POLITECHNIKA WARSZAWSKA
Wydział Elektroniki i Technik Informacyjnych
Instytut Informatyki

Rok akademicki 2013/2014

PRACA DYPLOMOWA MAGISTERSKA

Jakub Turek

[TYTUŁ]

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 interfejsu programowania aplikacji umożliwiającego efektywne rozwiązywanie problemów w modelowaniu dziedziny danych opartych o bazę danych Apache Cassandra.

Client-server Augmented Reality applications framework for Android system

Summary

The goal of this thesis is to create a framework supporting the development of multiuser applications for Android system. The framework should consist of a client-server bus, as well as several Augmented Reality components. An additional goal is to explore the possibility of adapting commonly used programming practises dedicated to large projects during Andorid applications development. All the goals have been fully accoplished - the output is the implementation of the client-server bus as well as AR components allowing to track the device's geographic coordinates and to render a stable three-dimensional graphics on the display of the device. In order to demonstrate the features of the framework, a sample multiplayer game has been created. The thesis includes a description of the created components' design process, as well as their functionality and structure.

Spis treści

Rozdział 1

Wstęp

Tu będzie wstęp

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 których 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[4], Instagram[5] oraz GitHub[6]. Największa światowa instalacja Cassandry obejmuje około 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[7]. 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.

[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 danych typu klucz-wartość.

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

[4]eBay - największy portal z aukcjami internetowymi na świecie

[5]Instagram - portal pozwalający na umieszczanie fotografii.

[6]GitHub - usługa pozwalająca na przechowywanie i wersjonowanie kodu źródłowego aplikacji.

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

2.1 Model danych

Model danych Apache Cassandra jest analogiczny do BigTable. [1] Można przedstawić go jako dwuwymiarowa mapa trójek wartości:

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

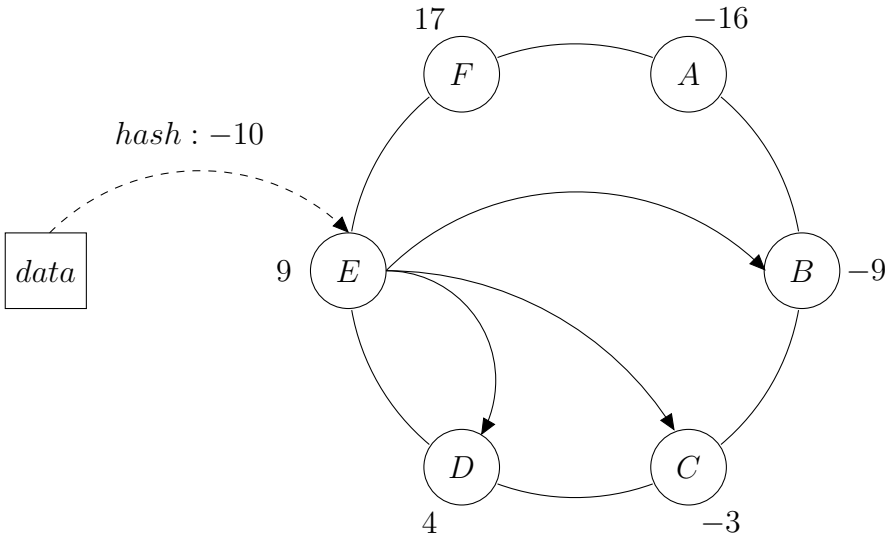
gdzie `RowKey` to identyfikator wiersza, `ColumnKey` to identyfikator kolumny, `Value` to wartość komórki, `Timestamp` to czas aktualizacji komórki, a `TTL` to czas życia danej wartości. [5] Na rysunku 2.1 przedstawiona jest schematyczna ilustracja wiersza danych. Pogrubiona wartość w lewej komórce to klucz wiersza, natomiast wyróżnione wartości w pierwszym wierszu oznaczają klucze poszczególnych kolumn. Każda komórka składa się z trzech wartości: wartości, czasu życia oraz „odcisku czasu".

	ABC	**DEF**	$\cdots$	**XYZ**
123456	test value	another test value	$\cdots$	not a test value
	20	∞	$\cdots$	∞
	1291987837942000	1291980736812000	$\cdots$	1291980736212000

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

2.2 Dystrybucja danych

Do dystrybucji danych wykorzystywana jest funkcja skrótu, która zachowuje kolejność elementów. Węzły są przedstawiane na w topologii pierścienia. Algorytm dystrybucji zostanie omówiony na przykładzie ze schematu 2.2:



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

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 tokenu 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.

