[7]Django ORM - część platformy Django, która odpowiada za operacje bazodanowe. Dokumentacja jest dostępna pod adresem `https://docs.djangoproject.com/en/dev/topics/db/`.

[8]Strona domowa projektu jest dostępna pod adresem `http://www.sqlalchemy.org`.

gu 3.6. Jednocześnie pobieranie wartości z obiektu możliwe jest tylko poprzez odwołania do zdenormalizowanych pól. Nie da się pomylić ze sobą zdenormalizowanych i pełnych instancji klasy *Item*.

Rozdział 4

Modelowanie obiektowe dla Cassandry

Na podstawie wyników wydajnościowych przedstawionych w sekcji 3.3 Autor zaproponował odmienną koncepcję mechanizmu mapowania obiektowego dla bazy danych Cassandra. Aby uwypuklić różnice pomiędzy przedstawioną propozycją i tradycyjnymi mechanizmami ORM Autor postuluje stosowanie odmiennego nazewnictwa. Omówiona w sekcji 3.5 koncepcja, która będzie rozwijana w dalszej części pracy, nazywana będzie **modelowaniem obiektowym**:

- Mechanizmy mapowania obiektowo-relacyjnego nie wymuszają na użytkowniku kolejności projektowania komponentów. Osiągalne są zarówno podejście *code first*[1], jak i *database first*[2]. W proponowanym przez Autora modelowaniu obiektowym, ze względu na występowanie struktur wysokiego poziomu odpowiadających wzorcom projektowym, podejście *database first* jest możliwe tylko teoretycznie lub w bardzo ograniczo-

[1]Code first (ang. dosłownie „najpierw kod źródłowy") - w kontekście ORM oznacza to modelowanie dziedziny za pomocą kodu źródłowego, z którego następnie generowany jest schemat bazy danych.

[2]Database first (ang. dosłownie „najpierw baza danych") - w kontekście ORM oznacza to modelowanie dziedziny jako schemat bazodanowy, do którego następnie pisany jest mapujący kod źródłowy.

nym zakresie. Znacznie bardziej efektywne jest podejście *code first*. Posiada ono wsparcie implementacyjne - generację schematu - i jest rekomendowane przez Autora.

- W mechanizmach mapowania obiektowo-relacyjnego mapowanie odpowiada najczęściej typom danych, natomiast w obrębie danego typu jest ono jednoznaczne. Przykładowo identyfikator konwertowany jest na liczbowy klucz główny, lista przechowywana jest jako relacja, a ciąg znaków przekształcany jest na typ *VARCHAR*. W przypadku modelowania obiektowego dla Cassandry wybór wyznaczać będzie natomiast sposób przechowywania danej wartości. W zależności od narzuconego sposobu modelowania listy będzie mogła zostać ona zrzutowana na wartość kolumny o typie `list`, nazwy kolumn w wierszu lub osobną tabelę.

- Mapowanie obiektowo-relacyjne dostarcza przede wszystkim niskopoziomowe odpowiedniki typów bazodanowych. Wyjątkiem jest relacja wiele-do-wielu. Modelowanie obiektowe dla Cassandry skupia się zarówno na polach niskiego poziomu (przykładowo kolumna o wartości tekstowej), jak i na reużywalności komponentów wysokopoziomowych odpowiadających wzorcom projektowym.

- Częścią modelowania obiektowego dla Cassandry są opcjonalne mechanizmy mapowania obiektowo-relacyjnego. Należą do nich generacja schematu oraz migracje pomiędzy wersjami modelu.

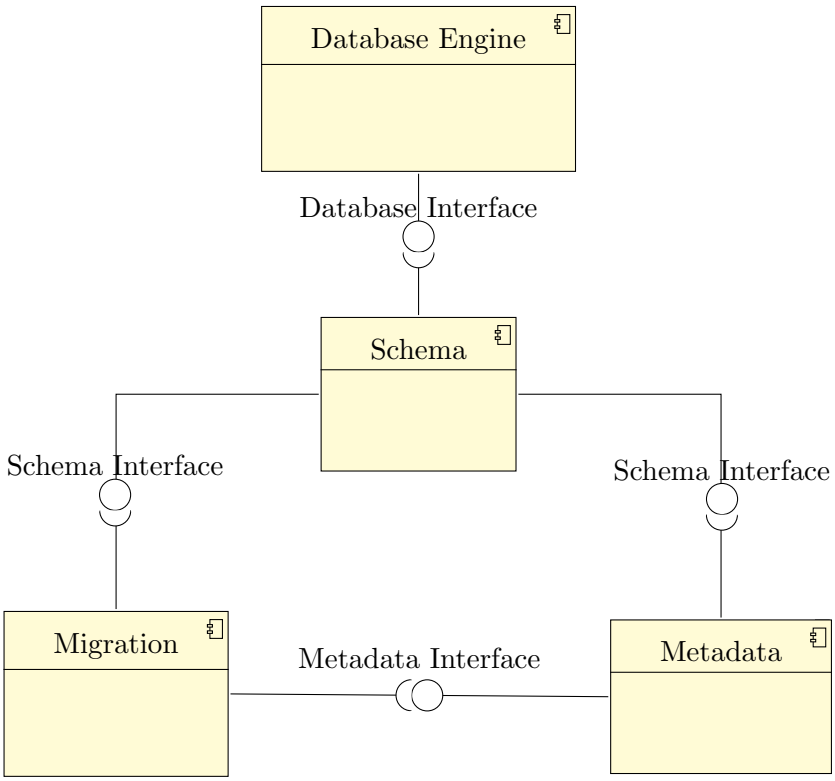
W dalszej części pracy omówiona zostanie konkretna implementacja modelowania obiektowego dostarczona wykonana przez Autora. Pełna nazwa tego projektu to **Object Modeling for Cassandra**. W dalszej części pracy używana będzie nazwa skrócona - *OMC*.

4.1 Object Modeling for Cassandra

OMC jest biblioteką dla języka Python, która dostarcza narzędzia do modelowania dziedziny danych dedykowane dla bazy Apache Cassandra. Pakiet udostępnia moduły, które umożliwiają zarządzanie środowiskiem bazodanowym i pracę z danymi bez odwoływania się do interfejsu Thrift/CQL. Funkcjonalności pakietu obejmują:

- Mapowanie obiektowe dla modelu danych Cassandry:
  - Pobieranie, wyszukiwanie i usuwanie danych za pomocą metod interfejsu.
  - Automatyczne pakowanie i rozpakowywanie wartości do/z postaci obiektów.
- Wsparcie dla specyficznych mechanizmów Cassandry:
  - Indeksy drugiego poziomu.
  - Liczniki.
  - Operacje na partiach danych.
- Wspomaganie modelowania zależności między obiektami z wykorzystaniem denormalizacji.
- Wspomaganie tworzenia modelu danych poprzez dostarczenie/wsparcie reużywalnych struktur wysokiego poziomu:
  - szeregów zdarzeń (ang. *time series*),
  - indeksów wartości unikalnych,
  - kolejek.
- Tworzenie i walidacja zgodności modelu danych z wymaganiami mapowania.
- Automatyczna migracja danych pomiędzy różnymi wersjami modelu.

- Narzędzia do populacji danych testowych i profilowania aplikacji.



Rysunek 4.1: Diagram komponentów mechanizmu OMC.

Na rysuku 4.1 zaprezentowano diagram komponentów mechanizmu. Składa się on z czterech podstawowych części:

Silnik bazodanowy jest oparty o sterownik języka CQL. W skład silnika wchodzą metody umożliwiające nawiązywanie i konfigurowanie połączenia z Cassandrą, zarządzanie klastrem, wykonywanie oraz zapewnianie bezpieczeństwa zapytań.

Schemat moduł, który służy do zarządzania fizycznym modelem danych. Odpowiada za tworzenie/utrzymywanie przestrzeni kluczy (`keyspace`), wierszy oraz kolumn. Ponadto pozwala aktualizować fizyczny model danych, a także wyznaczać różnice pomiędzy dwiema wersjami struktury.

Metadane moduł, który zarządza elementami interfejsu służącymi do opisu logicznego modelu danych (używanego wewnątrz mechanizmu). Jego odpowiedzialnością jest konwersja pomiędzy logiczną a fizyczną strukturą. Moduł dostarcza także implementacji metod umożliwiających operacje przetwarzania danych.

Migracja moduł, który potrafi wyznaczać i wersjonować różnice pomiędzy kolejnymi zmianami w modelu danych. Dostarcza również funkcjonalności do aktualizacji fizycznej struktury na żądanie użytkownika.

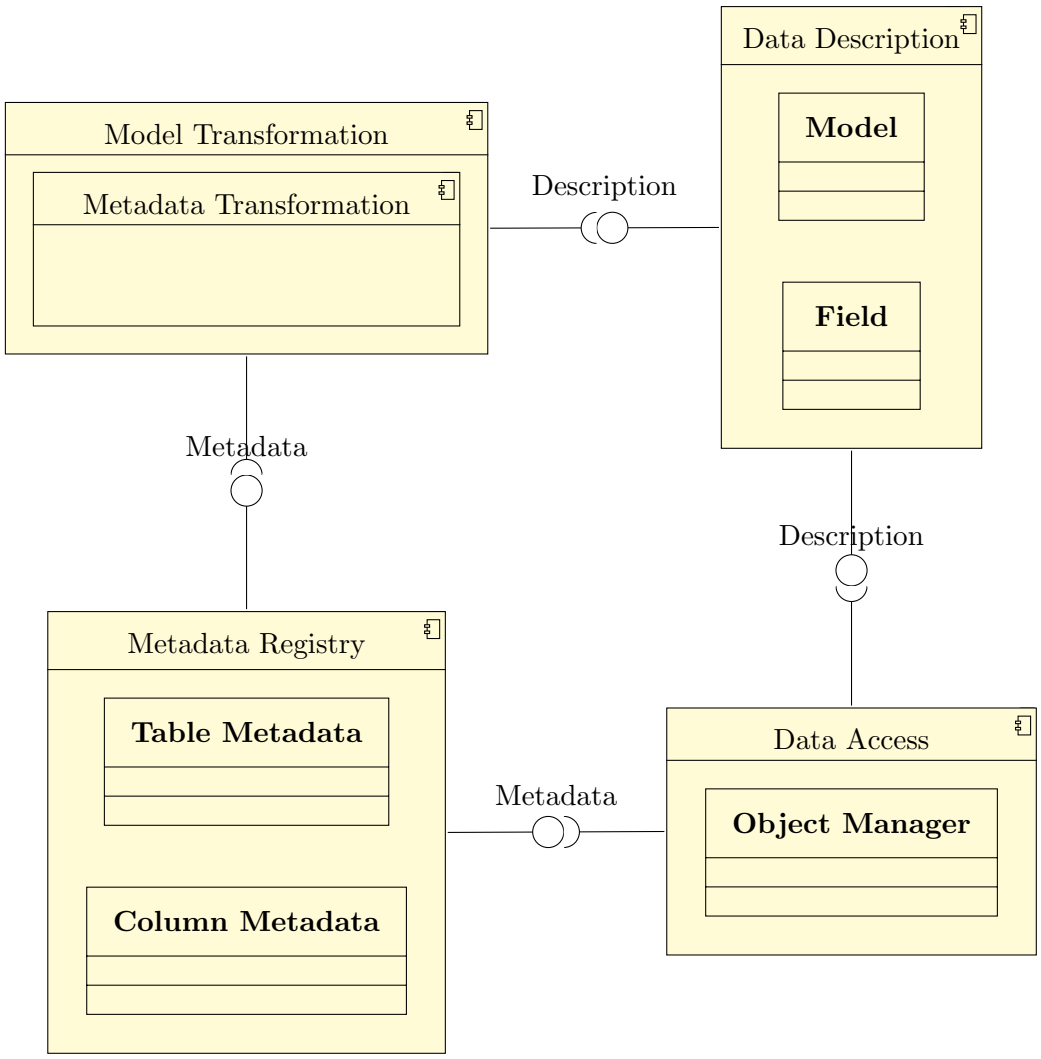
Spośród wymienionych powyżej komponentów najbardziej rozbudowany służy do opisu modelu logicznego za pomocą metadanych. Składają się na niego moduły zaprezentowane na diagramie 4.2. Mają one następującą odpowiedzialność:

Data Description dostarcza komponenty programistyczne do opisu logicznego modelu danych. Definiuje jednostkę tego opisu - `Model`. Ponadto zawiera interfejs pola (`Field`), a także dostarcza implementacji wielu jego różnych typów.

Metadata Registry odpowiada za rejestrowanie wszystkich modeli zdefiniowanych przez użytkownika oraz ich metadanych. Ponadto definiuje byty `TableMetadata` oraz `ColumnMetadata`. Pozwalają one wymieniać informacje pomiędzy warstwami obsługi logicznego oraz fizycznego modelu danych.

Model Transformation implementuje operacje inicjalnej transformacji klas opisujących logiczny model danych. Ponadto udostępnia metody do przekształcania ich do postaci metadanych.

Data Access dostarcza komponenty programistyczne do wykonywania operacji na logicznym modelu danych. Interfejs `ObjectManager` wraz z jego domyślną implementacją udostępniają operacje filtrowania, aktualizacji oraz usuwania danych.



Rysunek 4.2: Diagram komponentów modułu metadanych z wyszczególowionymi najistotniejszymi elementami.

4.2 Porównanie OMC z ORM

W tabeli 4.3 przedstawiono porównanie mechanizmu modelowania obiektowego dla Cassandry z typowymi systemami ORM[3]. Podstawową różnicą jest warstwa abstrakcji, na której operują te rozwiązania. Mechanizmy ORM są

[3]Opracowanie zostało przygotowane na podstawie cech mechanizmów Hibernate oraz SQLAlchemy.

	ORM	**OMC**
Mapowanie obiektów do schematu bazy danych	Niskiego poziomu. Modele przekładają się bezpośrednio na tabele, a pola na kolumny.	Wysokiego poziomu. Modele przekładają się na tabele, a pola mogą przekładać się na tabele, kolumny lub grupy kolumn.
Modelowanie zależności między danymi	Zaawansowane opcje konfiguracyjne modelowania relacji.	Proste, wydajne modelowanie zależności ściśle związane z fizyczną strukturą danych.
Obsługa dziedziczenia	Rozbudowane opcji konfiguracji dziedziczenia obiektów.	Proste dziedziczenie polegające na niejawnym włączeniu wszystkich pól do nadklasy.
Denormalizacja	Obsługiwana manualnie przez użytkownika.	W pełni automatycznie wspierana przez mechanizm (zapis i uaktualnianie na żądanie).
Wsparcie dla wzorców projektowania modelu	Brak.	Zaimplementowane wiele wzorców projektowych gotowych do użycia.
Obsługa mechanizmu migracji	Brak lub realizacja migracji jako oddzielny moduł.	Wbudowany mechanizm migracji.
Testowanie modelu	Ręczna populacja danych. Wbudowane mechanizmy mierzenia czasu zapytań.	Automatyczna populacja danych. Śledzenie zapytań po różnych parametrach.

Rysunek 4.3: Tabelaryczne porównanie funkcjonalności typowych mechanizmów ORM i modułu Object Modeling for Cassandra.

silnie powiązane ze strukturami bazodanowymi. Definiowane przez programistę klasy zawsze odpowiadają tabelom, natomiast pola mapowane są w stosunku jeden-do-jednego na kolumny. Bardzo rzadko występuje sytuacja, gdy mapowania realizowane są niejawnie[4]. W zależności od intencji użytkownika, mechanizm modelowania obiektowego dla Cassandry umożliwia wierne odwzorowanie zachowania systemów ORM. Dostarcza on implementację pól dla wszystkich typów danych, które mogą być umieszczone w kolumnach. Wykorzystując je możliwe jest stworzenie modelu danych niskiego poziomu.

Unikatową cechą modelowania obiektowego dla Cassandry jest możliwość wykorzystywania struktur wyższego poziomu. Zapewniają one abstrakcję od fizycznego schematu i/lub wykonywanych do bazy danych zapytań, która jest skoncentrowana na charakterystyce przechowywanych danych:

- Modelując zależności przy pomocy pola `DenormalizedField` użytkownik wskazuje wyłącznie nazwy powiązanej encji oraz pól, które mają być dostępne w trakcie zapytań. System sam zarządza takimi aspektami jak włączenie dodatkowych kolumn do definicji tabeli, dodanie pól do obiektu modelującego oraz aktualizacja powiązanych wartości. W typowym systemie ORM użytkownik musiałby jawnie zdefiniować pola modelu, a następnie w parametrach konfiguracyjnych określić sposoby zarządzania każdym z tych pól osobno.

- W tradycyjnym systemie ORM optymalizacja modelu kolejki danych wymagałaby rozbudowania wszystkich odwołań do danych w całej aplikacji. Modyfikacja wymagałaby dodania warunków, które nie stanowią logicznej części zapytania, a jedynie zwiększają szybkość pobierania danych. W mechanizmie modelowania obiektowego dla Cassandry wystarczy dodanie odpowiedniego parametru do modelu. System samodzielnie śledzi niezbędne wartości i modyfikuje odwołania w sposób niewidoczny dla kodu aplikacji.

[4]Przykładem niejawności jest użycie tabeli pośredniej w modelowaniu relacji typu wiele-do-wielu. Tabela nie jest zdefiniowana przez klasę w kodzie. Wymagane jest podanie odpowiednich parametrów konfiguracyjnych pola.

