

**POLITECHNIKA WROCŁAWSKA**  
**WYDZIAŁ ELEKTRONIKI**

---

**KIERUNEK:** Informatyka (INF)  
**SPECJALNOŚĆ:** Inżynieria systemów informatycznych (INS)

**PRACA DYPLOMOWA**  
**MAGISTERSKA**

Analiza porównawcza frameworków  
internetowych w języku Ruby w zastosowaniach  
GISowych

Comparative analysis of Ruby's web frameworks  
for Geographic Information Systems

**AUTOR:**  
Mikołaj Grygiel

**PROWADZĄCY PRACE:**  
dr inż. Roman Ptak, W4/K9

**OCENA PRACY:**



# **Spis treści**

<b>1</b>	<b>Badania</b>	<b>3</b>
1.1	Plan badań . . . . .	3
1.1.1	Aplikacja zaimplementowana na potrzeby badań . . . . .	3
1.1.2	Narzędzia wykorzystane podczas badań . . . . .	6
1.1.3	Środowisko badawcze . . . . .	6
1.2	Porównanie funkcjonalności wybranych frameworków . . . . .	7
1.2.1	Przechowywanie danych . . . . .	7
1.2.2	Obiektowe przetwarzanie danych . . . . .	8
1.2.3	Prezentowanie danych . . . . .	8
1.2.4	Podsumowanie . . . . .	9
1.3	Porównianie struktur wykonanych projektów . . . . .	10
1.4	Porównianie wydajności zaimplementowanych aplikacji . . . . .	16
1.4.1	Interfejs użytkownika . . . . .	16
1.4.2	Zapytania HTTP . . . . .	20
1.4.3	Komunikacja z bazą danych . . . . .	27
<b>2</b>	<b>Podsumowanie</b>	<b>31</b>
2.1	Analiza wyników badań . . . . .	31
2.2	Realizacja celu projektu . . . . .	31
<b>Literatura</b>		<b>33</b>



# Rozdział 1

## Badania

### 1.1 Plan badań

Na podstawie dostępnej dokumentacji oraz zaimplementowanych aplikacji w każdym z badanych narzędzi, frameworki zostaną porównane w następujących aspektach:

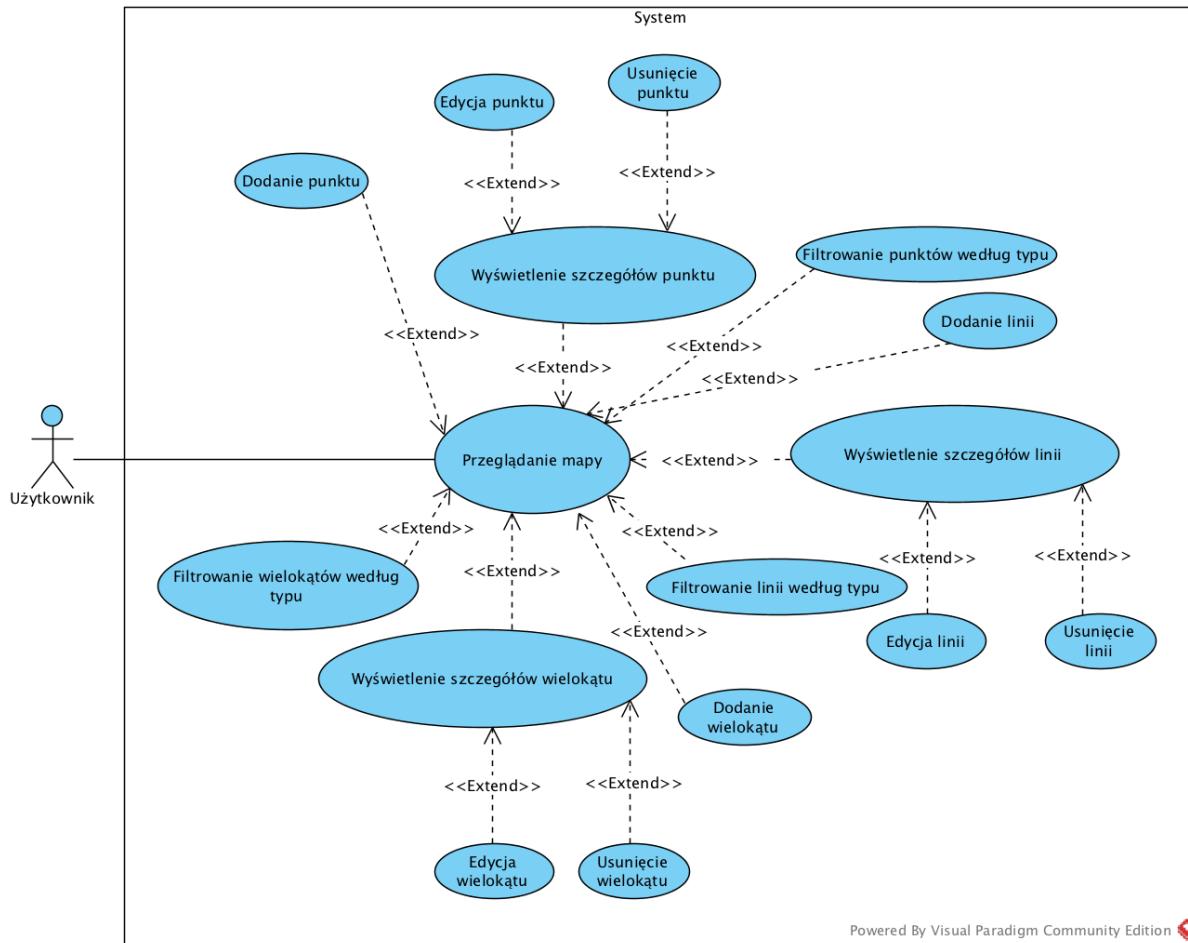
- dostępne funkcjonalności
- struktura utworzonego projektu
- wydajność zaimplementowanej aplikacji