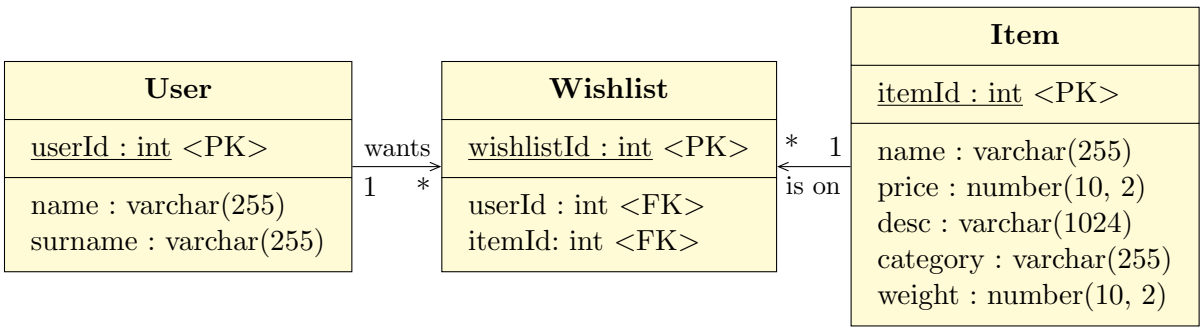
3. Każdy węzeł przechowuje dane, których funkcja skrótu zawiera się w przedziale $(token_{n-1}, token_n]$, gdzie n to numer kolejny węzła rosnący zgodnie z ruchem wskazówek zegara. W przykładzie węzeł C przechowuje wiersze o wartościach funkcji skrótu z przedziału $(-9, -3]$, natomiast węzeł D z przedziału $(-3, 4]$. 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.

2.3 Konsekwencje dla modelowania dziedziny

Wewnętrzna struktura i mechanizm dystrybucji danych wykorzystywany w Apache Cassandra zmienia podejście do modelowania dziedziny znane z relacyjnych 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. [6]

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 2.3: Modelowanie listy życzeń w relacyjnej bazie danych.

Diagram prezentujący zamodelowaną relację dla listy życzeń jest przedstawiony na rysunku 2.3. 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 2.4 zaprezentowano zapytanie, które wyświetla listę życzeń.

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

Rysunek 2.4: Zapytanie, które pobiera wszystkie przedmioty na liście życzeń użytkownika o identyfikatorze 202.

Cassandra umożliwia utworzenie dokładnej repliki relacyjnego modelu danych. Zostało to przedstawione na rysunku 2.5.

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	Cooking recipes	BOOKS	0.2

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

Rysunek 2.5: 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 2.6.

WishlistByUser

	579
123	-

	232
124	-

Rysunek 2.6: Definicja encji listy życzeń umożliwiająca filtrowanie względem użytkownika.

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 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 2.7.

WishlistByUser

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

Rysunek 2.7: Zdenormalizowana postać listy życzeń.

2.4 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[8]. 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.

Wraz z wydaniem 1.2 Apache Cassandry wprowadzony został nowy interfejs dostępu do tej bazy danych. Interfejs ten nosi nazwę CQL[9] 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. [7] Na listingu 2.8 przedstawiono zapytanie w języku CQL, które opisuje omawianą wcześniej encję *User*. Wynikiem wykonania tego zapytania jest prosty schemat modelu danych zaprezentowany na diagramie 2.9.

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

Rysunek 2.8: Zapytanie CQL, które tworzy encję *User*.

[8]Dokumentacja interfejsu dostępna jest pod adresem `https://wiki.apache.org/cassandra/API10`.
[9]CQL (ang. Cassandra Query Language) - język zapytań Cassandry.

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

Rysunek 2.9: Wynik wykonania zapytania 2.8.

2.5 Modelowanie - wzorce i antywzorce

Pomimo, że CQL znacząco upraszcza modelowanie w Cassandrze 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 analogiczną role do wzorców i antywzorców projektowych znanych z programowania. Na ich opis składa się możliwie ogólna definicja problemu, a także poprawny (lub niepoprawny) sposób jego rozwiązania.