OMC dostarcza szereg funkcjonalności, które usprawniają zarządzanie danymi i wykraczają poza zakres mapowania obiektowo-relacyjnego. Należy do nich system migracji, który umożliwia automatyczną aktualizację modelu pomiędzy wersjami. Jest to kluczowe zagadnienie w przypadku rozbudowy istniejących aplikacji korzystających z OMC:

- Bez mechanizmu migracji użytkownik musiałby ręcznie zaktualizować schemat danych. Rozbudowując istniejącą tabelę często zachodzi konieczność manualnego uzupełnienia wszystkich istniejących rekordów danych. Reguły aktualizacji nie zawsze są trywialne i mogą wymagać tworzenia rozbudowanych skryptów/aplikacji.

- Mechanizm migracji wykonuje powyższe zadania automatycznie. Pozwala na tworzenie skomplikowanych reguł aktualizacji istniejących rekordów z wykorzystaniem kodu źródłowego. Kod ten może odwoływać się do metod interfejsu OMC, a więc w szczególności odwoływać się do powiązanych danych.

Kolejnym wykraczającym poza zakres mapowania obiektowo-relacyjnego przykładem zarządzania danymi jest mechanizm do populacji wartości testowych. Umożliwia on inicjalne wypełnienie pustego schematu bazy danych. Automat wykorzystuje naiwną strategię generacji danych, która może być kontrolowana przez użytkownika w dowolnym zakresie. Może być wykorzystywany zarówno do testowania wydajności modelu, jak również wykonywania migracji z innego źródła danych.

Podsumowując porównanie, tradycyjne mapowanie obiektowe (dostosowane do charakterystyki systemu bazodanowego) jest z założenia jedną z części modelowania obiektowego dla Cassandry. Pakiet OMC ułatwia proces opisywania danych poprzez dostarczenie abstrakcji niezwiązanych z fizyczną strukturą/wykonywanymi odwołaniami. Dostarcza także dodatkowe funkcjonalności zarządzania danymi, bez których korzystanie z mapowania obiektowego w zastosowaniach produkcyjnych jest znacznie utrudnione.

4.3 OMC - podstawowe pojęcia

Środowisko aplikacyjne OMC definiuje trzy podstawowe pojęcia: model, pole oraz silnik.

4.3.1 Model

Jest to klasa, która opisuje atomowy względem kodu źródłowego obiekt modelujący bazę danych. Obiekt ten, w zależności od definicji, może być mapowany na jedną lub wiele tabel. Jedynym wymaganiem, które musi spełnić klasa należąca do modelu jest dziedziczenie po klasie bazowej `Model`. Ciało modelu stanowią pola.

4.3.2 Pole

Pole jest to zmienna klasy definiującej model. Opisuje atomową wartość z punktu widzenia kodu źródłowego. W zależności od typu pole może być mapowane na jedną kolumnę, wiele kolumn lub autonomiczną tabelę w bazie danych. Aby pole klasy mogło zostać zakwalifikowane jako pole modelu, w definicji klasy należy mu przypisać obiekt dziedziczący po klasie bazowej `Field`.

4.3.3 Silnik

Silnik jest to obiekt, który odpowiada za nawiązywanie połączenia z bazą danych. Po utworzeniu silnika wystarczy przypisać go do całego modelu. Wszystkie operacje bazodanowe będą domyślnie wykonywane przez przypisany silnik.

Utworzenie silnika wymaga podania adresu URL opisującego połączenie, nazwanego z języka angielskiego *connection string*. Strukturę tego adresu przedstawia listing 4.4.

Wszystkie elementy ujęte w znaki **{}** w adresie 4.4 należy zamienić na wartości według poniższego objaśnienia:

```
cassandra://{ip}:{port}/{keyspace}?rf={rf}&strategy={strategy}
```

Rysunek 4.4: Struktura adresu URL opisującego połączenie z bazą danych Cassandra.

ip adres IP węzła koordynatora, z którym będzie komunikować się kod źródłowy,

port port IP koordynatora (podanie portu jest opcjonalne; wartość domyślna to 9042),

keyspace przestrzeń nazw Cassandry, w której operuje OCM,

rf współczynnik replikacji[5] podawany przy tworzeniu przestrzeni nazw, jeżeli ta nie istnieje (parametr jest opcjonalny),

strategy nazwa strategii replikacji podawanej przy tworzeniu przestrzeni nazw, jeśli ta nie istnieje (parametr jest opcjonalny).

Do utworzenia silnika należy wykorzysać metodę klasy `Engine` o nagłówku `create_engine(connection_string)`. Aby przypisać silnik do modelu należy użyć metody `Model.bind(engine)`. Przykład wywołania przypisującego modelowi silnik dla serwera lokalnego i przestrzeni nazw `test` został zaprezentowany na listingu 4.5.

```
engine = Engine.create_engine('cassandra://127.0.0.1/test')
Model.bind(engine)
```

Rysunek 4.5: Przykład wywołania tworzącego silnik i przypisującego go do modelu.

4.4 Definiowanie modelu

Definiowanie modelu odbywa się poprzez stworzenie nowej klasy, która dziedziczy po klasie bazowej `Model`. Podstawowy model opisuje zawartość jednej

[5]Współczynnik replikacji (ang. replication factor) - określa na ile węzłów kopiowany jest każdy wstawiony wiersz.

tabeli, jego pola zaś odpowiadają kolumnom tej tabeli. Przykładowy model został przedstawiony na listingu 4.6.

```
class SampleTableModel(Model):
    id = UuidField(type=Uuid, partitioning_key=True)
    sample_field = TextField()
```

Rysunek 4.6: Przykładowy model danych OMC.

Po zdefiniowaniu klasy modelu oraz utworzeniu i przypisaniu silnika jak na listingu 4.5 uruchomi się algorytm weryfikacji schematu bazodanowego. Algorytm działa następująco:

1. Sprawdzenie czy jest zdefiniowana przestrzeń nazw (dla przykładu: `test`).
2. Jeżeli przestrzeń nazw nie jest zdefiniowana zostanie ona utworzona.
3. Dla każdego zdefiniowanego modelu sprawdź czy istnieje odpowiadająca mu tabela.
4. Jeżeli tabela istnieje, sprawdź czy definicja tabeli odpowiada modelowi. Jeśli nie, zgłoś wyjątek modelu danych.
5. Jeżeli tabela nie istnieje zostanie utworzona.

W wyniku działania algorytmu zostanie utworzona tabela z wykorzystaniem zapytania przedstawionego na listingu 4.7. Należy zauważyć, że OCM w schemacie bazodanowym wykorzystuje notację z podkreśleniami[6]. Gdyby klucz główny nie został zdefiniowany jawnie, mechanizm utworzyłby go sztucznie jako autogenerowane pole o nazwie `id` i typie `timeuuid`.

```
CREATE TABLE sample_table_model (id PRIMARY KEY, sample_field text);
```

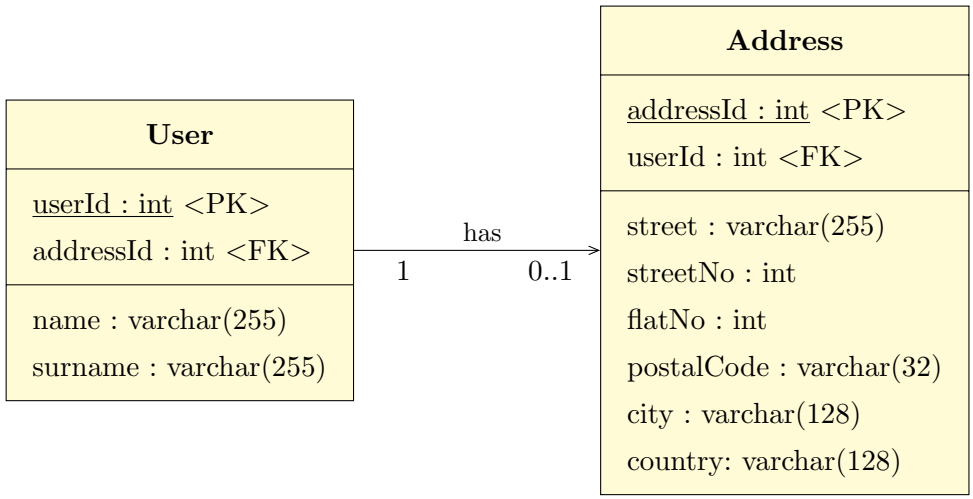
Rysunek 4.7: Zapytanie tworzące tabelę dla modelu 4.6.

[6]Underscore notation - zapis, w którym używane są wyłącznie małe litery, a poszczególne wyrazy są oddzielone znakiem _, na przykład: underscore_notation_example.

4.5 Modelowanie zależności między danymi

Autor celowo nie używa terminu „relacja", aby uniknąć skojarzenia z RDBMS[7]. Jak wykazały badania przeprowadzone w sekcji 3.2.1 tradycyjne podejście relacyjne w Apache Cassandra jest bardzo nieefektywne i powinno być unikane.

W przypadku relacyjnych baz danych modelowanie wymaga określenia definicji modelowanych obiektów - encji oraz zależności między nimi. Na diagramie 4.8 zaprezentowano efekt modelowania prostej dziedziny danych, która umożliwia przechowywanie adresu dla użytkownika. Charakterystyka RDBMS pozwala na pobranie danych użytkownika i adresu w jednym odwołaniu do bazy danych.



Rysunek 4.8: Modelowanie adresu użytkownika w relacyjnej bazie danych.

Model ten można dokładnie odwzorować w Apache Cassandra, jednakże jest to rozwiązanie niewydajne. Charakterystyka bazy sprawia, że sprawdzenie kraju zamieszkania użytkownika wymaga wykonania dwóch osobnych zapytań: pobrania użytkownika, a następnie wskazywanego przez odpowiadający mu wiersz adresu. Czas przeznaczony na wykonanie jednego zapytania można aproksymować wzorem:

[7]Relative Database Management Systems - relacyjne systemy zarządzania bazami danych

$$t_{SELECT} = t_{AC} + t_{HASH} + t_{CN} + t_{GET} + t'_{CN} + t'_{AC} \tag{4.1}$$

gdzie t_{AC}, t'_{AC} to czas komunikacji (i komunikacji zwrotnej) pomiędzy aplikacją a koordynatorem, t_{CN}, t'_{CN} to czas komunikacji (i komunikacji zwrotnej) pomiędzy koordynatorem a węzłem przechowującym wyszukiwany wiersz, t_{HASH} to czas obliczania funkcji skrótu dla klucza danego wiersza, a t_{GET} to właściwy czas pobrania wiersza. Pomiędzy czynnikami zachodzi zależność:

$$t_{AC}, t'_{AC}, t_{CN}, t'_{CN} >> t_{HASH}, t_{GET}. \tag{4.2}$$

Wynika stąd, że wykonanie dodatkowego zapytania wprowadza opóźnienia związane głównie z połączeniem pomiędzy poszczególnymi maszynami. Opóźnienia te w większości przypadków są relatywnie niskie (rzędu kilku milisekund), więc nie jest to krytyczny problem wydajnościowy. Ponieważ można ich uniknąć, należy traktować je jako błąd projektowania modelu.

Ze względu na specyfikę modelu danych Apache Cassandry poprawnym podejściem do projektowania schematu danych jest wyjście od definicji encji i wykonywanych na nich zapytań. Należy zauważyć, że zapytanie jest w pewnym sensie nadzbiorem w stosunku do relacji. Nie tylko zawiera ono informację, że dwa obiekty są ze sobą połączone, ale reprezentuje też sposób tego połączenia pozwalający na uzyskanie wymaganych informacji. Ponieważ zapytania najczęściej odwołują się do połączonych fragmentów wielu encji podstawowym „narzędziem" modelowania Cassandry jest denormalizacja. Jest to zasadne zwłaszcza w kontekście wydajnych operacji zapisu, które oferuje baza.

Niech w przykładzie 4.8 omawiany będzie model danych dla strony internetowej, na której występują następujące odwołania do modelu danych:

1. W głównym szablonie strony przewidziane jest miejsce na nazwę użytkownika i flagę, które są wyświetlane po zalogowaniu się.

2. W panelu administracyjnym wyświetlana jest lista użytkowników, która może być filtrowana po mieście i kraju zamieszkania. Nie są na niej prezentowane pełne dane adresowe.

3. Pełny adres jest widoczny po kliknięciu na osobną podstronę profilu użytkownika.

Z punktów 1 i 2 wynika, że w kontekście użytkownika pobierane są tylko dwie składowe adresu: kraj i miasto. W przypadku punktu 3 można wykonać osobne zapytanie, gdyż adres wyświetlany jest na podstronie. Poprawny projekt modelu danych, który umożliwia ściąganie kompletu wymaganych danych w jednym zapytaniu, został schematycznie przedstawiony na diagramie 4.9.

	name	**surname**	**addressId**	**city**	**country**
982	Jakub	Turek	1842	Warsaw	Poland

Rysunek 4.9: Poprawne rozwiązanie problemu modelowania użytkownika i adresu dla zadanych wymagań.

Silna denormalizacja pozwala na budowanie efektywnych i bardzo szybkich modeli danych. W zamian wymaga ona dużego nakładu planowania. Wprowadzanie zmian do modelu zdenormalizowanego jest trudne, gdyż struktura istniejących danych jest bardzo usztywniona. W niektórych przypadkach uzysk wydajności wynikający z wykorzystania struktury zdenormalizowanej na korzyść znormalizowanej jest niewielki, a poniesione koszta sprawiają, że cały proces jest nieopłacalny. Z tego względu mechanizm OMC zapewnia szerszy wybór narzędzi do modelowania zależności pomiędzy elementami modelu danych. Narzędzia te odzwierciedlają rozważania opublikowane na blogu technologicznym portalu aukcyjnego eBay - jednego z większych użytkowników Apache Cassandry. [15].

OMC posiada trzy wbudowane tryby, które automatyzują proces zarządzania zależnościami między danymi. Różnią się one pomiędzy sobą wydaj-

nością (mierzoną w liczbie zapytań niezbędnych do pobrania kompletu wymaganych danych) oraz stopniem normalizacji:

Denormalizacja przez pole Umożliwia włączenie zdenormalizowanych pól do zamodelowanej encji. Tworzy model jednostopniowy (do pobrania kompletu informacji wymagane jest jednokrotne pobranie wiersza z tabeli).

Denormalizacja przez tabelę Umożliwia ekstrahowanie wszystkich zdenormalizowanych zależności do osobnej tabeli. Tworzy model dwustopniowy (do pobrania kompletu informacji wymagane jest dwukrotne pobranie wiersza - z tabeli nadrzędnej oraz tabeli zdenormalizowanej).

Normalizacja przez tabelę Umożliwia wiązanie zależności poprzez osobną tabelę. Tworzy model trzystopniowy (do pobrania kompletu informacji wymagane jest trzykrotne pobranie wiersza - z tabeli nadrzędnej, tabeli mapowania oraz tabeli wskazywanej).

4.5.1 Denormalizacja przez pole

Mechanizm denormalizacji przez pole pozwala wskazać encję, której wybrane pola zostaną automatycznie dołączone w wybrane miejsce modelu. Po wskazaniu encji zależnej domyślnie przenoszony jest klucz partycjonowania tej encji. Jest on również dołączany do klucza klastrowania modelu. Dzięki temu możliwe jest zdefiniowanie wielu połączeń i docieranie od obiektu nadrzędnego do obiektu zależnego.

Denormalizacja przez pole wykorzystuje typ `DenormalizedField`. Jego działanie zostanie omówione na podstawie definicji danych zawartej na diagramie 4.9. Listing 4.10 przedstawiono definicję encji adres zapisaną w OCM. Model użytkownika został pokazany na listingu 4.11.

Pole `DenormalizedField` używa dwóch parametrów konfiguracyjnych. Opcja `relates` przechowuje nazwę modelu, który jest wskazywany przez dane pole. Parametr `fields` definiuje nazwy pól encji wskazywanej, które mają

```
class Address(Model):
    address_id = UuidField(partitioning_key=True)
    street = TextField()
    street_no = IntegerField()
    flat_no = IntegerField()
    postal_code = TextField(length=32)
    city = TextField()
    country = TextField()
```

Rysunek 4.10: Denormalizacja przez pole - definicja encji adres.

```
class User(Model):
    user_id = UuidField(partitioning_key=True)
    name = TextField()
    surname = TextField()
    address = DenormalizedField(relates=Address,
                                fields=['city', 'country'])
```

Rysunek 4.11: Denormalizacja przez pole - definicja encji użytkownik.

zostać włączone do definicji modelu. Ważne jest, że w celu uniknięcia kolizji nazw dołączane pola nazywane są według specjalnego klucza łączącego nazwę pola w modelu nadrzędnym z polami w encji zależnej. W podanym przykładzie dołączane pola będą miały nazwy: `address_address_id`, `address_city` oraz `address_country`.

W porównaniu do ręcznej denormalizacji użycie pola `DenormalizedField` wprowadza liczne usprawnienia:

- Cały klucz partycjonowania encji wskazywanej jest automatycznie włączany do modelu. Dzięki temu zawsze możliwe jest przejście od obiektu nadrzędnego do podrzędnego.
- Wykorzystanie zasady DRY[8] przy definiowaniu denormalizowanych pól. Gdyby w modelu Address deklaracja typu pola `country` została zmie-

[8]Don't Repeat Yourself (ang. „nie powtarzaj się") - zasada, która w kontekście programowania zaleca powielania tych samych lub bardzo podobnych fragmentów kodu w wielu miejscach.

niona na `IntegerField()` (kod kraju zamiast nazwy), typ denormalizowanego pola w modelu `User` zostałby automatycznie uaktualniony.