#### 1.1.1 Aplikacja zaimplementowana na potrzeby badań

W celu wykonania analizy porównawczej w każdym z wybranych frameworków została zaimplementowana aplikacja internetowa operująca na danych geograficznych posiadająca takie same funkcjonalności. Aplikacje zostały zrealizowane zgodnie z architekturą i konwencją wykorzystywaną w danym frameworku zawartych w dokumentacji narzędzia. W projektach wykorzystano jedynie biblioteki niezbędne do zrealizowania wymaganych funkcjonalności. Nie używano bibliotek modyfikujących architekturę aplikacji lub funkcjonalności frameworka. Aplikacja ma za zadanie wykonywać 4 podstawowe operacje na danych geograficznych:

- tworzenie
- odczytywanie
- aktualizowanie
- usuwanie

Wszystkie funkcjonalności wykonanej aplikacji zostały zaprezentowane na rysunku 1.1.



Rysunek 1.1 Diagram przypadków użycia aplikacji

Aby przeprowadzić badania w realistycznych warunkach aplikacja korzysta z gotowych zbiorów danych:

- Punkty - pochodzące ze zbioru "Państwowy rejestr nazw geograficznych - miejscowości", zbiór zawiera 256796 obiektów, dostępny jest pod adresem <http://www.codgik.gov.pl/index.php/dane/prng.html>.
- Linie - dane uzyskano z serwisu <http://download.geofabrik.de/europe/poland.html> pozwalającego pobrać drogi z terenu Polski z bazy danych OpenStreetMap. Zbiór obejmuje 2519660 obiektów.
- Wielokąty - obiekty pobrano z Państwowego Rejestru Granic dostępnego pod adresem <http://gis-support.pl/baza-wiedzy/dane-do-pobrania/>, zbiór liczy 59978 obiektów.

Każda wersja aplikacji posiada taką samą bazę danych, której schemat znajduje się na rysunku 1.2. Jako bazę danych wykorzystano PostgreSQL z rozszerzeniem PostGIS. Dane geograficzne w warstwie aplikacji przetwarzano używając biblioteki RGeo.

points	
id	SERIAL
coordinates	geometry
name	CHARACTER VARYING
object_type	CHARACTER VARYING
object_class	CHARACTER VARYING
voivodeship	CHARACTER VARYING
county	CHARACTER VARYING
commune	CHARACTER VARYING
terc	CHARACTER VARYING
created_at	TIMESTAMP(6) WITHOUT TIME ZONE
updated_at	TIMESTAMP(6) WITHOUT TIME ZONE