Przykładem antywzorca modelowania jest kolejka oparta na kolumnach. [8] 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 wykonywane 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. [9] Encja *Item* z diagramu 2.5 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. [17] 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. [18]

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 narzędzi do ochrony przez atakami typu SQL Injection[1]. [10]
- 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.

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

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 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. 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. [19] Interfejs JPA operuje na 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.

[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.

[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 interfejs 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.[11] 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

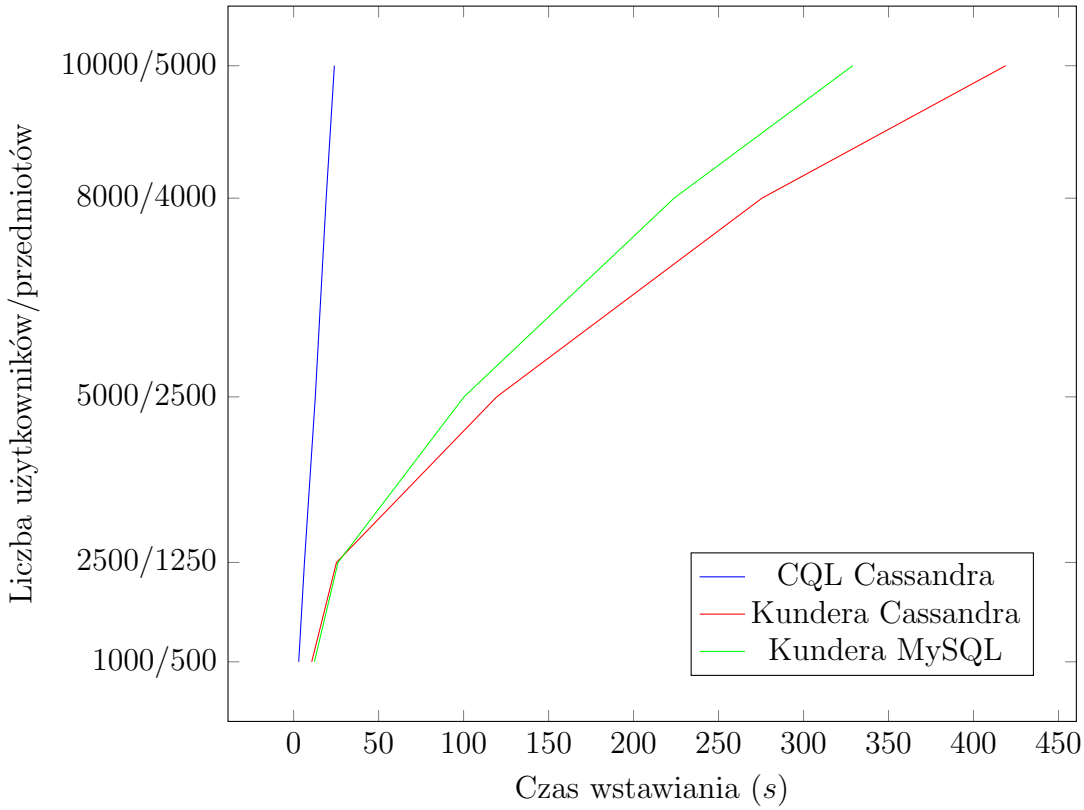
Wyniki pomiarów na stronie domowej projektu Kundera świadczą o tym, że biblioteka nie wprowadza znacznych narzutów wydajnościowych względem bezpośredniego wykorzystania interfejsu bazy danych. Ważniejsze jest jednak sprawdzenie jak duży narzut wprowadza dostosowanie relacyjnego modelu danych do Apache Cassandry. W tym celu Autor przeprowadził testy 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 2.7. Drugi test sprawdza czasy dla znormalizowanego modelu zaprezentowanego na diagramie 2.3, 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



Rysunek 3.3: Porównanie czasu wstawiania wielu rekordów.