- Generowany jest mechanizm automatycznego rozpakowywania wartości denormalizowanych z obiektu modelu. Zakładając, że zmienna `address` przechowuje obiekt typu `Address`, a `user` obiekt typu `User`, wywołanie `user.address = address` spowoduje automatyczne przypisanie wartości pól `address_*` zmiennej `user`.

- Generowana jest metoda do automatycznego pobierania wskazywanego wiersza danych. Wywołanie `user.address.get()` zwróci obiekt typu `Address`, pod warunkiem, że klucz `user.address_address_id` jest poprawny.

- W przypadku aktualizacji danych encji zależnej mechanizm potrafi na życzenie zaktualizować również wszystkie zdenormalizowane wartości, które wskazują na ten obiekt. Można tego dokonać wywołując metodę `update(update_related=True)` na obiekcie modelu.

Pole `DenormalizedField` można używać również przy pustej liście parametru `fields`. Wtedy dowiązywany jest wyłącznie identyfikator encji zależnej. Możliwe jest również korzystanie z funkcji `get()`, która pobierze obiekt reprezentujący zależny wiersz.

Denormalizacja przez pole jest najbardziej efektywnym czasowo sposobem modelowania relacji. Wynikowy model danych jest tożsamy z przedstawionym na diagramie 4.9.

4.5.2 Denormalizacja przez tabelę

Mechanizm denormalizacji przez tabelę działa analogicznie do denormalizacji przez pola, z następującymi różnicami:

- Zdenormalizowane dane przechowywane są w osobnej tabeli, a nie w obiekcie nadrzędnym.

- Zdenormalizowane dane nie są dostępne bezpośrednio jako pola obiektu, z którego następuje odwołanie.
- Domyślnie (przy braku specyfikacji listy pól) denormalizowane są wszystkie pola obiektu zależnego.

Na listingu 4.12 przedstawiono przykład denormalizacji przez tabelę opisanej jako model OMC. Ze względu na brak wyspecifikowanych pól (parametr `fields`) do tabeli denormalizacyjnej zostaną włączone wszystkie pola modelu `Address`. Schemat tabeli denormalizacyjnej, która powstanie na skutek zadeklarowania przykładowego modelu został zaprezentowany na diagramie 4.13.

```
class User(Model):
    user_id = UuidField(partitioning_key=True)
    name = TextField()
    surname = TextField()
    address = DenormalizedTable(relates=Address, model='UserAddress')
```

Rysunek 4.12: Denormalizacja przez tabelę - definicja encji użytkownik.

	addressId	**street**	**no**	**code**	**city**	**country**
233	754	Nowowiejska	15/19	00-665	Warsaw	Poland

Rysunek 4.13: Denormalizacja z wykorzystaniem tabeli. Wszystkie zdenormalizowane informacje są przechowywane w osobnej tabeli.

Do obsługi zdenormalizowanej tabeli generowana jest prosta klasa modelu. Opakowuje ona dostęp do zdenormalizowanych pól. Parametr `model` specyfikuje nazwę klasy tego modelu, którą następnie można wykorzystywać w kodzie. Dostęp do wartości zdenormalizowanych dostępny jest po pobraniu obiektu zależnego. Zakładając, że zmienna `user` jest instancją klasy `User`, wywołanie `user.address.get()` zwraca dane z wiersza zdenormalizowanej tabeli. Z wyniku wykonania metody można uzyskać później wartości pól, przykładowo `user.address.get().postal_code` dla kodu pocztowego.

4.5.3 Normalizacja przez tabelę

Normalizacja przez tabelę polega na wyekstrahowaniu mapowania identyfikatorów encji wzajemnie zależnych $(id_{relates}, id_{related})$ do osobnej tabeli. Na poziomie modelu danych przypomina to wykorzystanie tabeli pośredniczącej do modelowania relacji wiele-do-wielu w relacyjnych bazach danych. Podstawową różnicą jest jednak brak zwrotności. Tabela normalizująca opisuje połączenie jednokierunkowe i nie ma możliwości efektywnego odwrócenia tego związku bez utworzenia kolejnej tabeli normalizującej w drugim kierunku.

Przykład wykorzystania normalizacji przez tabelę w modelu OMC został przedstawiony na listingu 4.14. Fizyczny model danych stworzony dla przykładowego opisu przedstawia diagram 4.15. Mapowanie powiązań w OMC przy pomocy tabeli normalizacyjnej (`user_address_norm_table`) zachodzi niejawnie. Wywołanie w kodzie aplikacji metody `user.address.get()` zwracającej listę adresów użytkownika wykonuje wiele odwołań do bazy danych. Pierwsze pobiera z tabeli normalizującej wszystkie identyfikatory powiązanych obiektów. Kolejne zapytania ściągają do pamięci szczegóły tych obiektów.

```
class User(Model):
    user_id = UuidField(partitioning_key=True)
    name = TextField()
    surname = TextField()
    address = NormalizedTable(relates=Address)
```

Rysunek 4.14: Normalizacja przez tabelę - implementacja encji użytkownik.

Na podstawie opisu działania metody `get()` łatwo stwierdzić, że takie rozwiązanie bardzo źle skaluje się ze wzrostem gęstości połączeń. Przeciętna ilość zapytań niezbędnych do pobrania kompletu informacji z bazy danych wynosi:

$$|q_{avg}| = 1 + r_{avg} \tag{4.3}$$

gdzie r_{avg} jest średnią liczbą obiektów wskazywanych przez encję nadrzęd-

user_address_norm_table

	addressId
583	678

address

	street	**no**	**code**	**city**	**country**
678	Nowowiejska	15/19	00-665	Warsaw	Poland

Rysunek 4.15: Model danych tabeli normalizacyjnej. Identyfikatorem wiersza w tabeli *user_address_norm_table* jest identyfikator użytkownika.

ną. Dla licznych, gęstych zbiorów danych oznacza to występowanie sytuacji, w których należy n osobnych zapytań dla wszystkich n elementów zapytań, które znajdują się w zbiorze.

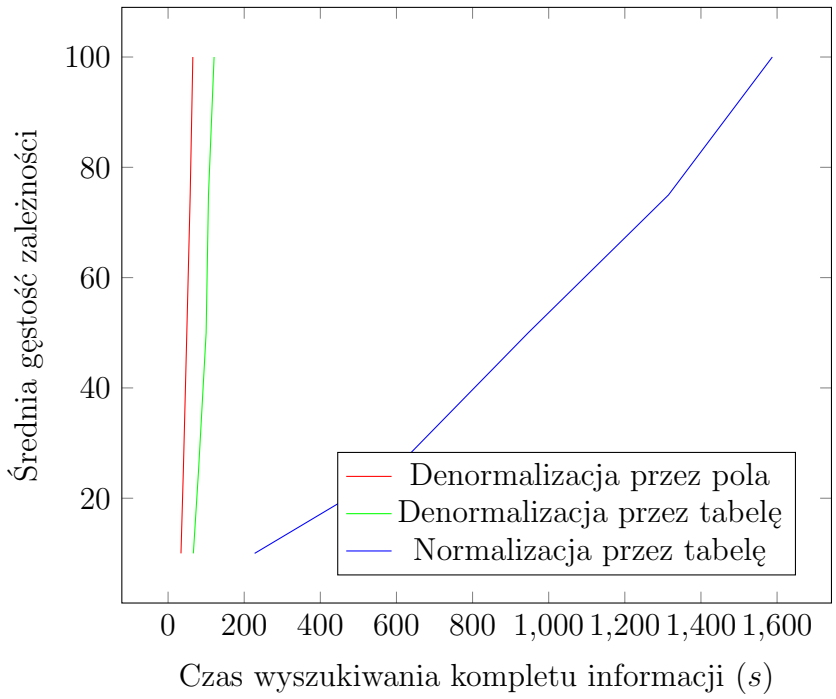
4.5.4 Porównanie wydajności metod zarządzania zależnościami

Trzy zaprezentowane metody różnią się od siebie wydajnością poszczególnych operacji w kontekście charakterystyki danych. Aby ułatwić wybór rodzaju modelowania w zależności od scenariusza użytkownia Autor przeprowadził szereg testów wydajnościowych, które ukazują zalety i wady poszczególnych typów modelowania. Wszystkie testy zostały przeprowadzone dla modelu danych użytkowników i przypisanych do nich adresów.

Skalowalność operacji wyszukiwania względem gęstości połączeń Celem pierwszego testu jest zbadanie zależności pomiędzy wydajnością operacji pobierania kompletu danych, a gęstością połączeń pomiędzy encjami zależnymi. Gęstość połączeń to średnia ilość podzależnych elementów przypadających na jeden element nadrzędny.

W trakcie testu zostało wygenerowanych 10000 użytkowników oraz 1000 adresów. W poszczególnych iteracjach mierzone były czas pobierania wszystkich użytkowników, wraz ze wszystkimi wymaganymi informacjami adresowymi, ale dla różnej gęstości połączeń pomiędzy danymi. Wyniki testu zo-

stały przedstawione na wykresie 4.16.



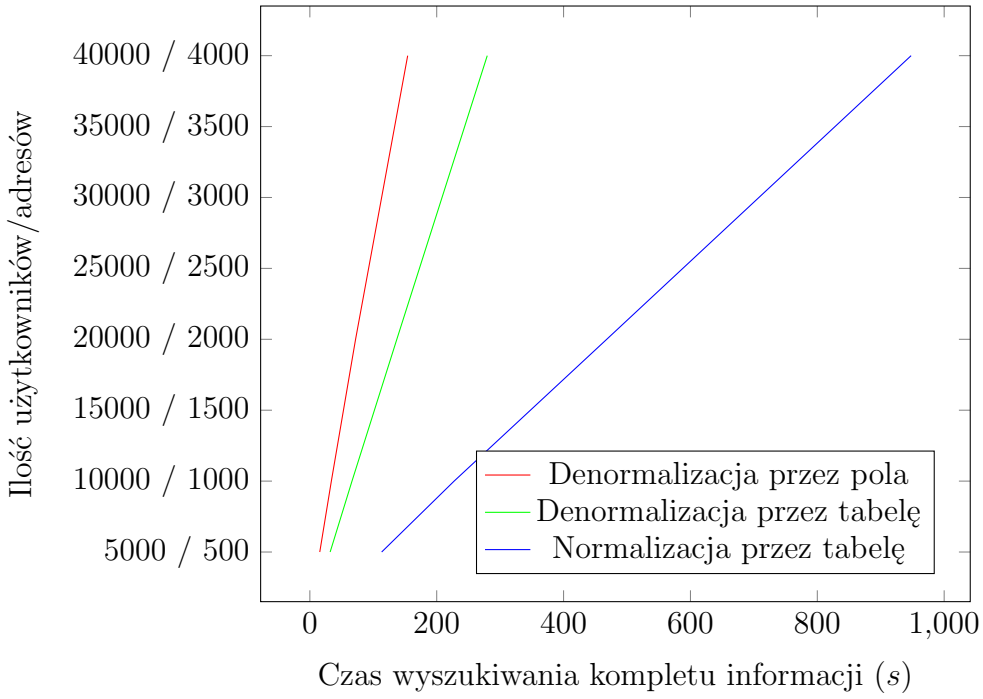
Rysunek 4.16: Porównanie czasu pobierania informacji dla różnych metod modelowania zależności (przy rosnącej gęstości połączeń).

Z wykresu można odczytać ogromną przewagę modeli zdenormalizowanych nad modelem znormalizowanym przy masowym odczycie wielu rekordów. Jest to zgodne z prognozami przedstawionymi w sekcji 4.5.3. Liczba zapytań potrzebnych do pobrania kompletu informacji z wykorzystaniem normalizacji przez tabelę rośnie wraz ze wzrostem encji w zbiorze danych, jak również z gęstością połączeń. Uzyskany wykres ma charakterystykę nieliniową. W przypadku denormalizacji przez pola/tabelę uzyskane wyniki mają charakterystykę w przybliżeniu liniową. Ilość wykonywanych zapytań zależy wyłącznie od liczby elementów w zbiorze danych.

Należy zauważyć, że wykres nie oddaje pełni różnic pomiędzy modelami. Ponieważ czas dostępu dla zależności znormalizowanej odbiega znacząco od pozostałych, skala nie odzwierciedla różnicy pomiędzy podejściami zdenormalizowanymi. Zgodnie z przewidywaniami mechanizm denormalizacji przez

pole jest dla wszystkich punktów pomiaru w przybliżeniu dwukrotnie szybszy od denormalizacji przez tabelę, ponieważ do pobrania informacji należy wykonać dwa razy mniej zapytań.

Skalowalność operacji wyszukiwania względem ilości encji Celem tego testu jest zbadanie szybkości wyszukiwania kompletu danych, kiedy gęstość połączeń w modelu jest stała, a zmienia się ilość encji nadrzędnych oraz zależnych. Wszystkie badania zostały przeprowadzone dla średniej gęstości połączeń wynoszącej 10. W kolejnych iteracjach liniowo zwiększana była ilość wierszy użytkownika i ilość adresów. Wyniki badań zostały przedstawione na wykresie 4.17.

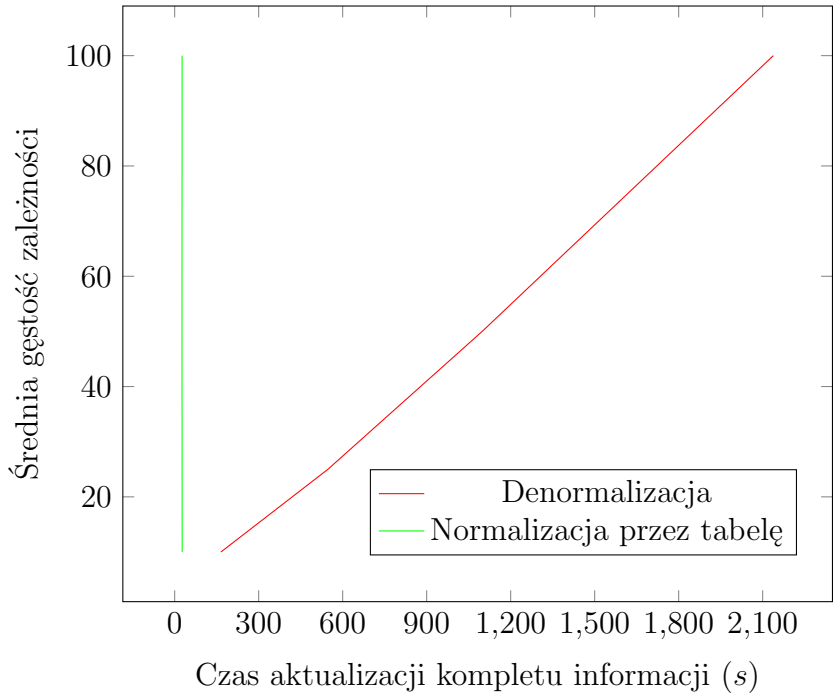


Rysunek 4.17: Porównanie czasu pobierania informacji dla różnych metod modelowania zależności (przy rosnącej ilości wierszy).

Kolejny raz efektywność zależności znormalizowanej przez tabelę znacząco odstaje od efektywności zależności zdenormalizowanych. Jednakże w przeciwieństwie do gęstości zależności, zwiększanie liczby wierszy powoduje linio-

wy wzrost czasu potrzebnego na pobranie kompletu danych. Wynika z tego, że model zdenormalizowany dużo lepiej sprawdza się w przypadku rzadkich zbiorów połączeń pomiędzy encjami. Podobnie jak w przypadku poprzedniego badania, model zdenormalizowany poprzez pola okazuje się dwa razy szybszy od denormalizacji z użyciem osobnej tabeli.

Skalowalność operacji aktualizacji względem gęstości połączeń Celem testu jest porównanie szybkości aktualizacji danych zależnych w przypadku różnych typów modelowania. W trakcie testu zostało wygenerowanych 10000 użytkowników i 1000 adresów. Zmieniana była gęstość połączeń pomiędzy encjami danych. Aktualizacji podlegały wszystkie elementy ze zbioru zależnych danych (adresy). Wyniki testu zostały przedstawione na wykresie 4.18.



Rysunek 4.18: Porównanie czasu aktualizacji informacji dla różnych metod modelowania zależności (przy rosnącej gęstości połączeń).

Z wykresu wynika, że czas aktualizacji rekordów znormalizowanych jest

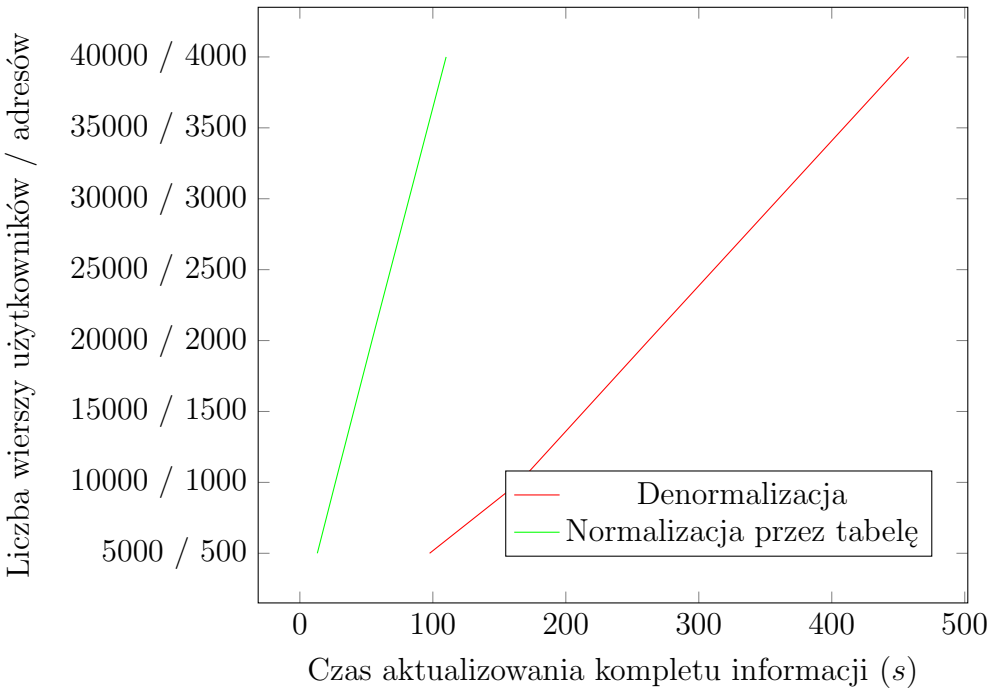
niezależny od gęstości połączeń. W modelu znormalizowanym wystarczy zaktualizować wszystkie encje zależne, więc mierzony czas faktycznie powinien być stały względem gęstości. Aktualizacja modelu znormalizowanego jest bardzo efektywna. Czas aktualizacji 1000 przedmiotów na środowisku lokalnym (przetwarzanie jednowątkowe) wynosił niewiele poniżej 30 sekund.