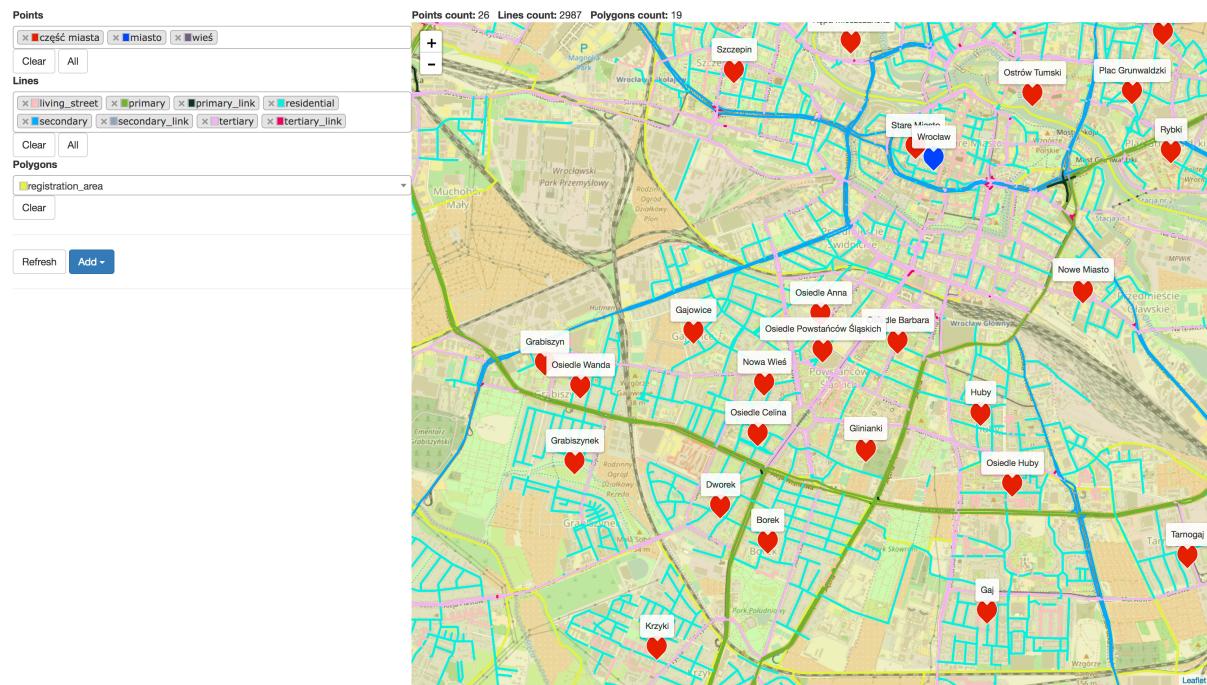
polygons	
id	SERIAL
name	CHARACTER VARYING
coordinates	geometry
terc	CHARACTER VARYING
unit_type	INTEGER
created_at	TIMESTAMP(6) WITHOUT TIME ZONE
updated_at	TIMESTAMP(6) WITHOUT TIME ZONE

lines	
id	SERIAL
name	CHARACTER VARYING
coordinates	geometry
road_type	CHARACTER VARYING
created_at	TIMESTAMP(6) WITHOUT TIME ZONE
updated_at	TIMESTAMP(6) WITHOUT TIME ZONE

Rysunek 1.2 Schemat ERD bazy danych

Do prezentacji danych na mapie wykorzystano bibliotekę LefletJS, ponieważ jest darmowa i ma otwarte źródła. Mapa pochodzi z serwisu OpenStreetMap. Widok gotowej aplikacji znajduje się na rysunku 1.3.



Rysunek 1.3 Widok gotowej aplikacji

### 1.1.2 Narzędzia wykorzystane podczas badań

Do zmierzenia wydajności zaimplementowanych aplikacji zastosowano następujące narzędzia:

- Capybara - biblioteka w języku Ruby pozwalająca symulować interakcje prawdziwego użytkownika z wybraną aplikacją internetową. Capybara dostarcza szereg metod pozwalających zapisać jako scenariusz kolejne kroki postępowania użytkownika m.in. kliknięcie wybranego elementu lub wprowadzenie danych do formularza. Do wspierania JavaScriptu, Capybara używa Selenium. Selenium jest narzędziem, które umożliwia automatyczne wykonanie testów akceptacyjnych aplikacji internetowej w wybranej przeglądarce. W przeprowadzonych badaniach testy uruchomione zostały w przeglądarce Firefox. [1]
- JMeter - aplikacja napisana w języku Java dostępna na licencji otwartego źródła. JMeter służy do wykonywania testów wydajności aplikacji internetowych. W niniejszych badaniach JMeter wykorzystywał protokół HTTP do wysyłania określonych zapytań do aplikacji. W badaniach pomocna również okazała się funkcja symulowania wielu użytkowników - JMeter uruchamia wybrany scenariusz na wielu wątkach aby sprawdzić jak zachowa się aplikacja przy wielu równoległych zapytaniach. [9]
- Ruby Benchmark - biblioteka pozwalająca na zmierzenie czasu wykonywania podanego blogu kodu. Ruby Benchmark został wykorzystany przy zmierzeniu czasu wykonywania scenariuszy utworzonych przy pomocy Capybary oraz czasu wykonywania operacji na bazie danych. [6]