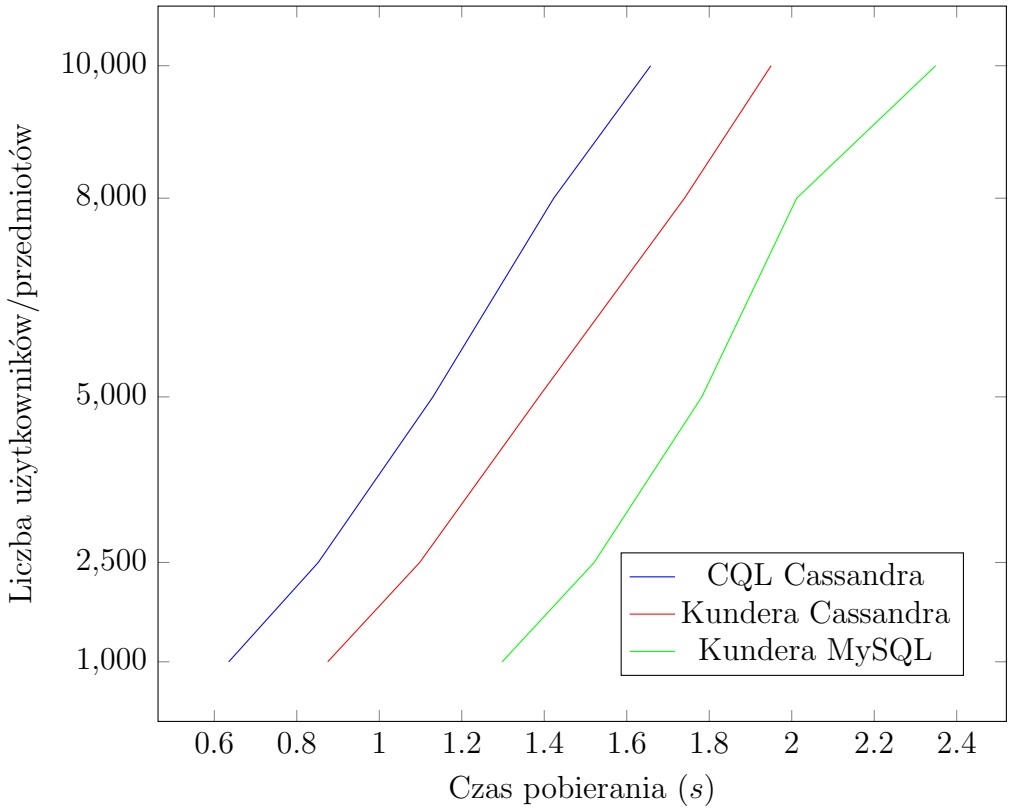
zyskiem z takiego rozwiązania będzie możliwość wykorzystania natywnego mechanizmu klastrowania węzłów Cassandry. Dzięki temu wyeliminowany zostanie pojedynczy punkt awarii systemu. Ten sam efekt można jednak uzyskać stosując rozwiązana dedykowane dla relacyjnych baz danych. Przykładami takich rozwiązań 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. Przypuszczalnie 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 da-

[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.

nych.

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 wydajności pomiędzy szybkością zapisu danych w modelach zoptymalizowanym i niezoptymalizowanym świadczy o tym, że do efektywnego modelowania dziedziny w Apache Cassandra niezbędna jest wiedza na temat sposobu przechowywania danych przez tę bazę i podejmowanie świadomych wyborów. Dostarczona przez ORMy mechanizmy mogą natomiast 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.

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.

```
@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.

[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.

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.

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. [20]

3.5 Mapowanie obiektowe dla Cassandry - koncepcja

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 2.7.

```
@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.

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 najpopularniejsze mechanizmy to Django

```
@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.

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 listingu 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*.

[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`.

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 ograniczonym 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

[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.

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.

4.2 OMC - podstawowe pojęcia

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

4.2.1 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.

4.2.2 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`.

4.2.3 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 4.1.

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

Rysunek 4.1: Struktura adresu URL opisującego połączenie z bazą danych Cassandra.

Wszystkie elementy ujęte w znaki **{}** w adresie 4.1 należy zamienić na wartości według poniższego objaśnienia:

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[3] 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).

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

Do utworzenia silnika służy metoda `Engine.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 4.2.

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

Rysunek 4.2: Przykład wywołania tworzącego silnik i przypisującego go do modelu.

4.3 Definiowanie modelu

Definiowanie modelu odbywa się poprzez stworzenie nowej klasy, która dziedziczy po klasie bazowej `Model`. Podstawowy model opisuje zawartość jednej tabeli, jego pola zaś odpowiadają kolumnom tej tabeli. Przykładowy model został przedstawiony na listingu 4.3.

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

Rysunek 4.3: Przykładowy model danych OMC.

Po zdefiniowaniu klasy modelu oraz utworzeniu i przypisaniu silnika jak na listingu 4.2 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 4.4. Należy zauważyć, że OCM w schemacie bazodanowym wykorzystuje notację z podkreśleniami[4]. 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);
```