Oba modele zdenormalizowane mają fatalną wydajność aktualizacji kompletu danych. Problemem jest brak możliwości efektywnego wyszukiwania identyfikatorów wszystkich użytkowników, którzy są powiązani z danym adresem. Klauzula aktualizacji w Apache Cassandra nie wspiera wykorzystania indeksów drugiego stopnia. Bez wykorzystania indeksów drugiego stopnia nie jest możliwe znalezienie encji nadrzędnych, które są powiązane z daną encją zależną. Stąd wykonanie zestawu operacji `UPDATE` jest zawsze poprzedzone wyszukiwaniem `SELECT`. Kolejną przeszkodą jest ilość aktualizowanych danych. W przypadku modelu znormalizowanego wystarczyło wykonanie $|Z|$ operacji, gdzie Z to zbiór elementów zależnych (adresów). Dla modelu zdenormalizowanego liczba ta jest zwielokrotniona przez średnią gęstość połączeń pomiędzy wierszami.

Skalowalność operacji aktualizacji względem ilości encji Celem testu jest porównanie szybkości aktualizacji danych zależnych w przypadku stałej gęstości zależności dla różnych typów modelowania. Średnia gęstość w trakcie wszystkich iteracji testów wynosi 10. Liniowo zmieniają się natomiast liczby wierszy w tabelach użytkowników i adresów. Wyniki testu zostały przedstawione na wykresie 4.19.

Podobnie jak w poprzednim badaniu wydajności aktualizacji, efektywniejszy okazał się model znormalizowany. Mierzony czas rośnie proporcjonalnie do ilości nadpisywanych rekordów. Dla zadanego problemu jest to rozwiązanie optymalne.

W przypadku denormalizacji można zauważyć, że wzrost liczby wierszy w tabeli ma łagodniejszy wpływ na charakterystykę wykresu czasu aktualizacji niż rosnąca gęstość połączeń. Przykładowo dwukrotny wzrost gęstości



Rysunek 4.19: Porównanie czasu aktualizacji informacji dla różnych metod modelowania zależności (przy rosnącej ilości wierszy).

połączeń odnosi podobny skutek czasowy co ośmiokrotny wzrost ilości danych.

4.5.5 Wnioski z analizy wydajności modelowania zależności

Nie ma uniwersalnego sposobu, który pozwala na uzyskanie optymalnego wyniku przy mapowaniu zależności w Apache Cassandra. Dobór metody powinien zależeć od charakterystyki przechowywanych danych.

Szukając pewnego rodzaju uogólnienia Autor wprowadził pomocniczą definicję typowego zbioru danych. Posiada on zbiór następujących cech:

- Duża liczba wierszy w zbiorze (> 100000).
- Operacje wyszukiwania są znacznie częstsze niż aktualizacji. Stosunek

częstotliwości ich występowania wynosi $20 \div 1$ (istnieje 5% szansy, że użytkownik przeglądający profil zaktualizuje dane adresowe).

- Gęstość występowania zależności jest niewielka. Na jeden wiersz encji nadrzędnej przypada średnio nie więcej niż dziesięć wierszy encji zależnej.

Dla tak określonego zbioru danych zdecydowanie najlepszym wyborem jest modelowanie z wykorzystaniem denormalizacji poprzez pola. Głównym kryterium jest tutaj znacząca przewaga w czasie wyszukiwania danych i akceptowalny narzut przeznaczony na aktualizację danych. Liczne artykuły prezentujące studium przypadku użycia Apache Cassandry każą sądzić, że ten rodzaj modelowania zależności jest prawdopodobnie najlepiej dostosowany do specyfiki schematu danych.

W przypadku małych zbiorów danych (preferowana ilość wierszy w tabelach $\leqslant 10000$) o niskiej gęstości zależności warto rozważyć użycie reprezentacji znormalizowanej. Nie jest ona obarczona narzutem na aktualizację danych. Zaletą tej reprezentacji jest większa elastyczność schematu. W przeciwieństwie do postaci zdenormalizowanej, gdzie może zdarzyć się, że przy dodaniu kolejnego zapytania trzeba rozszerzyć tabelę denormalizacji o kolejną kolumnę, zapewnia on dostęp do kompletu danych bez konieczności wprowadzania zmian w strukturze.

Reprezentacja zdenormalizowana jest zalecana w przypadku, gdy wyszukiwanie danych nie jest kluczowym kryterium. Przykładem mogą być dane służące do generacji raportów, gdzie liczba aktualizacji może znacznie przewyższać liczbę wyszukiwań.

Duży problem stanowi modelowanie zależności o dużej liczbie połączeń między danymi. Wraz ze wzrostem gęstości połączeń możliwości zawężają się do:

- Zaniedbania aktualizacji zdenormalizowanych danych. W praktyce jest to bardzo kłopotliwe i może być zastosowane jeżeli udaje się zdenormalizowanie wyłącznie danych przeznaczonych tylko do odczytu.

- Poświęcenia czasu aktualizacji kosztem czasu dostępu do danych i wyboru modelu zdenormalizowanego.
- Poświęcenia czas dostępu do danych kosztem czasu aktualizacji i wyboru modelu znormalizowanego.
- Zmiany modelowania danych. Przekształcony schemat może między innymi redukować zagęszczenie zależności lub konieczność wykonywania operacji aktualizacji.

Jeżeli żadna z powyższych możliwości nie daje się zastosować lub nie wprowadza akceptowalnej poprawy wydajności należy rozważyć wybór innego silnika bazodanowego. Dla zbiorów o bardzo gęstej siatce zależności naturalne zastosowanie mają grafowe bazy danych. Przykładem dojrzałego projektu silnika grafowego o dobrych własnościach skalowania jest Neo4j[9].

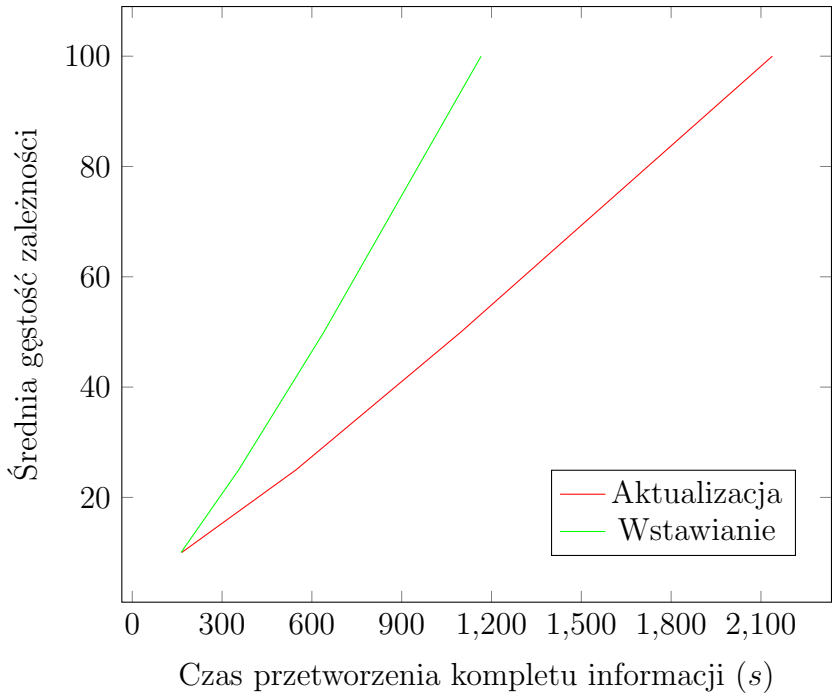
W omawianym w sekcji 4.5 przykładzie można wyeliminować operacje aktualizacji danych przy specyficznych założeniach:

- Żadna encja nie jest powiązana z użytkownikiem jako encja nadrzędna.
- W trakcie aktualizacji adresu aplikacja dysponuje pełnym kompletem informacji o użytkowniku.
- Nie istnieje ryzyko konfliktów synchronizacji (nie jest możliwa równoczesna edycja tego samego profilu przez różne instancje aplikacji).
- Zduplikowanie istniejącego adresu nie ma wpływu na poprawność danych.

Jeżeli powyższe założenia są spełnione aktualizacja może być zastąpiona poprzez wstawienie nowych danych do bazy. Ta operacja jest bardziej wydajna ze względu na pominięcie etapu wyszukiwania użytkowników, którzy są przypisani do danego adresu. Porównanie szybkości aktualizacji i wstawiania

[9]Strona domowa projektu Neo4j dostępna jest pod adresem `http://www.neo4j.org`.

nowych danych do bazy (z unieważnianiem poprzednich) zostało przedstawione na wykresie 4.20. Testy zostały przeprowadzone na środowisku lokalnym dla stałej liczby wierszy w tabelach (10000 użytkowników oraz 1000 adresów). W kolejnych iteracjach modyfikowana była średnia gęstość zależności w modelu. Modyfikacji / wstawianiu podlegały wszystkie wiersze.



Rysunek 4.20: Porównanie czasu wstawiania i aktualizacji informacji (przy rosnącej gęstości połączeń).

Wyniki pokazują, że dla gęstych modeli danych operacja wstawiania może być nawet dwukrotnie wydajniejsza niż aktualizacja. Zastosowanie wielowątkowej aplikacji do przeprowadzenia testów mogło dalej pogłębić te różnicę. Z wyniku testu można wywnioskować, że odnajdywanie/nakładnie odpowiednich ograniczeń na dane i przekształcanie modelu może znacząco poprawiać wydajność również w przypadkach, w których w ogólności Cassandra nie sprawdza się zbyt dobrze.

Na podstawie porównania wydajności różnych metod modelowania zależności między danymi można stwierdzić, że nie ma przypadku, w którym de-

normalizacja z wykorzystaniem tabeli ma przewagę nad pozostałymi dwoma metodami. W każdym przypadku może być ona zastąpiona denormalizacją przez pola/modelem znormalizowanym z uzyskaniem identycznej lub lepszej wydajności. Stosowanie tego rodzaju relacji jest uzasadnione jeżeli:

- Uzyskana wydajność jest zadowalająca, a istotne jest fizyczne uporządkowanie i wydzielenie danych (na przykład gdy część systemu wykorzystuje wydruki w formie surowej).
- Opisywane są bardzo szerokie wiersze i problemem jest przekroczenie dwóch miliardów kolumn[10].

4.6 Wsparcie dla wzorców modelowania

Jednym z pryncypiów projektowych mechanizmu OMC jest obsługa wzorców modelowania. W literaturze zostały zebrane i opisane liczne przykłady efektywnych i nieefektywnych projektów modelu danych dla Apache Cassandry. Z przeanalizowanych opisów Autor wybrał zbiór reużywalnych wzorców. Następnie, dla każdego wzorca, przygotowany został mechanizm, który ułatwia jego modelowanie w środowisku OMC. Dzięki temu mechanizm został wzbogacony o obsługę:

- Szeregów zdarzeń, czyli zbioru uporządkowanych względem czasu wiadomości.
- Kolejek *First In, First Out*, *Last In, Last Out* i podobnych.
- Indeksów wartości unikalnych.
- Aktualizacji selektywnej.

[10] Apache Cassandra narzuca limit dwóch miliardów kolumn w jednym wierszu.

4.6.1 Szereg zdarzeń

Apache Cassandra została zaprojektowana z myślą o wspieraniu szeregów zdarzeń. Szereg zdarzeń to uporządkowane względem czasu wystąpienia wpisy do bazy danych, które opcjonalnie mogą posiadać z góry określony czas życia. Typowym przykładem szeregu zdarzeń są wartości odczytów z urządzeń pomiarowych. Wartości te zwyczajowo są pobierane cyklicznie, a do wykorzystania w czasie rzeczywistym przydatne są tylko najnowsze odczyty. Doskonały opis modelowania szeregów zdarzeń przedstawiony jest we wpisie Patricka McFadina na portalu **Planet Cassandra**. [31] Artykuł analizuje przykład stacji meteorologicznych. Przedstawia trzy sposoby zapisu szeregu zdarzeń w Apache Cassandra.

Odczyty dla tego samego urządzenia w pojedynczym wierszu Najprostszą możliwością jest zebranie wszystkich próbek dla danego urządzenia w jednym wierszu tabeli. Kolejne wartości zapisywane są ze śladem czasu w nazwie kolumny. Listing 4.21 przedstawia wzorzec zapisany jako model OMC.

```
class TimeSeriesPatternOne(Model):
    weatherstation_id = TextField(partitioning_key=True)
    event_time = TimestampField(clustering_key=True, auto_on_create=True)
    temperature = TextField()
```

Rysunek 4.21: Modelowanie szeregu zdarzeń. Zapis kolejnych odczytów jako kolumny.

Zapis śladu czasu w nazwie kolumny można uzyskać poprzez dołączenie pola do złożonego klucza głównego tabeli jako *clustering key*[11]. Dołączenie to dokonuje się poprzez ustawienie flagi `clustering_key` na `True`.

Udogodnieniem dla programisty korzystającego z OMC jest możliwość automatycznej generacji śladu czasu dla chwili obecnej. Ślad czasu może

[11]Clustering key (ang. dosłownie „klucz klastrowania") - część złożonego klucza głównego tabeli, która określa w jakiej kolejności grupowane są wpisy względem klucza partycjonowania (identyfikatora wiersza).

być wygenerowany w trakcie tworzenia obiektu (`auto_on_create=True`) lub w trakcie wstawiania obiektu do bazy danych (`auto_on_save=True`).

Grupowanie odczytów dla urządzenia i fragmentu daty W przypadku gdy odczytów jest zbyt dużo by mieściły się w jednym wierszu sugerowanym rozwiązaniem jest dodatkowe partycjonowanie wierszy po komponencie daty: miesiącu, dniu, itd. Autor wzorca sugeruje utworzenie w modelu dodatkowego pola, do którego wpisywany jest komponent daty i dołączenie tego pola do klucza partycjonowania. Takie modelowanie dziedziny może prowadzić do potencjalnych błędów. Ta sama dana - ślad czasu - przechowywana jest w dwóch polach, które różnią się formatowaniem.

OMC posiada mechanizm partycjonowania dat po komponencie. W modelu znajduje się tylko jedna dana - faktyczny ślad czasu. OMC na jej podstawie automatycznie tworzy i formatuje klucz partycjonowania. Dzięki temu programista nie musi dbać o ręczne wyznaczanie komponentu daty. Listing 4.22 przedstawia opis modelu z partycjonowaniem względem dnia.

```
class TimeSeriesPatternTwo(Model):
    weatherstation_id = TextField(partitioning_key=True)
    event_time = TimestampField(clustering_key=True,
                                auto_on_create=True,
                                partitioning_by_day=True)
```

Rysunek 4.22: Modelowanie szeregu zdarzeń. Partycjonowanie zdarzeń po komponencie daty.

Pole `event_time` modelu `TimeSeriesPatternTwo` jest kluczem klastrowania. Dodatkowo posiada ono ustawioną flagę `partitioning_by_day=True`, która spełnia trzy zadania:

- Przy tworzeniu schematu tabeli dodaje automatycznie nową kolumnę `event_time_day`[12] o typie tekstowym. Pole to jest automatycznie włączane do klucza partycjonowania.

[12]Nazwa jest tworzona z nazwy pola oraz nazwy komponentu daty.

- Przy zapisywaniu obiektu do bazy danych automatycznie wypełnia wartość dodatkowego pola o ślad czasu obcięty do z dokładnością do dnia.

- Przy wyszukiwaniu obiektów po dacie automatycznie filtruje dane zarówno po wartości pola `event_time_day`, jak i `event_time`.

Dostępne jest kilka rodzajów komponentów daty, po których można dokonać partycjonowania. Różnią się one ostatnim członem nazwy parametru. Wybór obejmuje partycjonowanie po latach (`_year`), miesiącach (`_month`), dniach (`_day`), godzinach (`_hour`), minutach (`_minute`), a także sekundach (`_second`).

Odczyty z ograniczoną pamięcią Korzystając z wbudowanego mechanizmu Cassandry możliwe jest tworzenie automatycznie przedawniających się wartości. Wstawiając pozycję do bazy danych można ustawić czas życia, po którym zostanie ona usunięta. Przedawnianie się wartości może być wykorzystywane w połączeniu z odwrotną kolejnością sortowania wpisów (od najnowszego do najstarszego). Dzięki temu bardzo łatwo jest budować widoki, które prezentują kilka najnowszych informacji.