### 1.1.3 Środowisko badawcze

Do przeprowadzenia badań użyto dwóch komputerów:

#### 1. Komputer - serwer

- MacBook Pro(Retina, 13 cali, model: Early 2015)
- Procesor: Intel Core i5 2,7 GHz 2 rdzenie
- Pamięć RAM: 8GB DDR3 1867 MHz
- Dysk: 120GB APPLE SSD SM0128G

#### 2. Komputer - tester

- MacBook Mini 6.1
- Procesor: Intel Core i5 2,5 GHz 2 rdzenie
- Pamięć RAM: 8GB DDR3 1600 MHz
- Dysk: 240GB GOODRAM SSD CX100

Na komputerze - serwer uruchomiona była aktualnie testowana aplikacja. Komputer - tester wykonywał zaimplementowane badania. Podczas badań komputery były podłączone ze sobą lokalnie przy pomocy sieci Wi-Fi 802.11n i nie miały dostępu do zewnętrznej sieci. Na komputerach uruchomione były tylko niezbędne aplikacje do przeprowadzenia badań.

## 1.2 Porównanie funkcjonalności wybranych frameworków

### 1.2.1 Przechowywanie danych

Ruby on Rails posiada największe wsparcie w przechowywaniu danych geograficznych z rozpatrywanych narzędzi. Biblioteka ActiveRecord domyślne wykorzystywana w Ruby on Rails do mapowania obiektowo-relacyjnego posiada adaptery dla następujących baz danych: PostgreSQL z dodatkiem PostGIS, MysqlSpatial, SpatiaLite. Za pomocą adapterów, w migracjach można używać typów kolumn z przestrzennych baz danych. Przy komunikacji typy przestrzenne z bazy danych mapowane są na typy zaimplementowane w bibliotece RGeo. Konfiguracja nie wymaga wiele pracy, wystarczy dodać odpowiednią bibliotekę do projektu i w konfiguracji bazy danych ustawić odpowiedni adapter. Od tego momentu można w migracjach przy definiowaniu kolumn używać wszystkich typów przestrzennych dostarczanych przez wybraną bazę danych. Dane przestrzenne podczas wykonywanych akcji między aplikacją, a bazą danych są automatycznie mapowane.

Tworzenie bazy danych przestrzennej za pomocą Rody nie stanowi problemu. Framework pozwala używać kodu SQL w migracjach dlatego można włączyć dowolne rozszerzenie wybranej bazy danych i używać dodatkowych typów. Roda nie zapewnia specjalnego wsparcia przy mapowaniu danych przestrzennych na obiekty, dane są zwracane w formie tekstowej binarnej wartości pola.

Hanami w migracjach pozwala ozywać kodu napisanego w języku SQL do np. definiowania typów lub włączania dodatków w bazie danych, dlatego bez problemów można stworzyć i wersjonować bazę danych z typami przestrzennymi. Jednak framework, podobnie jak Ruby on Rails, posiada mechanizm automatycznego mapowania typów bazy danych do typów obiektów, który nie pozwala na definiowanie własnych typów lub własnych reguł mapowania. To ograniczenie powoduje otrzymanie wyjątku przy próbie uruchomienia aplikacji zawierającej nieznany typ. Framework wspiera jedynie typy przestrzenne zdefiniowane w bazie danych PostgreSQL, które są zapisywane postaci tekstuowej zawierającej współrzędne. Możliwe jest zapisanie wszystkich 3 podstawowych typów przestrzennych: punktu, linii, wielokąta. PostgreSQL dostarcza podstawowe operacje wyszukiwania danych przestrzennych tylko dla punktów, dlatego zastosowanie PostgreSQL bez dodatku PostGIS mocno ogranicza możliwości budowanej aplikacji. Dodanie obsługi typów geometrycznych dostarczonych przez dodatek PostGIS nie wymaga dużej ilości kodu, wymagana jest dobra znajomość języka Ruby i mechanizmu przeddefiniowania istniejących już klas tzw. "monkey patching". Najłatwiejsze rozwiązanie pozwalające używać typów geometrycznych z biblioteki PostGIS znajduje się na listingu 1.1. Przy takim rozwiążaniu wartości z kolumny geometrycznej mapowane są na typ String z wartością binarną.