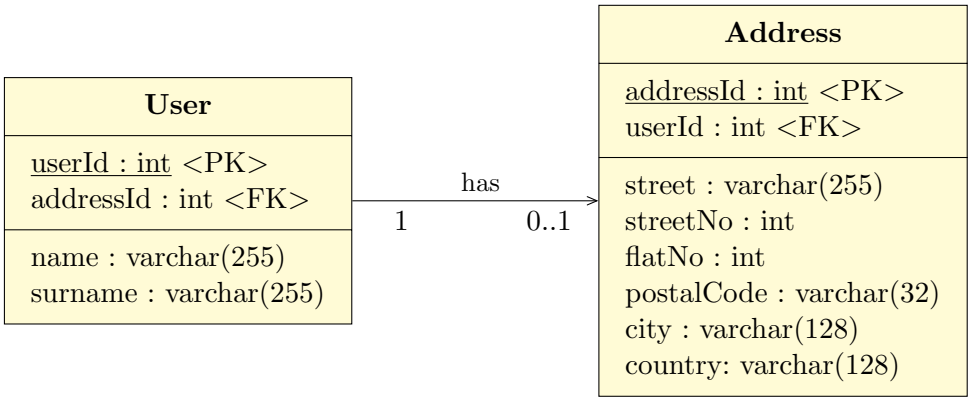
Rysunek 4.4: Zapytanie tworzące tabelę dla modelu 4.3.

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

4.4 Modelowanie zależności między danymi

Autor celowo nie używa terminu „relacja", aby uniknąć skojarzenia z RDBMS[5]*. 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 4.5 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 4.5: 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:

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

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.

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

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 4.5 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 4.6.

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

Rysunek 4.6: 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. [6].

OMC posiada trzy wbudowane tryby, które automatyzują proces zarządzania zależnościami między danymi. Różnią się one pomiędzy sobą wydajnością (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).

4.4.1 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 4.6. Listing 4.7 przedstawiono definicję encji adres zapisaną w OCM. Model użytkownika został pokazany na listingu 4.8.

```
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 4.7: 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 4.8: Denormalizacja przez pole - definicja encji użytkownik.

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 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[6] przy definiowaniu denormalizowanych pól. Gdyby w modelu Address typ pola `country` został zmieniony 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 4.6.

4.4.2 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.

[6] 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.

- Domyślnie (przy braku specyfikacji listy pól) denormalizowane są wszystkie pola obiektu zależnego.

Na listingu 4.9 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 4.10.

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

Rysunek 4.9: Denormalizacja przez tabelę - definicja encji użytkownik.

	addressId	street	street_no	postal_code	city	country
233	754	Nowowiejska	15/19	00-665	Warsaw	Poland

Rysunek 4.10: 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.

4.4.3 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 4.9. Mechanizm wykorzystuje połączenie niejawne. Tabela normalizująca nie jest mapowana jako część modelu danych. Zamiast tego, podczas wywołania metody `user.address.get()` (gdzie `user` jest obiektem klasy `User`), następują dwa odwołania. Pierwsze pobiera z tabeli normalizującej identyfikatory powiązanych obiektów, a drugie pobiera szczegóły tych obiektów.

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

Rysunek 4.11: 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 nadrzędną. 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 Wsparcie dla wzorców modelowania

OMC został zaprojektowany w taki sposób, aby dostarczać mechanizmów wspierających wzorce projektowe. W tej sekcji Autor opisuje wszystkie wzorce, które były analizowane, modelowane, testowane i optymalizowane w trakcie prac implementacyjnych nad OMC.

4.5.1 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**. [12] 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 4.12 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 4.12: 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*[7]. 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 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 4.13 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 4.13: 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 dodatkową kolumnę `event_time_day`[8] o typie tekstowym. Pole to jest automatycznie włączane do klucza partycjonowania.
- 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`.

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

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

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`) oraz 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 4.14.

```
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 4.14: Modelowanie szeregu zdarzeń. Przedawniające się odczyty jako kolumny z sortowaniem od najnowszego.