OMC zapewnia dwa rodzaje wsparcia dla czasu życia wartości. Czas życia może być ustawiany dla indywidualnych obiektów. Przy zapisywaniu obiektu należy na jego instancji wywołać metodę `save(ttl=20)`. Parametr `ttl` definiuje czas życia wartości w sekundach. Czas życia można również ustawić globalnie dla wszystkich instancji modelu. Wykorzystuje się do tego pole `__ttl__` klasy. Zaprezentowano to na listingu 4.23.

Dla każdego pola, które jest częścią klucza klastrowania można ustawić odwrotną kolejność sortowania elementów. Do tego celu należy ustawić flagę `descending_clustering` na `True`.

```
class TimeSeriesPatternThree(Model):
    __ttl__ = 20
    weatherstation_id = TextField(partitioning_key=True)
    event_time = TimestampField(clustering_key=True,
                                descending_clustering=True
                                auto_on_create=True)
    temperature = TextField()
```

Rysunek 4.23: Modelowanie szeregu zdarzeń. Przedawniające się odczyty jako kolumny z sortowaniem od najnowszego.

4.6.2 Wsparcie dla kolejek

W sekcji 2.8 został przytoczony antywzorzec kolejki. Problem został opisany przez Alekseya Yeschenko na blogu firmy DataStax. [23] Wynika on ze sposobu obsługi operacji usuwania przez silnik Apache Cassandra. Usuwanie nie jest realizowane natychmiast. Zamiast tego kasowana kolumna oznaczana jest specjalnym znacznikiem *tombstone* i rezyduje w pamięci aż do momentu, gdy nastąpi właściwe wyrzucanie. Przy dużej liczbie usunięć kolumn z wiersza może zdarzyć się, że zapytanie filtrujące z limitem będzie działało znacznie wolniej niż dla wiersza z identyczną ilością kolumn, ale bez poprzedzających usunięć. Wynika to z faktu, że stwierdzenie, że dana wartość jest „martwa" wymaga pobrania kolumny i zdeserializowania jej wartości. Jeżeli po deserializacji wykryty zostanie znacznik *tombstone*, wtedy pobierana jest kolejna wartość. Proces powtarzany jest aż do skutku.

Kolejka może być zamodelowana efektywniej w przypadku, gdy możliwe jest wskazanie miejsca, w którym rozpoczynają się nieusunięte kolumny. Przykładem może być kolejka typu FIFO[13]. Wyszukując kolejny element w kolejce można nałożyć warunek, że czas wstawienia elementu do kolejki jest późniejszy niż czas ostatnio usuniętego elementu. OMC dostarcza wsparcia dla takich kolejek, co zostało przedstawione na listingu 4.24.

Parametr `__track_deletes__` umożliwia śledzenie operacji usuwania.

[13]FIFO (First In, First Out) - rodzaj kolejki, w którym elementy obsługiwane są według kolejności wstawienia.

```
class FIFOQueue(Model):
    __track_deletes__ = ('enqueued_at', 'ASC')
    name = TextField(partitioning_key=True)
    enqueued_at = UuidField(type=TimeUuid,
                            auto_generate=True,
                            clustering_key=True)
    payload = DataField()
# create three queue elements
first_element = FIFOQueue(name = '1', payload = 'firstPayload')
second_element = FIFOQueue(name = '2', payload = 'secondPayload')
third_element = FIFOQueue(name = '3', payload = 'thirdPayload')
# save elements into database
first_element.save()
second_element.save()
third_element.save()
# delete first two elements from database
first_element.delete()
second_element.delete()
# get element with name '3'
FIFOQueue.objects().find(name='3')[:1]
```

Rysunek 4.24: Modelowanie kolejki typu FIFO w OMC.

Przyjmuje on parę łańcuchów znaków: nazwę pola, którego wartość jest śledzona podczas usuwania oraz warunek zapisywania śledzonej wartości. Na chwilę obecną obsługiwane jest śledzenie pól typu `UuidField` (w trybie `timeuuid`) oraz `TimestampField`, a także dwa warunki śledzenia: `ASC` oraz `DESC`. Warunek *ascending* oznacza, że śledzona wartość jest aktualizowana tylko w przypadku, gdy jest mniejsza od usuwanej obecnie wartości. Warunek *descending* działa odwrotnie - aktualizuje śledzoną wartość tylko w przypadku, gdy jest większa od usuwanej obecnie.

Ustawienie parametru `__track_deletes__` powoduje, że śledzona wartość jest automatycznie dodawana do wszystkich zapytań wyszukujących obiekty. Oznacza to, że ostatnie wywołanie w listingu 4.24 będzie miało ustawione trzy warunki filtrowania:

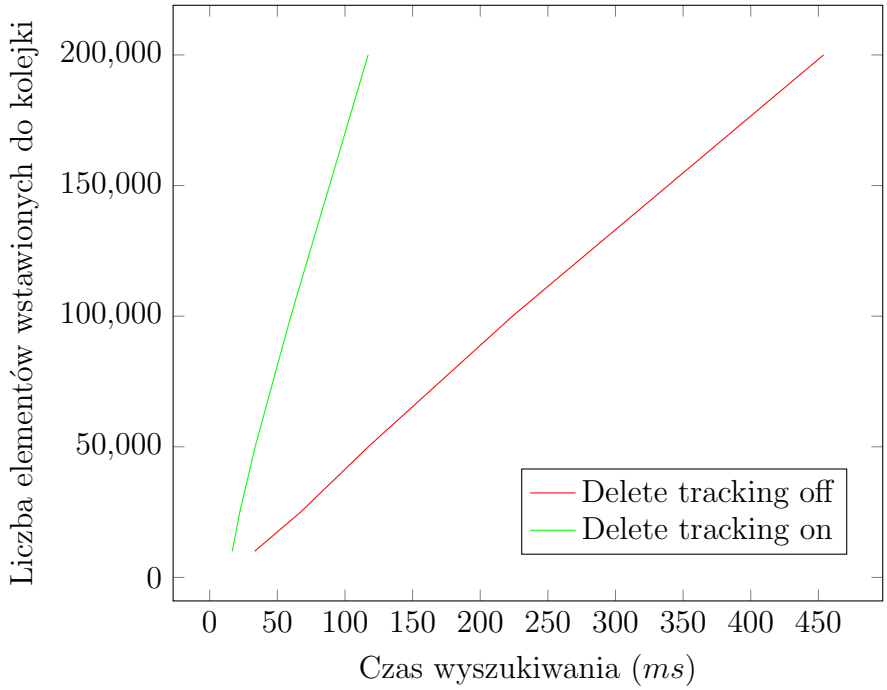
- Nazwa elementu (`name`) jest równa `'3'` - wynika z `find(name='3')`.
- Wybierz maksymalnie jeden element (`LIMIT 1`) - wynika z zakresu `[:1]`.
- Czas dodania do kolejki (`enqueued_at`) jest większy bądź równy wartości `second_element.enqueued_at` - automatycznie dodawane przez OMC.

Wykorzystanie opcji `__track_deletes__` w sposób znaczący wpływa na wydajność kolejki FIFO. Na wykresie 4.25 przedstawiono porównanie szybkości działania wyszukiwania pierwszego nieusuniętego elementu kolejki przy włączonym oraz wyłączonym śledzeniu. Pomiar wydajności został przeprowadzony na lokalnej konfiguracji klastra Apache Cassandra. Czas mierzony był przez aplikację wykorzystującą mechanizm OMC. Aplikacja ta wstawiała do kolejki y przedmiotów, po czym usuwała pierwsze $y-1$ przedmiotów z tej kolejki. Mierzony był czas wyszukiwania pierwszego elementu należącego do kolejki (z ograniczeniem `LIMIT 1`, podobnie jak przy wywołaniu metody `find` w listingu 4.24).

Pomimo znaczącej poprawy czasu dostępu do danych przy wykorzystaniu parametru `__track_deletes__` można zauważyć, że modelowanie nie jest optymalne. W przypadku braku usunięć danych z tabeli czas dostępu do pojedynczej danej powinien być w przybliżeniu stały. Z wykresu wynika jednak, że wyszukiwanie wydłuża się wraz ze wzrostem liczby elementów w kolejce. Pozwala to wyciągnąć wniosek, że efektywne modelowanie w Cassandrze powinno całkowicie eliminować lub radykalnie ograniczać ilość wykonywanych operacji usunięć.

4.6.3 Selektywna aktualizacja

Apache Cassandra jest przystosowana do obsługi wielu równoległych źródeł danych. Ze względu na brak transakcyjności jednym ze sposobów na uniknięcie problemów synchronizacji jest zalecana przez Dave'a Gardnera selektywna



Rysunek 4.25: Porównanie czasu wstawiania wielu rekordów.

aktualizacja. [24] Edytując zawartość danego wiersza należy przesyłać do bazy danych wyłącznie wartości, które rzeczywiście uległy modyfikacji. Inaczej istnieje prawdopodobieństwo niecelowego nadpisania zmian wprowadzonych równolegle przez inne źródło danych. OMC domyślnie korzysta z mechanizmu selektywnej aktualizacji, co przedstawiono na listingu 4.26.

W pierwszej kolejności metoda `save()` sprawdza, którą z dwóch operacji, aktualizacji czy wstawienia obiektu, należy wykonać. Przy wykonywaniu aktualizacji do bazy danych przesyłana jest tylko wartość zmienionych pól, w danym przykładzie tylko priorytetu. Jeżeli pomiędzy pobraniem obiektu, a aktualizacją priorytetu i zapisaniem obiektu w bazie danych inny węzeł dokona zmiany opisu obiektu, opis ten nie zostanie ponownie nadpisany poprzednią wartością.

Mechanizm aktualizacji selektywnej można wyłączyć. Można dokonać tego wywołując metodę `save()` z parametrem `selective_update=False`. Alternatywnie selektywna aktualizacja może zostać wyłączona dla całego mo-

```
class SelectiveUpdate(Model):
    name = TextField(partitioning_key=True)
    description = TextField(length=1024)
    priority = IntegerField()
# find object with name 'su_test'
su_test = SelectiveUpdate.objects().find(name='su_test').get()
# update priority
su_test.priority = 1
# update object
su_test.save()
```

Rysunek 4.26: Przykład selektywnej aktualizacji.

delu poprzez zadeklarowanie pola `__selective_update__ = False`.

4.6.4 Indeks wartości unikalnych

Aby umożliwić wyszukiwanie po wartościach kolumny, która nie należy do klucza głównego tabeli należy użyć indeksu drugiego rzędu (*secondary index*). Według dokumentacji Apache Cassandry [32] indeksy drugiego stopnia powinny być używane na kolumnach, które silnie grupują wiersze encji nadrzędnej. Wynika to z wewnętrznej implementacji indeksów drugiego stopnia. Załóżmy, że w systemie istnieje tabela przechowująca dane użytkowników, w której zapisane są wiersze zaprezentowane na diagramie 4.27.

	name	**surname**	**email**	**city**
jturek	Jakub	Turek	J.Turek@pw.edu.pl	Warsaw
	name	**surname**	**email**	**city**
manisero	Michał	Aniserowicz	M.Aniserowicz@pw.edu.pl	Warsaw

Rysunek 4.27: Przykładowe wartości w tabeli użytkownicy.

Przykładowy wygląd danych w tabeli, która realizuje indeks drugiego stopnia dla miasta (kolumna `city`), został zaprezentowany na diagramie 4.28.

Wadą rozwiązania przedstawionego na diagramie 4.28 jest brak skalowalności. Przy dodawaniu użytkowników wiersz rozszerza się o kolejne kolumny,

	jturek	**manisero**
Warsaw	null	null

Rysunek 4.28: Przykładowa tabela, która mogłaby realizować indeks drugiego stopnia dla kolumny *city* z diagramu 4.27.

które przechowywane są na tym samym węźle. Zajmuje to dużo miejsca na jednej partycji. Dodatkowo wszystkie zapytania o dany indeks zawsze odwołują się do jednego węzła, co skutkuje nierównomiernym rozłożeniem obciążenia. Ze względu na wymienione wady indeksy drugiego stopnia są realizowane lokalnie. Każdy węzeł zawiera informacje o wszystkich indeksowanych kolumnach, ale wyłącznie o kluczach wierszy, które znajdują się na danym węźle. Oznacza to, że jeżeli użytkownicy **jturek** oraz **manisero** znajdują się na różnych węzłach, również indeks jest równomiernie dystrybuowany pomiędzy te dwa węzły.

Lokalne indeksowanie wymusza odwołania do wszystkich węzłów dla każdego wyszukiwania. Zakładając, że indeksowanych wierszy (zarówno wszystkich, jak i tych z wyniku zapytania) jest dużo więcej niż węzłów, nadal znacząco przyspiesza to wyszukiwanie. Sytuacja zmienia się, gdy indeksowana jest wartość unikalna. Wtedy wbudowany w Cassandrę mechanizm wymusza wykonanie wielu zapytań zamiast jednego.

Indeks wartości unikalnych można zamodelować samodzielnie w prosty sposób. [33] Wystarczy utworzyć nową tabelę, w której unikalna wartość jest identyfikatorem. W wierszu dla danej unikalnej wartości zapisany jest odpowiedni identyfikator. Przykład takiego indeksu dla adresu e-mail (kolumna `email`) użytkowników przedstawiono na diagramie 4.29.

OMC wspiera automatyczne tworzenie indeksów dla wartości unikalnych. Na listingu 4.30 przedstawiono sposób utworzenia indeksu dla wartości unikalnych dla pola `email`.

Flaga `searchable_unique=True` powoduje, że dla danego pola tworzony jest indeks unikalny. Indeks jest w całości zarządzany przez OMC. Podczas wywołania `User.objects().find(email='J.Turek@pw.edu.pl')` me-

UserEmailIndex		
		jturek
	J.Turek@stud.elka.pw.edu.pl	null
		manisero
	M.Aniserowicz@stud.elka.pw.edu.pl	null

Rysunek 4.29: Przykład tabeli przechowującej indeks dla unikalnych wartości (adres e-mail użytkownika).

```
class User(Model):
    id = UuidField(type=TimeUuid, auto_generate=True, partition_key=True)
    name = TextField()
    surname = TextField()
    email = TextField(searchable_unique=True)
    city = TextField(searchable=True)
```

Rysunek 4.30: Modelowanie tabeli użytkownicy z wykorzystaniem indeksu dla wartości unikalnych.

chanizm wyszukuje identyfikator użytkownika w stworzonym przez siebie indeksie. OMC aktualizuje indeksy podczas wykonywania operacji `save()` oraz `delete()`.

Zarówno indeks wartości unikatowych, jak również domyślny indeks drugiego stopnia (tworzony przy użyciu flagi `searchable=True`) nazywane są przy użyciu tej samej konwencji. Pierwszą składową jest nazwa encji, drugą - nazwa pola, a trzecia to słowo „index". Dla przykładu 4.30 indeks wartości unikatowych zostanie umieszczony w tabeli o nazwie `user_email_index`.

4.7 Przetwarzanie partiami

Apache Cassandra zapewnia wsparcie dla operacji atomowych. [34] Wykorzystując partie[14] można wykonać zestaw operacji, z których (cytując dokumentację) „jeżeli jedna się powiedzie, to wykonają się wszystkie". Nie jest to jednak odpowiednik transakcji znanych z relacyjnych baz danych. Apache

[14]Partia (ang. batch) - zgrupowany zestaw zapytań INSERT, UPDATE lub DELETE.

Cassandra nie zapewnia integralności wykonywanych w partii zapytań: dane mogą być modyfikowane równolegle przez inne zapytania. Przykład atomowych operacji został zaprezentowany na diagramie 4.31. Jest to zestaw dwóch zapytań, które usuwają z bazy danych użytkownika i właściwy mu indeks dla pola e-mail.

```
BEGIN BATCH
  DELETE FROM user WHERE userId = 'jturek'
  DELETE FROM user_mail_index WHERE mail = 'J.Turek@pw.edu.pl'
APPLY BATCH;
```

Rysunek 4.31: Przykład operacji na partii, która usuwa z bazy danych użytkownika *jturek* i jego indeks dla pola email.

Silnik wykorzystywany w OCM zapewnia wsparcie dla przetwarzania partiami. Potrafi kolejkować wszystkie zapytania DML[15], a następnie wykonywać zapytania z kolejki jako operacja atomowa. Przykład wykorzystania operacji na partiach w OCM zaprezentowano na listingu 4.32.

```
# starts batch
Model.begin_batch()
# create new Item
item = Item(name='item', description='desc', price=12.5)
item.save()    # save is queued
wishlist = Wishlist(userId='jturek', itemId=item.id)
wishlist.save()    # save is queued
# executes item.save() and wishlist.save()
Model.apply_batch()
```

Rysunek 4.32: Przykład wykonania operacji na partii, która wstawia do bazy nowy przedmiot i dodaje go do listy życzeń użytkownika.

Wywołanie metody `Model.begin_batch()` powoduje rozpoczęcie kolejkowania zapytań DML. Operacja wstawienia nowego przedmiotu (`item.save()`)

[15]Data Modification Language (ang. język modyfikacji danych) - podzbiór zapytań obejmujących wstawianie (INSERT), aktualizację (UPDATE) oraz usuwanie (DELETE) danych.

nie wykonuje się od razu, ale jest umieszczana w kolejce. Do kolejki trafia również operacja aktualizacji listy życzeń użytkownika (`wishlist.save()`). Wywołanie metody `Model.apply_batch()` powoduje, że partia dwóch zgrupowanych zapytań wykonuje się.