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

4.5.2 Wsparcie dla kolejek

W sekcji 2.5 został przytoczony antywzorzec kolejki. Problem został opisany przez Alekseya Yeschenko na blogu firmy DataStax. [8] 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[9]. 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 4.15.

```
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]
```

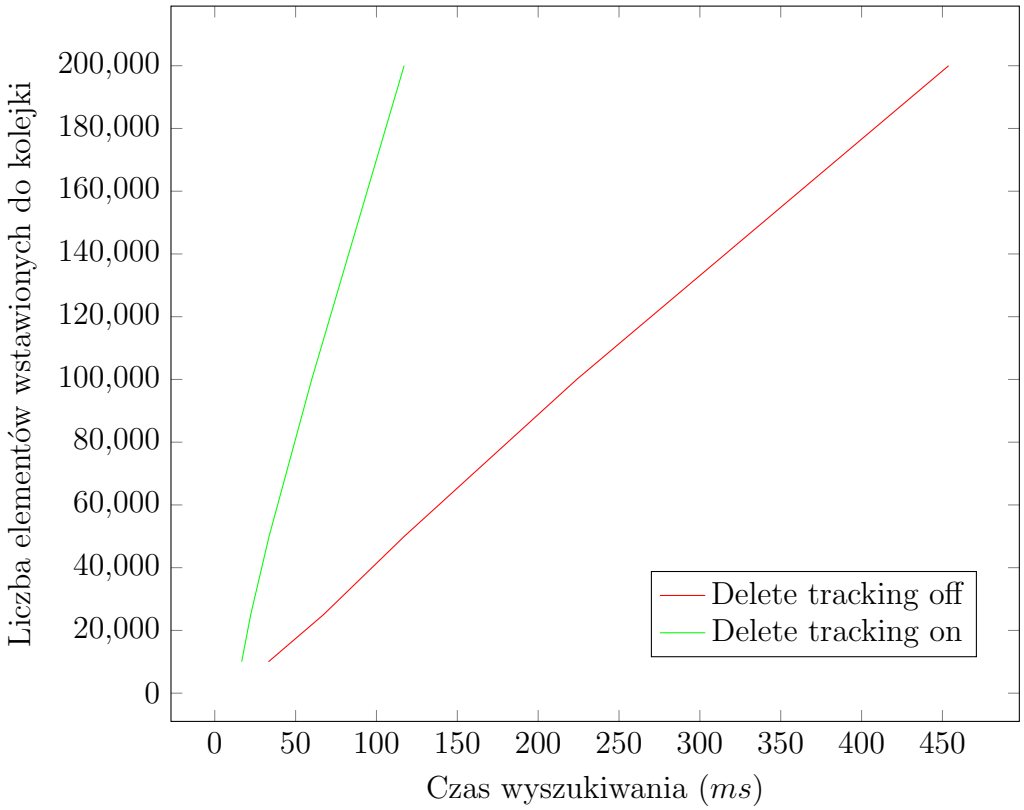
Rysunek 4.15: Modelowanie kolejki typu FIFO w OMC.

Parametr `__track_deletes__` umożliwia śledzenie operacji usuwania. 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` 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 4.15 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`) większy od `second_element.enqueued_at` - automatycznie dodawane przez OMC.

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



Rysunek 4.16: Porównanie czasu wstawiania wielu rekordów.

Wykorzystanie opcji `__track_deletes__` w sposób znaczący wpływa na wydajność kolejki FIFO. Na wykresie 4.16 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 4.15).

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.

4.5.3 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 aktualizacja. [9] 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 4.17.

```
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 4.17: Przykład selektywnej aktualizacji.

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 modelu poprzez zadeklarowanie pola `__selective_update__ = False`.

4.5.4 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 [13] 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 4.18.

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

Rysunek 4.18: Przykładowe wartości w tabeli użytkownicy.

Przykładowy wygląd danych w tabeli, która realizuje indeks drugiego stopnia dla miasta (kolumna `city`), został zaprezentowany na diagramie 4.19.

	jturek	**manisero**
Warsaw	null	null

Rysunek 4.19: Przykładowa tabela, która mogłaby realizować indeks drugiego stopnia dla kolumny *city* z diagramu 4.18.

Wadą rozwiązania przedstawionego na diagramie 4.19 jest brak skalowalności. Przy dodawaniu użytkowników wiersz rozszerza się o kolejne kolumny, 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. [14] 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 4.20.

UserEmailIndex

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

Rysunek 4.20: Przykład tabeli przechowującej indeks dla unikalnych wartości (adres e-mail użytkownika).

OMC wspiera automatyczne tworzenie indeksów dla wartości unikalnych. Na listingu 4.21 przedstawiono sposób utworzenia indeksu dla wartości unikalnych dla pola `email`.