4.8 Wsparcie dla liczników

Apache Cassandra udostępnia wsparcie dla specjalnego typu danych - liczników. [35] Liczniki to kolumny specjalnego typu `counter`, na których można wykonywać tylko dwie operacje: inkrementacja lub dekrementacja. Poza specjalnym typem zmienia się również sposób interakcji z tabelą. Nie można wstawiać do niej wartości, a jedynie aktualizować liczniki dla różnych kombinacji pozostałych pól.

OMC wspiera modelowanie liczników. Przykład przedstawiono na listingu 4.33.

```
class TripCounter(Model):
    country = TextField(partitioning_key=True)
    visits = CounterField()
sweden = TripCounter(country='Sweden')
sweden.visits.increment(1)
sweden.save()
poland = TripCounter(country='Poland')
poland.visits.increment(5)
poland.visits.decrement(2)
poland.save()
s_visits = TripCounter.find(country='Sweden').get().visits    # 1
p_visits = TripCounter.find(country='Poland').get().visits    # 3
```

Rysunek 4.33: Przykład użycia liczników w OCM.

Obsługa modelu posiadającego licznik różni się od wykorzystania standardowego modelu. Przede wszystkim wartości wszystkich pól są ustawione w trybie `read only`, czyli nie jest możliwe wykonanie operacji podstawienia `sweden.country = 'Germany'`. Dzięki temu wszystkie operacje wyko-

nywane na liczniku są zawsze odpowiednio interpretowane, gdyż nie może się zmienić ich kontekst. W przeciwnym razie trudno byłoby ocenić intencje programisty, który najpierw inkrementuje licznik o 2, a następnie zmienia wartość kraju dla obiektu ze zwiększonym licznikiem. Mogło być to działanie celowe (operacje wykonane w nieintuicyjnej kolejności) lub zwykły błąd.

Pole typu `CounterField` udostępnia dwie operacje: inkrementację wartości licznika (`increment()`) oraz dekrementację (`decrement()`). Metody wywołane bez podania parametru standardowo zwiększają/zmniejszają wartość o 1. Obie funkcje są łączne. Oznacza to, że tak jak w przykładzie można dokonywać wielu operacji na jednym liczniku, które przed uaktualnieniem obiektu w bazie danych są sumowane.

Po wykryciu pola licznikowego w definicji modelu OCM automatycznie zmienia sposób obsługi obiektu. Nowo powstały obiekt jest traktowany tak, jakby został wcześniej pobrany z bazy danych - operacja `save()` dokonuje aktualizacji, a nie wstawienia.

4.9 Migracje pomiędzy wersjami modelu

Typowym problemem związanym z wykorzystaniem baz danych jest aktualizacja schematu. Dotyczy on również narzędzi do mapowania obiektowego. Problem ten pojawia się, gdy udało się wdrożyć pewien schemat i istnieją już dane na nim operujące, a wymagana jest modyfikacja modelu. Typowe scenariusze obejmują między innymi dodawanie kolumn do istniejących tabel, dodawanie nowych modeli oraz usuwanie istniejących tabel.

Modyfikacje wprowadzane do działającego modelu zazwyczaj nie są skomplikowane i łatwo wprowadzić je ręcznie. Dla przedstawionej propozycji mapowania obiektowego dla Cassandry można jednak znaleźć przypadki, których automatyzacja zaoszczędzi dużo pracy. Niech w modelu danych będzie wykorzystane pole typu `DenormalizedField`. Jeżeli wymagane będzie dodanie nowej zdenormalizanej danej należy:

1. Rozszerzyć model danych o nową kolumnę.

2. Dla każdego połączenia pobrać daną źródłową i przepisać ją do zdenormalizowanego pola.

Dzięki wbudowanemu wsparciu dla migracji OMC pozwala wykonywać takie operacje automatycznie. Dedykowany mechanizm pozwala tworzyć punkty migracyjne, czyli kolejne wersje modelu danych. Potrafi również odczytywać wersję modelu na podstawie schematu bazy danych i aktualizować ją do najnowszej (lub wybranej) rewizji.

Mechanizm migracji w OMC jest oparty na najnowszej wersji projektu Django. [36] Podstawową jednostką migracji jest klasa `Migration`. Definiuje ona dwa pola:

zależności czyli skrypty migracyjne, które muszą zostać wykonane przed uruchomieniem operacji,

operacje czyli zmiany w modelu, które należy wykonać aby zaktualizować model danych.

Niech dany będzie model użytkownika przedstawiony na listingu 4.34. W pierwszej wersji modelu identyfikator jest typu `text`. Poza tym użytkownik posiada jeszcze imię typu tekstowego. W drugiej wersji modelu dodano pole tekstowe nazwisko oraz zmieniono typ identyfikatora na `timeuuid`.

Pierwszym krokiem jest stworzenie początkowej migracji. Odpowiada za to metoda `Model.migrations().make(initial=True)`. Wynikiem będzie powstanie nowego pliku `migration_initial.py`. Zakładając, że polecenie zostało wywołane dla pierwszej wersji modelu, zawartość tego pliku jest przedstawiona na listingu 4.35.

Po utworzeniu migracji początkowej i zmianie modelu danych należy utworzyć kolejny punkt migracji. Wywołanie metody `make()` utworzy plik `migration_0001.py`, którego treść jest przedstawiona na listingu 4.36. Aby zaktualizować bazę danych do nowego schematu należy uruchomić polecenie `Model.migrations().migrate()`. OMC spróbuje wykonać migrację, ale nie uda się to. Apache Cassandra nie potrafi zrzutować wartości pól tekstowych na typ `timeuuid`.

```
class User(Model):
    id = TextField(partitioning_key=True)
    # id = UuidField(type=TimeUuid, partitioning_key=True)
    name = TextField()
    # surname = TextField()
```

Rysunek 4.34: Model użytkownika. Linie poprzedzone znakiem # oznaczają zmiany w kolejnej wersji modelu.

```
class MigrationDescription(Migration):
    dependencies = ['__first__']
    operations = [
        Migration.add_model(name='User',
          fields = [
            ('id', TextField(partitioning_key=True)),
            ('name', TextField())
          ])
    ]
```

Rysunek 4.35: Migracja początkowa. Utworzenie modelu danych użytkownika.

```
class MigrationDescription(Migration):
    dependencies = ['initial']
    operations = [
        Migration.alter_field(model_name='User',
            name='id',
            field=UuidField(type=TimeUuid, partitioning_key=True)),
        Migrations.add_field(model_name='User',
            name='surname',
            field=TextField())
    ]
```

Rysunek 4.36: Pierwsza migracja. Do modelu dodano nowe pola oraz zmieniono typ identyfikatora.

Aby migracja zadziałała poprawnie należy nieco zmodyfikować kod z listingu 4.36:

- Zastąpić wpis `alter_field` przez `rename_field`, aby zmienić nazwę kolumny na tymczasową (przykładowo: `temp_id`).
- Dodać wpis `add_field`, który utworzy nowe pole odpowiadające identyfikatorowi typu `timeuuid`.
- Dodać wpis `run_method`, a następnie zaimplementować metodę, która zrzutuje wartości z poprzedniej kolumny na nową.
- Dodać wpis `delete_field`, który usunie pole tymczasowe `temp_id`.

4.9.1 Wydajność migracji danych

Migracja danych wymaga wykonania wielu sekwencyjnych zapytań do bazy danych. Nie jest to rozwiązanie efektywne czasowo. W celu zbadania wydajności mechanizmu migracji Autor posłużył się modelem przedstawionym na listingu 4.37.

```
class User(Model):
    id = TextField(key=True)
    name = TextField()
    item = DenormalizedField(
        relates=Item,
        fields=['name', 'price'])
```

```
class Item(Model):
    id = TextField(key=True)
    name = TextField()
    price = TextField()
    # color = TextField(
     default='Black')
```

Rysunek 4.37: Definicja modelu danych do testów mechanizmu migracji.

Model składa się z dwóch encji - przedmiotu i użytkownika, który przez zdenormalizowane pola jest związany z przedmiotem. W drugiej wersji modelu do encji `Item` zostało dodane pole `color`. Dodatkowo pole to zostało włączone do zdenormalizowanych pól modelu `User`. W skrypcie migracyjnym ustawiono, że dla istniejących przedmiotów pole `color` ma być uzupełniane losową wartością ze zbioru: czerwony, zielony lub niebieski. Dla tak zdefiniowanego problemu w trakcie migracji muszą zostać wykonane następujące czynności:

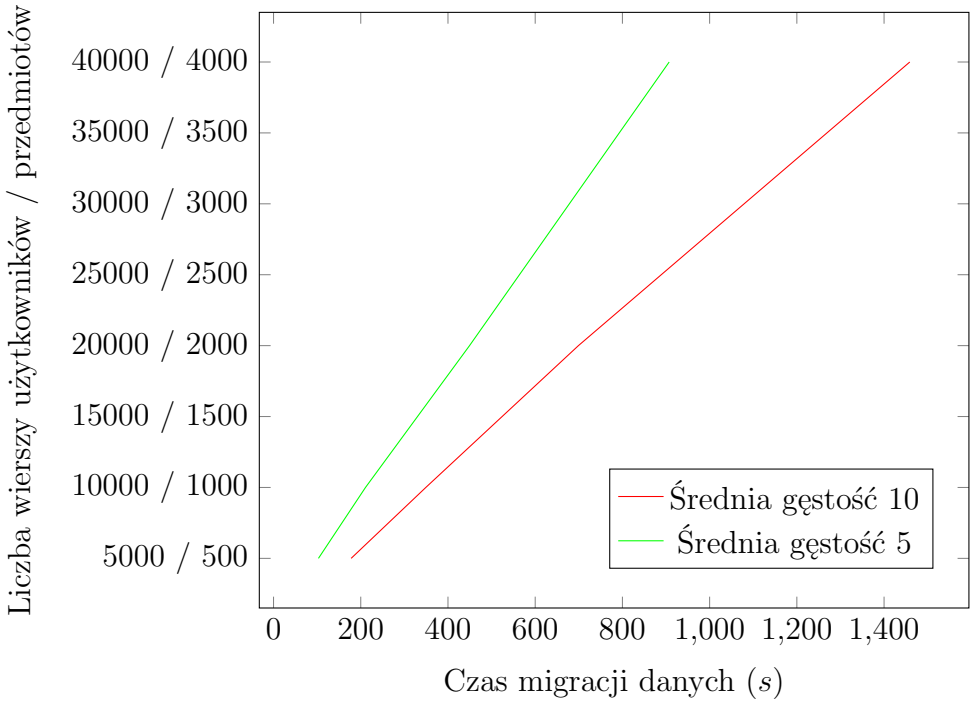
1. Do tabeli `item` dodawana jest kolumna `color` typu `text`.

2. Wybierane są wszystkie przedmioty. Dla każdego przedmiotu aktualizowana jest wartość pola `color`.

3. Do tabeli `user` dodawana jest kolumna `item_color` typu `text`.

4. Dla każdego użytkownika pobierane są wszystkie przedmioty. Dla każdego przedmiotu aktualizowana jest wartość zdenormalizowanego pola `item_color`.

Badanie wydajności migracji zostało przeprowadzone na środowisku lokalnym. Zmianie podlegała liczba użytkowników i przedmiotów. Testy przeprowadzone były dla dwóch średnich gęstości zależności: 5 oraz 10. Wyniki badania zostały przedstawione na wykresie 4.38.

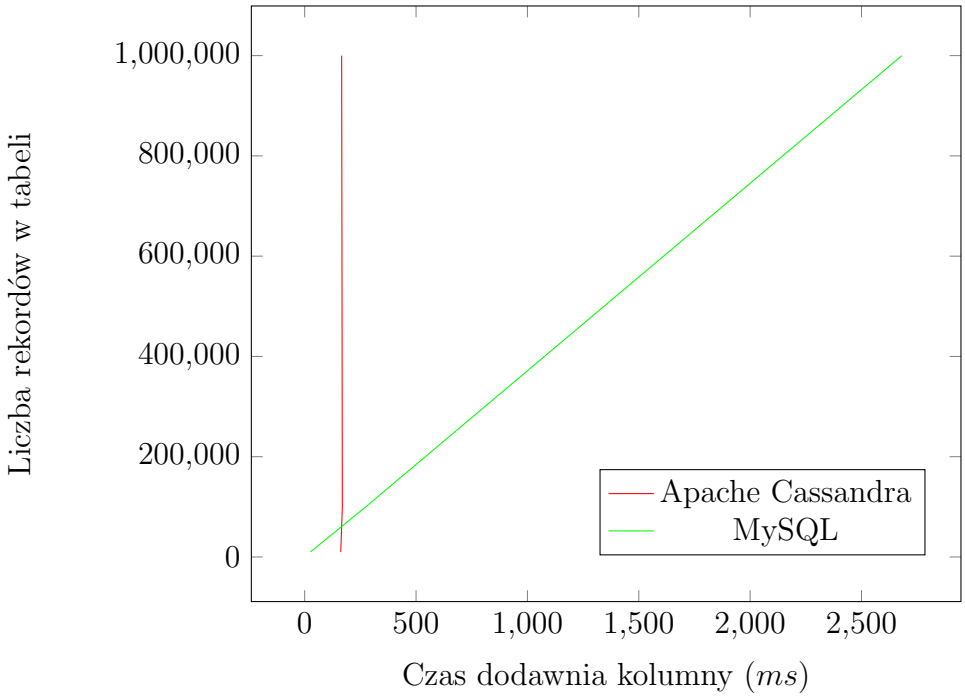


Rysunek 4.38: Porównanie czasów migracji danych dla różnej ilości rekordów.

Z wykresu widać, że migracja danych denormalizacyjnych jest procesem czasochłonnym. Przetworzenie pojedynczej zmiany, dla relatywnie niewielu

wierszy (< 500000), zajmuje 23 minuty. Z tego względu zdecydowanie odradza się wykonywanie tego typu migracji w trakcie produkcyjnego działania systemu.

Patrząc na wyniki przedstawione na wykresie 4.38 należy pamiętać, aby nie utożsamiać ich z wydajnością bazy danych dla operacji przekształcających schemat danych. Ze względu na charakterystykę modelu, Apache Cassandra dużo szybciej niż relacyjne bazy danych dokonuje przekształceń takich jak dodawnie lub usuwanie kolumn. Na wykresie 4.39 przedstawiono porównanie wydajności operacji `ALTER TABLE ... ADD ...` w bazach MySQL oraz Cassandra.



Rysunek 4.39: Porównanie czasów dodawania kolumny do istniejącej tabeli dla bazy relacyjnej oraz Cassandry.

Na podstawie wykresu można zauważyć, że czas dodawania kolumny do istniejącej tabeli w Cassandrze jest niezależny od liczby wierszy. Inaczej jest w przypadku relacyjnej bazy danych, gdzie czas dodawania kolumny rośnie liniowo wraz z ilością rekordów w tabeli. Z tego względu proste migracje

danych mogą być wykonywane podczas produkcyjnego działania systemu.

4.9.2 Potencjalne problemy migracji

Mechanizm migracji dla Cassandry posiada pewne wady w stosunku do analogicznych implementacji dla relacyjnych baz danych. [36] Kluczowy problem jest związany z brakiem możliwości niezawodnego pobrania wszystkich rekordów z danej tabeli. Pomimo, że w najnowszej wersji sterownika dla języka Python został wprowadzony mechanizm paginacji, [37] nadal może wydarzyć się sytuacja przekroczenia czasu żądania podczas pobierania wierszy. OCM używa paginacji o rozmiarze strony równym 10000 rekordów. W zależności od konfiguracji klastra może być to zbyt wysokie ustawienie. Z drugiej strony zmniejszenie rozmiaru paginacji ma zły wpływ na wydajność mechanizmu.

Drugi problem wiąże się z brakiem transakcyjności Apache Cassandry. Jeżeli migracja nie powiedzie się, podczas gdy którakolwiek ze zmian została wprowadzona do modelu danych, ponowne uruchomienie mechanizmu nie zadziała. Z tego względu zaleca się ostrożne wykorzystanie migracji i poprzedzanie każdej takiej operacji wykonaniem kopii zapasowej danych.

4.10 Profilowanie i testowanie modelu danych

OCM posiada zintegrowane narzędzia umożliwiające profilowanie i testowanie wydajności modelu danych. Najbardziej podstawowym narzędziem jest logowanie zapytań. Pozwala ono przechwytywać wszystkie odwołania do bazy i zapisywać je do pliku wraz z czasem wykonania. Dodatkowo użytkownik OMC może zażądać żeby zapisywane były tylko i wyłącznie zapytania, których wykonanie jest dłuższe niż zadany czas. Przykład uruchomienia takiej funkcjonalności został przedstawiony na listingu 4.40.

```
Model.engine.enable_logging(file='log.txt', longer_than=1000)
```

Rysunek 4.40: Wypisywanie do pliku log.txt wszystkich zapytań, których czas wykonania przekracza 1000 milisekund.

Możliwe jest również wypisywanie najwolniejszych zapytań. W trybie śledzenia niewydajnych zapytań OMC zapisuje zadaną liczbę odwołań do bazy o najdłuższym czasie wykonania, posortowanych zaczynając od najwolniejszego. Przykład wykorzystania tej funkcjonalności został przedstawiony na listingu 4.41.

```
Model.engine.trace_slow_queries(amount=10)
# perform ORM database operations
Model.engine.output_slow_queries(file='slow_queries.txt')
```

Rysunek 4.41: Wypisywanie do pliku slow_queries.txt 10 najwolniejszych odniesień do bazy danych.

4.10.1 Zasilanie danymi

Nie da się zmierzyć wydajności modelu danych przy pustej bazie, stąd OMC udostępnia dwa narzędzia ułatwiające populację testowymi danymi:

Eksport i import Pozwala na przenoszenie danych pomiędzy instancjami aplikacji. Przykładowym scenariuszem jest przeniesienie danych z serwera produkcyjnego na środowisko testowe w celu optymalizacji modelu.

Generacja Wygenerowanie testowego zestawu danych na podstawie automatycznych lub zdefiniowanych przez użytkownika algorytmów.

Eksport i import wykorzystuje natywny mechanizm języka CQL - polecenia `COPY TO` oraz `COPY FROM`. Służą one do zapisywania danych z Cassandry w formacie CSV[16].

Mechanizm eksportu w OMC wywołuje polecenie `COPY TO` dla wszystkich tabel, a następnie łączy wyniki w jeden plik, w którym definicje tabel znajdują się w wierszach rozpoczynających się znakiem komentarza `#`. Mechanizm

[16]Comma Separated Values (ang. „wartości oddzielone przecinkami") - format pliku, w którym każda linia oznacza wiersz danych, a wartości poszczególnych kolumn są rozdzielone przecinkami.

importu rozdziela pliki, a następnie na każdym z nich wywołuje zapytanie `COPY FROM`. Aby mechanizm działał poprawnie obie instancje Cassandry muszą posiadać już istniejący, spójny model danych. Przykładowe wywyołanie zostało przedstawione na listingu 4.42.

```
# first Cassandra instance
Model.engine.export_to_csv(name='data_export')
# second Cassandra instance
Model.engine.import_from_csv(name='data_export')
```

Rysunek 4.42: Przykład eksportowania i importowania danych do/z plików CSV.

Używając mechanizmu eksportu należy pamiętać, że obecna implementacja poleceń `COPY ...` jest bardzo niedopracowana. [38] Problem stanowi wykorzystanie zapytania `SELECT * FROM table WITH LIMIT 99999999`, które służy do pobierania danych. Cechą charakterystyczną dla Cassandry jest ustawiany w konfiguracji węzłów parametr `read_request_timeout_in_ms`, który standardowo ma zbyt restrykcyjną wartość, aby powyższe polecenie mogło wykonać się dla jakichkolwiek danych. Przed uruchomieniem eksportu należy odpowiednio dostosować konfigurację bazy danych.

Niestety, mimo zmiany konfiguracji dla ogromnych zbiorów danych nadal mogą wystąpić przekroczenia czasu żądania. Z tego względu funkcja `export_to_csv` ma zdefiniowany parametr `excludes`, który przyjmuje listę nazw modeli wyłączonych z eksportu. W nowej wersji CQL problemy te mają zostać rozwiązane dzięki wbudowanym w język mechanizmom paginacji.

Mechanizm generacji danych testowych umożliwia definiowanie reguł, według których zostaną stworzone dane. Dzięki temu użytkownik może w łatwy sposób przygotować skrypty zasilające bazę danych w sposób odwzorowujący realne wymagania: długość i objętość danych, gęstość i charakterystykę relacji i tym podobne.

Na listingu 4.43 przedstawiono sposób uruchamiania generacji danych testowych. Parametrem funkcji `populate_test_data` jest słownik, który definiuje nazwy modelu i ilość przypadających mu wierszy. W przypadku po-

minięcia modelu w słowniku dane dla niego nie zostaną wygenerowane.

```
Model.engine.populate_test_data({'User': 10000, 'Address': 1000})
```

Rysunek 4.43: Uruchamianie generacji danych testowych.

Domyślnie używany jest naiwny algorytm generacji danych testowych. Pola zastępowane są wartościami domyślnymi. Przykładowo pola tekstowe uzupełniane są frazą `lorem ipsum`, pola typu `timestamp` lub `timeuuid` wypełniane są czasem obecnym, natomiast zależności generowane są przy założeniu, że istnieje 0.5% szansy na sformułowanie związku.

Dane wygenerowane przy naiwnym podejściu nie nadają się dobrze do testów wydajnościowych. Dlatego użytkownik OMC ma możliwość zdefiniowania własnych algorytmów generacji danych testowych. Wymaga to zdefiniowania klasy wewnętrznej modelu o nazwie `Generator`, a w niej metody `generate()`. Przykład wykorzystania generatora przedstawia listing 4.44.

```
class User(Model):
    username = TextField(partitioning_key=True)
    age = IntegerField()
    class Generator(object):
        def generate(item_no):
            username = 'User: ' + str(item_no)
            age = random.randrange(18, 55)
```

Rysunek 4.44: Przykład wykorzystania generatora danych testowych.

Zarówno generatory, jak również mechanizm importu/eksportu posiadają wiele zastosowań, które nie muszą być ograniczone do badania wydajności. Przykładowe wykorzystania to:

- Migracja danych z innego systemu.
- Synchronizacja danych pomiędzy różnymi środowiskami.
- Zarządzanie kopiami zapasowymi danych.

Rozdział 5

Studium przypadku

W rozdziale Autor przedstawia proces modelowania danych w mechanizmie OMC. Przebieg procesu jest omówiony na przykładzie istniejących modeli danych, które są obsługiwane przy pomocy CQL. Dzięki takiemu podejściu łatwo jest wskazać korzyści, które zapewnia użytkownikowi stosowanie OMC.

5.1 Twissandra

W sekcji 2.7.1 został szczegółowo omówiony projekt Twissandra. Przedstawia on kompletny model danych w języku CQL dla serwisu o funkcjonalności analogicznej do platformy Twitter. Autor pracy przedstawi proces tworzenia analogicznego schematu w OMC z rozbiciem na poszczególne encje danych.

5.1.1 Użytkownik

Encja użytkownika posiada tylko dwie właściwości - nazwę oraz hasło. Nawet dla tak prostego przypadku mechanizm OMC może wprowadzić znaczne ułatwienie dla użytkownika. W zaprezentowanym w sekcji 2.7.1 przykładzie hasło było przechowywane jako otwarty tekst. W docelowym systemie takie rozwiązanie byłoby nieakceptowalne ze względów bezpieczeństwa.

Zgodnie z obecnymi standardami bezpieczeństwa przechowywanie hasła w bazie danych wymaga wykonania następujących operacji: [39]

1. Przechowywanie w bazie danych wartości funkcji skrótów haseł. Rekomendowanym algorytmem do obliczania funkcji skrótu jest `SHA-256`.

2. Wykorzystanie „soli". Polega to na łączeniu losowego ciągu znaków (o długości 16 bitów lub większej) z hasłem. Na podstawie tak uzyskanego ciągu obliczana jest funkcja skrótu. Losowe znaki, które służą jako „sól" są zapisywane, obok skrótu, w bazie danych. Dzięki temu:

   - Dwaj użytkownicy z identycznym hasłem (z dużym prawdopodobieństwem zależnym od jakości losowania „soli") w bazie danych będą mieli różne funkcje skrótu.
   - Zwiększa się liczba kombinatorycznych możliwości to sprawdzenia w przypadku ataków typu brute force[1]. Każda możliwość musi zostać przetestowana dla dowolnej „soli" występującej w danych.

3. Wykorzystanie iteracyjnego algorytmu PBKDF2 opisanego w standardzie RFC 2898. [40] Pozwala on skalować czas niezbędny do wyznaczenia funkcji skrótu dla jednego hasła poprzez zwiększanie liczby wymaganych iteracji obliczeń. Dzięki temu zwiększa się czas niezbędny do zgadnięcia klucza użytkowników, którzy nie stosują kryptograficznie bezpiecznych haseł.

W przypadku modelowania encji użytkownika z wykorzystaniem języka CQL odpowiedzialność za bezpieczeństwo przechowywania haseł spada na programistę aplikacji. Mechanizm OMC posiada dedykowany do przechowywania kluczy typ pola `PasswordField`. Pozwala on wybrać algorytm obliczania funkcji skrótu, potrafi automatycznie dodawać „sól", a także wyspecyfikować liczbę iteracji PBKDF2. W kolumnach bazy danych, oprócz skrótu i „soli" zapisywana jest także liczba iteracji. W przyszłości informacja ta może zostać wykorzystania do zwiększenia bezpieczeństwa haseł.

[1]Brute force (ang. brutalna siła) - atak polegający na sprawdzaniu wszystkich możliwych haseł aż do momentu znalezienia poprawnego. [41]

Pole `PasswordField` charakteryzuje się tym, że przeliczenia dokonywane są w trakcie ustawiania wartości. Oznacza to, że w pamięci przechowywane jest wyłącznie wynikowa wartość funkcji skrótu. Wartość hasła może być jedynie ustawiona lub porównana - sprawdzenie zgodności kluczy dokonywane jest za pomocą metody `check`.

Definicja modelu użytkownika w OMC została zaprezentowana na diagramie 5.1. W schemacie bazodanowym pole `username` zostanie zmapowane na kolumnę tekstową, natomiast pole `password` będzie automatycznie obsługiwane przez trzy kolumny - `password` do przechowywania wartości skrótu klucza, `password_salt` z „solą" oraz `password_iterations`, w którym przechowywana jest siła algorytmu PBKDF2. Widać tutaj istotną różnicę w stosunku do tradycyjnych systemów mapowania obiektowo-relacyjnego. Definicja modelu nie skupia się na fizycznej strukturze danych, ale na rodzaju przechowywanych informacji (w tym wypadku klucza).

```
class User(Model):
    username = TextField(partition_key=True)
    password = PasswordField(algorithm='SHA256',
                             salt=True,
                             iterations=10000)
```

Rysunek 5.1: Użytkownik Twissandry zamodelowany w OMC.

Do modelu można odwoływać się przy pomocy zarządcy obiektów (pozwala on wyszukiwać wpisy w bazie danych) lub bezpośrednio do instancji. Przykładem wykorzystania zarządcy jest pobranie użytkownika o pseudonimie `jturek` zaprezentowane w pierwszej linii listingu 5.2. W drugiej linii widać odwołanie bezpośrednio do instancji, które sprawdza czy `UnsafePassword` jest hasłem znalezionego użytkownika.

```
user = User.objects().get('jturek')
user.password.check('UnsafePassword') # false
```

Rysunek 5.2: Przykłady odwołania do encji użytkownika.

5.1.2 Śledzeni użytkownicy

Fakt śledzenia użytkownika odnotowywany jest poprzez model `Followers`. Przechowuje on relacje pomiędzy dwiema osobami wraz z datą utworzenia zależności. Propozycja modelu została przedstawiona na listingu 5.3.

```
class Followers(Model):
    user = RelatedField(relates=User,
                        partition_key=True)
    follower = RelatedField(relates=User,
                            clustering_key=True,
                            searchable=True)
    since = TimestampField(auto_add_now=True)
```

Rysunek 5.3: Śledzeni użytkownicy w Twissandrze zamodelowani w OMC.

Model `Followers` wykorzystuje prosty typ zależności - `RelatedField`. Przenosi on składowe klucza ze wskazywanej do danej encji. Dodatkowo umożliwia operowanie zarówno na obiektach, jak również na prostych wartościach. Na listingu 5.4 zaprezentowano dwa sposoby połączenia obiektu `Followers` z tym samym użytkownikiem. Pierwszy z nich polega na podstawieniu do pola instancji powiązanego obiektu. Drugi pozwala bezpośrednie ustawienie komponentu klucza jako wartości. Dla pola o nazwie `field` i komponentu klucza o nazwie `component` podstawienie to może być wykonane przez odwołanie do pola `field_component`. Mechanizm ten oferuje wsparcie dla złożonych kluczy.

```
follower_one = Followers()
follower_one.user = User.objects().get('jturek')
follower_two = Followers()
follower_two.user_username = jturek'
```

Rysunek 5.4: Ustawianie wartości pola RelatedField przez obiekt powiązany lub jego klucz.

Podczas pobierania instancji modelu `Followers` z bazy danych domyślnie uzupełniane są wyłącznie wartości kluczy. Na żądanie użytkownik może ściągnąć obiekt wraz z elementami powiązanymi. Zostało to zaprezentowane na listingu 5.5. Pobierając obiekt `follower_one` bez użycia metody `related()` odwołanie do pól encji zależnej powoduje wyjątek aplikacji. W przypadku instancji `follower_two` wszystkie właściwości powiązane są dostępne do odczytu.

```
follower_one = Followers.objects().find(user='jturek',
                  follower='manisero')
follower_one.user.password.check('Pass1') # exception
follower_two = Followers.objects().find(user='jturek',
                  follower='manisero').related()
follower_two.user.password.check('Pass2') # ok
```

Rysunek 5.5: Pobieranie instancji obiektu wraz z elementami powiązanymi.

W trakcie opisywania modelu danych Twissandry Autor zauważył, że dodanie indeksu do tabeli `followers` na kolumnie `follower` umożliwia odwrócenie zależności. Z wykorzystaniem OMC jest to jeszcze prostsze - wystarczy dla zadanego pola ustawić flagę `searchable`. Mechanizm sam utworzy odpowiedni indeks i umożliwi wyszukiwanie po wartościach wskazanej kolumny. Na listingu 5.6 przedstawiono odwołanie do metod interfejsu, które umożliwia odnalezienie wszystkich osób śledzonych przez użytkownika `jturek`.

```
Followers.objects().find(follower='jturek')
```

Rysunek 5.6: Wyszukiwanie wszystkich osób śledzonych przez *jturek*.

Wykorzystując OMC do modelowania tabeli `Followers` użytkownik otrzymuje jeszcze jedno udogodnienie. Pole z odciskiem czasu o nazwie `since` jest oznaczone parametrem `auto_add_now`. Dzięki temu przy dodawaniu nowego wpisu jego wartość, jeżeli nie została wyspecyfikowana jawnie, zostanie wypełniona obecną datą i czasem. Wprawdzie język CQL pozwala w prosty sposób osiągnąć ten sam efekt (za pomocą funkcji `dateOf(now())`), ale

trzeba o tym pamiętać w każdym fragmencie aplikacji. Posiadając poprawnie zaprojektowany model w OMC użytkownik może zaniedbać istnienie pola aż do momentu odczytywania jego wartości.

5.1.3 Wpisy

Przedstawiony w sekcji 2.7.1 schemat tabeli dla wpisów posiada trzy właściwości: identyfikator, nazwę użytkownika oraz treść. Jest to prosty model, który wykorzystuje jedynie mechanizmy omówione dotychczas. Został on zaprezentowany na listingu 5.7.

```
class Tweet(Model):
    __manager__ = TweetManager
    id = UuidField(partition_key=True,
                   auto_generate=True)
    user = RelatedField(relates=User)
    body = TextField()
```

Rysunek 5.7: Wpis w Twissandrze zamodelowany w OMC.

W przypadku dodawania nowych wpisów warto rozważyć problem osi czasu. Z analizy modelu Twissandry wiadomo, że wstawienie nowego wiersza do tabeli `tweets` implikuje konieczność uzupełnienia powiązanych rekordów w tabeli `timeline`. OMC pozwala przeciążać obiekty zarządzające encjami. Dzięki temu możliwe jest zmodyfikowanie metody `save()`, która zapisuje instancję w bazie danych w taki sposób, aby oś czasu była wypełniana automatycznie.

Pole `__manager__` klasy `Model` wskazuje na typ obiektu zarządzający danymi tego modelu. Dzięki temu możliwe jest zastąpienie standardowej implementacji elementem, który posiada dedykowaną logikę. Podczas zapisywania nowego wiersza zarządca wpisów `TweetManager`:

1. Odszuka wszystkich użytkowników śledzących autora wpisu.
2. Dla każdego ze śledzących utworzy wpis na osi czasu.

3. Utworzy wpis na osi czasu autora wpisu.

Implementację według powyższego algorytmu przedstawia listing 5.8.

```
class TweetManager(ObjectManager):
    def save(entity):
        super(ObjectManager, self).save(entity)
        followers = Followers.objects().find(
                        user=entity.user_username)
        for follower in followers:
            timeline = Timeline(user=follower.username,
                                tweet=entity)
            timeline.save()
        user_timeline = Timeline(user=entity.user_username,
                                 tweet=entity)
        user_timeline.save()
```

Rysunek 5.8: Automatyczne zapisywanie wpisu do osi czasu użytkownika oraz śledzących go osób.

Możliwość włączenia uniwersalnych metod przetwarzania danych bezpośrednio do funkcji obsługujących model jest korzystna. Sposób obsługi przechowywanych informacji stanowi o ich charakterystyce. Pojęciowo jest on integralną częścią modelu. Korzyści są widoczne zwłaszcza w ramach rozwoju systemu. Jeżeli zajdzie konieczność dodania niestandardowej logiki biznesowej obsługującej encje zagwarantowane jest, że modyfikacje wystarczy wprowadzić w jednym miejscu.

Klasa `ObjectManager` posiada metodę, która umożliwia modyfikację instancji modelu bezpośrednio przed zapisem. Jej działanie zostanie zaprezentowane dla następujących prawdopodobnych założeń dla serwisu Twitter:

- W systemie wpisy dodawane są tylko w kontekście aktualnie zalogowanego użytkownika.

- Istnieje obiekt `LoginManager`, który zwraca informację o aktualnie zalogowanym użytkowniku.

Przyjmując powyższe założenia można zmodyfikować zarządcę wpisów w taki sposób, aby w polu `user` zawsze zapisywana była nazwa aktualnie uwierzytelnionego użytkownika. Do tego celu nadpisano metodę wywoływaną przed każdym zapisem - `before_save()`. Jej użycie przedstawia listing 5.9.

```
class TweetManager(ObjectManager):
    def before_save(entity):
        super(ObjectManager, self).before_save(entity)
        entity.user = LoginManager.get_authenticated_user()
```

Rysunek 5.9: Automatyczne uzupełnianie pola *user* aktualnie uwierzytelnionym użytkownikiem.