```
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 4.21: Modelowanie tabeli użytkownicy z wykorzystaniem indeksu dla wartości unikalnych.

Flaga `searchable_unique=True` powoduje, że dla danego pola tworzony jest indeks unikalny. Indeks jest w całości zarządzany przez OMC. Podczas wyszukiwania metodą `User.objects().find(email='J.Turek@stud.elka.pw.edu.pl')` mechanizm 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 4.21 indeks wartości unikatowych zostanie umieszczony w tabeli o nazwie `user_email_index`.

4.6 Przetwarzanie partiami

Apache Cassandra zapewnia wsparcie dla operacji atomowych. [15] Wykorzystując partie[10] 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 Cassandra nie zapewnia integralności wykonywanych w partii zapytań: dane mogą być modyfikowane równolegle przez inne zapytania. Przykład atomowych operacji został zaprezentowany na diagramie 4.22. 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@stud.elka.pw.edu.pl'
APPLY BATCH;
```

Rysunek 4.22: 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[11], a następnie wykonywać zapytania z kolejki jako operacja atomowa. Przykład wykorzystania operacji na partiach w OCM zaprezentowano na listingu 4.23.

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

4.7 Wsparcie dla liczników

Apache Cassandra udostępnia wsparcie dla specjalnego typu danych - liczników. [16] Liczniki to kolumny specjalnego typu `counter`, na których można wykonywać tylko dwie

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

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

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

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 4.24.

```
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 4.24: 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 `sweden.country = 'Germany'`. Dzięki temu wszystkie operacje wykonywane 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 (`increment()`) oraz dekrementację (`decrement()`). Metody wywołane bez podania parametru standardowo zwiększają/zmniejszają wartość licznika 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.

4.8 Lista życzeń użytkownika - studium przypadku

Encję użytkownika przy pomocy OMC można opisać modelem `User` zaprezentowanym na listingu 4.25. Model posiada trzy pola: identyfikator (`id`), imię (`name`) oraz nazwisko (`surname`).

```
class User(Model):
    id = UuidField(type=TimeUuid, auto_generate=True, partition_key=True)
    name = TextField(selectable=True)
    surname = TextField(selectable=True)
```

Rysunek 4.25: Tabela użytkownika opisana przy pomocy OMC.

Identyfikator jest opisany polem typu `UuidField`. Ze względu na zadeklarowany parametr `type=TimeUuid` pole zostanie zmapowane w Cassandrze jako dana o typie `timeuuid`. Mechanizm OMC posiada wbudowaną walidację dla identyfikatorów. Przypisanie polu `id` instancji modelu `User` błędnej wartości spowoduje zgłoszenie wyjątku `AttributeError`. Parametr `auto_generate=True` oznacza, że jeżeli identyfikator nie zostanie ustawiony manualnie, to zostanie wygenerowany automatycznie podczas mapowania modelu na bazę danych. Ustawienie `partition_key=True` oznacza natomiast, że pole będzie częścią klucza wiersza tabeli.

Imię i nazwisko opisane są polem typu `TextField`. Jest ono mapowane na bazodanowy typ `text`. Deklaracja `selectable=True` oznacza, że na polu zostanie utworzony indeks drugiego stopnia. Dzięki temu możliwe jest przeszukiwanie tabeli po wartości tego pola. Na listingu 4.26 przedstawiono odwołanie do OCM, dzięki któremu można wyszukać wszystkich użytkowników o imieniu Jakub i nazwisko Turek.

```
User.objects().find(name='Jakub', surname='Turek')
```

Rysunek 4.26: Przykład wyszukiwania użytkownika po imieniu i nazwisku.

Opis encji przedmiotu w OMC został zaprezentowany na listingu 4.27.

```
class Item(Model):
    id = UuidField(type=TimeUuid, auto_generate=True, partition_key=True)
    name = TextField(searchable=True, length=255)
    price = DecimalField(total_positions=8, decimal_positions=2)
    desc = TextField()
    category = DictionaryField(entries=['BOOKS', 'CLOTHES', 'FURNITURE'])
    weight = DecimalField(total_positions=5, decimal_positions=1)
```

Rysunek 4.27: Tabela przedmiotu opisana przy pomocy OMC.