5.1.4 Oś czasu

Oś czasu jest najbardziej skomplikowanym elementem modelu Twissandry. Jednocześnie ukazuje najwięcej korzyści wykorzystania mechanizmu OMC do modelowania dziedziny danych.

Oś czasu jest bezpośrednio związana z użytkownikiem. Do realizacji tego powiązania, podobnie jak w przypadku wpisów lub śledzonych osób, można wykorzystać pole typu `RelatedField`. Znacznie ciekawszym przypadkiem jest relacja osi czasu i wpisu. Zgodnie z wnioskami przedstawionymi w sekcji 2.7.1 do wydajnej realizacji tego powiązania można wykorzystać denormalizację.

Relizując denormalizację przy pomocy języka CQL nie ma możliwości wprowadzenia żadnego powiązania pomiędzy odpowiadającymi kolumnami. Dane traktowane są zupełnie niezależnie, a za zachowanie ich spójności odpowiada wyłącznie programista aplikacji. Mechanizm OMC realizuje logiczne powiązanie danych przy pomocy pola `DenormalizedField`. Świadczą o tym następujące możliwości:

- Wartość pola może być ustawiana na podstawie instancji obiektu zależnego (w tym wypadku `Tweet`).
- Istnieje możliwość sprawdzenia poprawności denormalizacji. Zapis obiektu `Timeline` za pomocą metody `save(validation=True)` nie powiedzie się jeżeli dane będą niezgodne z wskazywaną encją. Odpowiada to (rozbudowanemu o sprawdzanie kolumn niebędących kluczem) mechanizmowi klucza obcego w relacyjnych bazach danych.
- Mechanizm udostępnia interfejs do automatycznej aktualizacji encji zależnych. Przy modyfikacji obiektu klasy `Tweet` z wykorzystaniem metody `update(update_related=True)` zaktualizują się również odpowiadające temu wpisowi instancje osi czasu.

Do realizacji powiązania poprzez denormalizację wymagane jest podanie dwóch danych. W parametrze `relates` wskazywana jest klasa modelu zależnego, natomiast parametr `fields` to lista nazw pól, które mają zostać zdenormalizowane. Istotne jest, że klucz encji zależnej automatycznie dołączany jest do listy denormalizowanych pól. Na listingu 5.10 pokazana jest definicja modelu osi czasu, która wykorzystuje `DenormalizedField`.

```
class Timeline(Model):
    user = RelationField(relates=User,
                         partition_key=True)
    tweet_time = UuidField(type=TimeUuid,
                           auto_generate=True,
                           clustering_key=True,
                           order_descending=True,
                           partition_by_day=True)
    tweet = DenormalizedField(relates=Tweet,
                              fields=['body'])
```

Rysunek 5.10: Oś czasu w Twissandrze zamodelowana w OMC.

Oś czasu powinna być posortowana od najnowszych do najstarszych wpisów. Podobnie jak CQL, mechanizm OMC umożliwia sortowanie pól należących do klucza typu `clustering`. Służą do tego flagi `order_ascending` oraz `order_descending`, które układają dane odpowiednio w porządku rosnącym lub malejącym według danej wartości. Na listingu 5.10 zaprezentowano sortowanie malejące względem pola `tweet_time`, czyli czasu publikacji wpisu.

W sekcji 2.7.1 Autor zaproponował żeby oś czasu była partycjonowana po nazwie użytkownika oraz dacie wpisu (z dokładnością do dnia). Można wykorzystać do tego celu metodę `before_save()` omówioną przy okazji encji wpisu:

1. Do modelu `Timeline` dodać tekstowe pole `day` i dołączyć je do klucza typu partycjonującego.
2. W metodzie `before_save()` pobrać datę wpisu z pola `tweet_time`, sformatować ją do postaci ciągu znaków i wpisać do pola `day`.

Rozwiązanie to wprowadza pewną automatyzację. Podobnie jak w przypadku implementacji wykorzystującej czysty język CQL wymaga jednak napisania kodu źródłowego służącego do formatowania dat. Pamiętając, że OMC oferuje wsparcie dla wzorców projektowania można znacznie prościej zaprojektować model. Opisany w sekcji 4.6.1 szereg zdarzeń jest odpowiedzialny za partycjonowanie wpisów po komponentach daty. Wykorzystując wzorzec wystarczy dodać do pola `tweet_time` parametr konfiguracyjny `partition_by_day=True` i wszystkie opisane wcześniej czynności zostaną automatycznie wykonane przez OMC. Takie rozwiązanie zostało uwzględnione na listingu 5.10 pokazującym model osi czasu.

5.2 Wnioski

Studium przypadku dla projektu Twissandra pozwala wyciągnąć wnioski dotyczące mechanizmu modelowania obiektowego dla Cassandry. OMC jest narzędziem wspierającym tworzenie schematu danych. Może być ono zastosowa-

ne bez znajomości języka CQL i/lub podstawowej wiedzy na temat wewnętrznej reprezentacji informacji w Cassandrze. Jednakże podobnie jak w przypadku mapowania obiektowo-relacyjnego zbudowanie efektywnego modelu wymaga znajomości zagadnień związanych z bazą danych. Proces modelowania wymaga podejmowania decyzji analogicznych do definiowania schematu w języku CQL.

W odróżnieniu od CQL mechanizm OMC pozwala definiować dziedzinę danych na różnych poziomach abstrakcji. Tworzy on nie tylko fizyczny schemat, ale ponadto opisuje charakterystykę danych, zależności pomiędzy encjami, a także sposób ich przetwarzania. Dzięki dedykowanym mechanizmom dla wzorców projektowania użytkownik oszczędza dużo pracy. W OMC definiowany jest problem, natomiast jego rozwiązaniem zajmuje się narzędzie.

Filozofią modelowania obiektowego dla Cassandry jest budowanie efektywnych, logicznych schematów danych. Przykładowo dodając nowy element `Timeline` w OMC wystarczy podać dwie dane: instancję użytkownika `User` oraz wpisu `Tweet`. Odpowiada to logicznej strukturze danych. Oś czasu jest bowiem złączeniem użytkownika serwisu i widocznych dla niego wpisów. Dla kontrastu język CQL opisuje fizyczną strukturę. Dodając nowy element osi czasu należy manualnie uzupełnić wartości aż pięciu kolumn.

Rozdział 6

Podsumowanie

Tematem pracy było zbadanie możliwości przystosowania istniejących rozwiązań mapowania obiektowo-relacyjnego do bazy danych Apache Cassandra. Celem było zachowanie wysokiej wydajności oraz możliwości stosowania wzorców projektowania, tak aby umożliwić łatwą optymalizację. Już na wstępie pracy okazało się, że taka definicja celów jest wzajemnie sprzeczna. Przeprowadzone badania wykazały, że odwzorowanie relacyjnego modelu danych w Apache Cassandrze jest możliwe, jednakże niweluje większość zalet tego silnika bazodanowego. Uzyskane wyniki wydajności operowania na takim schemacie były porównywalne z relacyjnymi systemami baz danych.

Autor postanowił zrezygnować z jednego spośród wzajemnie sprzecznych założeń i odrzucił pełną zgodność z istniejącymi narzędziami do mapowania obiektowo-relacyjnego. Umożliwiło to znaczne rozszerzenie zakresu pracy. Zamiast prostego odwzorowania dziedziny danych w świecie obiektowym udało się zaproponować i zrealizować kompleksowy mechanizm wspomagający modelowanie, a także zarządzanie danymi. Rozwiązanie realizuje postawione cele zachowania wydajności i wspomagania wykorzystywania wzorców projektowych. Dodatkowo, przynajmniej częściowo, spełnia postulaty zgodności z mapowaniem obiektowo-relacyjnym dla platformy Django. Integracja z istniejącymi aplikacjami, poza drobnymi modyfikacjami kodu źródłowego, przebiega bezproblemowo.

Praca dowiodła skuteczności podejścia „najpierw kod źródłowy” w odniesieniu do modelowania dziedziny i obsługi danych w Apache Cassandra. Dzięki niemu proces tworzenia i użytkownia wydajnego schematu jest dużo prostszy. Użytkowanie wysokopoziomowych wzorców modelowania nie wymaga zagłębiania się w szczegóły implementacyjne, co ułatwia proces uczenia się. Autor ma nadzieję, że przeprowadzone przez niego badania przyczynią się do rozwoju prac nad modelowaniem dziedziny danych w Apache Cassandrze. W przyszłości rozwiązania wykorzystujące zaprezentowane w pracy pomysły mogą stać się standardem komunikacji dla tej bazy danych.

Praca może być dalej rozwijana w kierunku rozszerzania mechanizmu o kolejne wzorce modelowania. Ponadto implementacja może być dalej optymalizowana pod względem wydajnościowym. Przede wszystkim chodzi o usprawnienie procesów eksportowania i importowania danych z/do Apache Cassandry, a także migracji. Operacje te są bowiem najbardziej czasochłonne spośród wszystkich zaprezentowanych w pracy.

Autor pragnie podziękować Michałowi Aniserowiczowi za inspirację do stworzenia mechanizmów migracyjnych, a także zgodę na wykorzystanie danych osobowych na potrzeby przykładu 4.27. Bez niego nie byłoby możliwe ukończenie badań w pełnym zakresie zaprezentowanym w pracy.

Bibliografia

[1] *Big Data*. W: Gartner IT Glossary [online], dostęp 25 lipca 2014, `http://www.gartner.com/it-glossary/big-data/`.

[2] Avinash Lakshman, Prashant Malik, *Cassandra - A Decentralized Structured Storage System*, Facebook, 2009.

[3] *Cassandra Overview*. W: Welcome to Cassandra [online], dostęp 31 lipca 2014, `http://cassandra.apache.org`.

[4] Dr. Eric A. Brewer, *Consistency vs. Availability (ACID vs. BASE)*, „PODC Keynote", czerwiec 2000, s. 4-7.

[5] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber, *Bigtable: A Distributed Storage System for Structured Data*, Google Inc., 2006.

[6] Patrick McFadin, *The data model is dead, long live the data model*, C* Summit EU, 2013.

[7] *The write path to compaction*. W: Apache Cassandra 2.0 Documentation [online], dostęp 25 sierpnia 2014, `http://www.datastax.com/documentation/cassandra/2.0/cassandra/dml/dml_write_path_c.html`.

[8] Billy Bosworth, *Why should I use Cassandra?*. W: DataStax Developer Blog [online], dostęp 14 sierpnia 2014, http://www.datastax.com/2012/01/why-should-i-use-cassandra.

[9] Srinath Perera, *Consider the Apache Cassandra database*, IBM developerWorks, czerwiec 2012, s. 4-6.

[10] Charles Bell, Mats Kindahl, Lars Thalmann, *MySQL High Availability: Tools for Building Robust Data Centers*, Wyd. 1, Sebastopol, O'Reilly, ISBN 978-0-596-80730-6, s. 165-168.

[11] *Why Migrate from MySQL to Cassandra?*. W: DataStax White Paper June 2012 [online], dostęp 20 sierpnia 2014, http://www.datastax.com/wp-content/uploads/2012/08/WP-DataStax-MySQLtoCassandra.pdf.

[12] *7 Using Partitioning in an Online Transaction Processing Environment*. W: Oracle® Database VLDB and Partitioning Guide [online], dostęp 27 sierpnia 2014, http://docs.oracle.com/cd/E11882_01/server.112/e25523/part_oltp.htm.

[13] Mats Kindahl, Alfranio Correia, Narayanan Venkateswaran, *MySQL Sharding: Tools and Best Practices for Horizontal Scaling*, „MySQL Connect", wrzesień 2013.

[14] Paul Gil, *What is 'Saas' (Software as a Service)?*. W: About Technology [online], dostęp 28 sierpnia 2014, http://netforbeginners.about.com/od/s/f/what_is_SaaS_software_as_a_service.htm.

[15] Jay Patel, *Cassandra Data Modeling Best Practices, Part 1*. W: ebay tech blog [online], dostęp 28 lipca 2014, http://www.ebaytechblog.com/2012/07/16/cassandra-data-modeling-best-practices-part-1/.

[16] *API*. W: Cassandra Wiki [online], dostęp 21 lipca 2014, https://wiki.apache.org/cassandra/API.

[17] Tyler Hobbs, *Twissandra*. W: GitHub [online], dostęp 29 sierpnia 2014, `https://github.com/twissandra/twissandra`.

[18] Earl Oliver, *Design and Implementation of a Short Message Service Data Channel for Mobile Systems*, Cheriton School of Computer Science, University of Waterloo, 2007.

[19] *UUID and timeuuid*. W: CQL for Cassandra 1.2 [online], dostęp 29 sierpnia 2014, `http://www.datastax.com/documentation/cql/3.0/cql/cql_reference/uuid_type_r.html`.

[20] *Compound keys and clustering*. W: CQL for Cassandra 1.2 [online], dostęp 29 sierpnia 2014, `http://www.datastax.com/documentation/cql/3.0/cql/ddl/ddl_compound_keys_c.html`.

[21] *CassandraLimitations*. W: Cassandra Wiki [online], dostęp 28 sierpnia 2014, `http://wiki.apache.org/cassandra/CassandraLimitations`.

[22] *Using a composite partition key*. W: CQL for Cassandra 1.2 [online], dostęp 30 sierpnia 2014, `http://www.datastax.com/documentation/cql/3.0/cql/cql_reference/create_table_r.html`.

[23] Aleksey Yeschenko, *Cassandra anti-patterns: Queues and queue-like datasets*. W: DataStax Developer Blog [online], dostęp 24 lipca 2014, `http://www.datastax.com/dev/blog/cassandra-anti-patterns-queues-and-queue-like-datasets`.

[24] Dave Gardner, *Cassandra concepts, patterns and anti-patterns*, ApacheCon EU 2012, listopad 2012, s. 47-49.

[25] Jeffrey M. Barnes, *Object-Relational Mapping as a Persistence Mechanism for Object-Oriented Applications*, Macalester College, 2007.

[26] Martin Fowler, *Martin Fowler on ORM Hate*. W: Javalobby [online], dostęp 29 lipca 2014, `http://java.dzone.com/articles/martin-fowler-orm-hate`.

[27] *User input and Doctrine ORM*. W: Doctrine 2 ORM 2 documentation [online], dostęp 30 lipca 2014, http://doctrine-orm.readthedocs.org/en/latest/reference/security.html.

[28] John O'Conner, *Using the Java Persistence API in Desktop Applications*. W: Oracle Java Platform, Standard Edition [online], dostęp 31 lipca 2014, http://www.oracle.com/technetwork/articles/javase/persistenceapi-135534.html.

[29] *Kundera*. W: Github [online], dostęp 21 lipca 2014, https://github.com/impetus-opensource/Kundera.

[30] *Hibernate OGM roadmap*. W: Hibernate Project [online], dostęp 28 lipca 2014, http://hibernate.org/ogm/roadmap/.

[31] Patrick McFadin, *Getting Started with Time Series Data Modeling*. W: Planet Cassandra [online], dostęp 19 lipca 2014, http://planetcassandra.org/blog/post/getting-started-with-time-series-data-modeling/.

[32] *About Indexes in Cassandra*. W: Apache Cassandra 1.1 Documentation [online], dostęp 23 lipca 2014, http://www.datastax.com/docs/1.1/ddl/indexes.

[33] Richard Low, *The sweet spot for Cassandra secondary indexing*. W: Richard Low's blog [online], dostęp 22 lipca 2014, http://www.wentnet.com/blog/?p=77.

[34] *BATCH*. W: CQL for Cassandra 1.2 Documentation [online], dostęp 23 lipca 2014, http://www.datastax.com/documentation/cql/3.0/cql/cql_reference/batch_r.html.

[35] *Using a counter*. W: CQL for Cassandra 2.0 Documentation [online], dostęp 21 lipca 2014, http://www.datastax.com/documentation/cql/3.1/cql/cql_using/use_counter_t.html.

[36] *Migrations*. W: Django documentation [online], dostęp 27 lipca 2014, https://docs.djangoproject.com/en/1.7/topics/migrations/.

[37] Tyler Hobbs, *DataStax Python Driver 2.0 Released*. W: DataStax Developer Blog [online], dostęp 25 lipca 2014, http://www.datastax.com/dev/blog/datastax-python-driver-2-0-released.

[38] Paul Cannon, *Simple data importing and exporting with Cassandra*. W: DataStax Developer Blog [online], dostęp 24 lipca 2014, http://www.datastax.com/dev/blog/simple-data-importing-and-exporting-with-cassandra.

[39] Paul Ducklin, *Serious Security: How to store you users' passwords safely*. W: Naked Security [online], dostęp 3 września 2014, http://nakedsecurity.sophos.com/2013/11/20/serious-security-how-to-store-your-users-passwords-safely/.

[40] B. Kaliski RSA Laboratories, *RFC 2898*. W: PKCS #5: Password-Based Cryptography Specification Version 2.0 [online], dostęp 4 września 2014, https://www.ietf.org/rfc/rfc2898.txt.

[41] Christof Paar, Jan Pelzl, Bart Preneel, *Understanding Cryptography: A Textbook for Students and Practicioners*, Wyd. 1, Springer 2010, ISBN 3-642-04100-0, s. 7.

Oświadczenie

Oświadczam, że Pracę Dyplomową pod tytułem „Mechanizm modelowania danych i mapowania obiektowego dla Apache Cassandry", którą kierował dr inż. Jakub Koperwas, wykonałem samodzielnie, co poświadczam własnoręcznym podpisem.

.....

Jakub Turek