W stosunku do poprzedniej definicji zostały użyte dwa nowe typy pól: `NumberField` oraz `DictionaryField`. Pierwsze z nich służy do przechowywania wartości liczb dziesiętnych. Cassandra nie posiada odpowiednika znanego z relacyjnych baz danych stałoprzecinkowego typu `NUMBER`. O umieszczanie poprawnie zaokrąglonych wartości musi zadbać autor oprogramowania korzystającego z Apache Cassandra. Aby to ułatwić, OMC dostarcza programowe wsparcie dla operacji stałoprzecinkowych. Pole `DecimalField` posiada dwa parametry. Opcja `total_positions` opisuje maksymalną liczbę cyfr dziesiętnych, z których może składać się liczba, a `decimal_positions` definiuje ile z tych cyfr należy do części ułamkowej pola. `DecimalField` w przezroczysty dla programisty sposób wspiera przypisywanie i wszystkie operacje arytmetyczne dla typów `int`, `long` oraz `float`.

Pole `DictionaryField` dostarcza wsparcia dla słowników jednokrotnego wyboru. Parametr `entries` pozwala na wyspecyfikowanie listy dozwolonych wartości pola. Podanie wartości spoza listy powoduje zgłoszenie wyjątku `AttributeError`. Typ pola jest automatycznie wnioskowany z wartości parametru `entries`. Dla listy elementów typu `str` pole zostanie zmapowanie na wartość typu `text`. W przypadku, gdy typ nie może być wywnioskowany na podstawie listy, należy podać go jawnie za pomocą parametru `type`.

Opis encji listy życzeń przedstawia listing 4.28.

```
class Wishlist(Model):
    userId = UuidField(type=TimeUuid, partition_key=True)
    item = DenormalizedField(relates=Item,
                             clustering_keys=['itemId'],
                             fields=['name', 'price'])
```

Rysunek 4.28: Tabela listy życzeń opisana przy pomocy OMC.

Pole `DenormalizedField` spełnia postulaty przedstawione w sekcji 3.5. Istotny jest fakt, że na podstawie parametru `clustering_keys=['itemId']` pole `itemId` staje się częścią złożonego klucza głównego tabeli.

Bibliografia

[1] http://static.googleusercontent.com/media/research.google.com/pl//archive/bigtable-osdi06.pdf

[2] http://www.datastax.com/documentation/articles/cassandra/cassandrathenandnow.html

[3] http://cassandra.apache.org

[4] Towards Robust Distributed Systems, Dr. Eric A. Brewer, PODC Keynote

[5] The data model is dead, long live the data model, Patrick McFadin

[6] http://www.ebaytechblog.com/2012/07/16/cassandra-data-modeling-best-practices-part-1/

[7] https://wiki.apache.org/cassandra/API

[8] http://www.datastax.com/dev/blog/cassandra-anti-patterns-queues-and-queue-like-datasets

[9] http://www.slideshare.net/davegardnerisme/cassandra-concepts-patterns-and-antipatterns

[10] http://doctrine-orm.readthedocs.org/en/latest/reference/security.html

[11] https://github.com/impetus-opensource/Kundera

[12] http://planetcassandra.org/blog/post/getting-started-with-time-series-data-modeling/

[13] http://www.datastax.com/docs/1.1/ddl/indexes

[14] http://www.wentnet.com/blog/?p=77

[15] http://www.datastax.com/documentation/cql/3.0/cql/cql_reference/batch_r.html

[16] http://www.datastax.com/documentation/cql/3.1/cql/cql_using/use_counter_t.html

[17] http://digitalcommons.macalester.edu/cgi/viewcontent.cgi?article=1006&context=mathcs_hon

[18] http://java.dzone.com/articles/martin-fowler-orm-hate

[19] http://www.oracle.com/technetwork/articles/javase/persistenceapi-135534.html

[20] http://hibernate.org/ogm/roadmap/

Oświadczenie

Oświadczam, że Pracę Dyplomową pod tytułem *"[TYTUŁ]"*, którą kierował dr inż. Jakub Koperwas, wykonałem samodzielnie, co poświadczam własnoręcznym podpisem.

.....

Jakub Turek