Fragment kodu 1.1 Obsługa typów geometrycznych z biblioteki PostGIS przez Hanami

```
require 'rom/sql/extensions/postgres/inferencer'
require 'rom/sql/extensions/postgres/types'

ROM::SQL::Schema::PostgresInferencer.class_eval do
  alias map_db_type_original map_db_type

  def map_db_type(db_type)
    # najpierw następuje próba użycia oryginalnej metody mapowania typów
    # jeśli zakończy się ona nie powodzeniem, to następuje sprawdzenie
    # czy typ kolumny z bazy danych rozpoczyna się od słowa 'geometry',
    # wtedy dane są rzutowane do typu ROM::Types::String
    map_db_type_original(db_type) || (db_type.start_with?('geometry') ? ROM
      ::Types::String : nil)
  end
end
```

## 1.2.2 Obiektowe przetwarzanie danych

Do obiektowego przetwarzania danych dostępne są dwie biblioteki Rgeo i GeoRuby. Ruby on Rails, dzięki wspomnianym wcześniej adapterom dużo wygodniej jest korzystać z Rgeo dzięki automatycznemu mapowaniu danych relacyjnych na obiekty. Frameworki Roda oraz Hanami w żaden sposób nie rozszerzają możliwości bibliotek Rgeo i GeoRuby. W Hanami, zaimplementowana metoda mapująca wartości przestrzenne na typ obiektowy zwraca wartość binarną, Roda domyślne zwraca dane przestrzenne w takiej formie. Wartość binarną dalej można zamienić na typ z biblioteki Rgeo, co zaprezentowano na listingu 1.2

Fragment kodu 1.2 Tworzenie obiektu Rgeo z zapisu binarnego danych przestrzennych

```
module CoordinatesHelper
  FACTORY = RGeo::Geographic.spherical_factory(:srid => 4326)

  # utworzenie obiektu Rgeo z binarnego zapisu danych przestrzennych
  def coordinates_object
    RGeo::WKRep::WKBParser.new(FACTORY, support_ewkb: true,
      default_srid: 4326).parse(coordinates)
  end

  # rzutowanie obiektu przestrzennego do postaci "well known text"
  def coordinates_text
    coordinates_object.as_text
  end
end
```

## 1.2.3 Prezentowanie danych

Framework Ruby on Rails posiada biblioteki do integracji obu narzędzi do prezentacji danych geograficznych wspomnianych w rozdziale ??:

1. leaflet-rails - dołącza do projektu Ruby on Rails bibliotekę leafletjs napisaną w języku JavaScript wraz ze stylami oraz dodaje metody do widoków pozwalające na dołączenie mapy i podstawową konfigurację bez potrzeby używania JavaScriptu.

2. Google Maps for Rails - biblioteka opakowuje JavaScriptową bibliotekę GoogleMaps JavaScript API, ułatwiając jej dodanie do projektu.

Dzięki takiej integracji zgodnie z sposobem dołączania bibliotek do projektu w języku Ruby [8], wspomniane biblioteki są przechowywane centralnie w systemie zamiast w projekcie. Roda oraz Hanami w przeciwieństwie do Ruby on Rails nie posiadają bibliotek wspomagających dołączenie do projektu biblioteki Leafletjs lub Google Maps Javascript API. Żeby skorzystać jednej z wymienionych bibliotek należy dołączyć jej kod do projektu.

Przy odczycie danych w formacie JSON, Ruby on Rails oraz Roda automatycznie serializują model do postaci JSON. W projekcie wykorzystującym Hanami, niezbędne jest dodanie zewnętrznej biblioteki realizującej serializowanie danych np. *Roar* i zdefiniowanie odpowiedniego prezentera. Przykładowy prezenter został zaprezentowany na listingu 1.3

Fragment kodu 1.3 Prezenter dla modelu Point

```
require 'roar/decorator'
require 'roar/json'

module Web::Representers
  class Point < Roar::Decorator
    include Roar::JSON

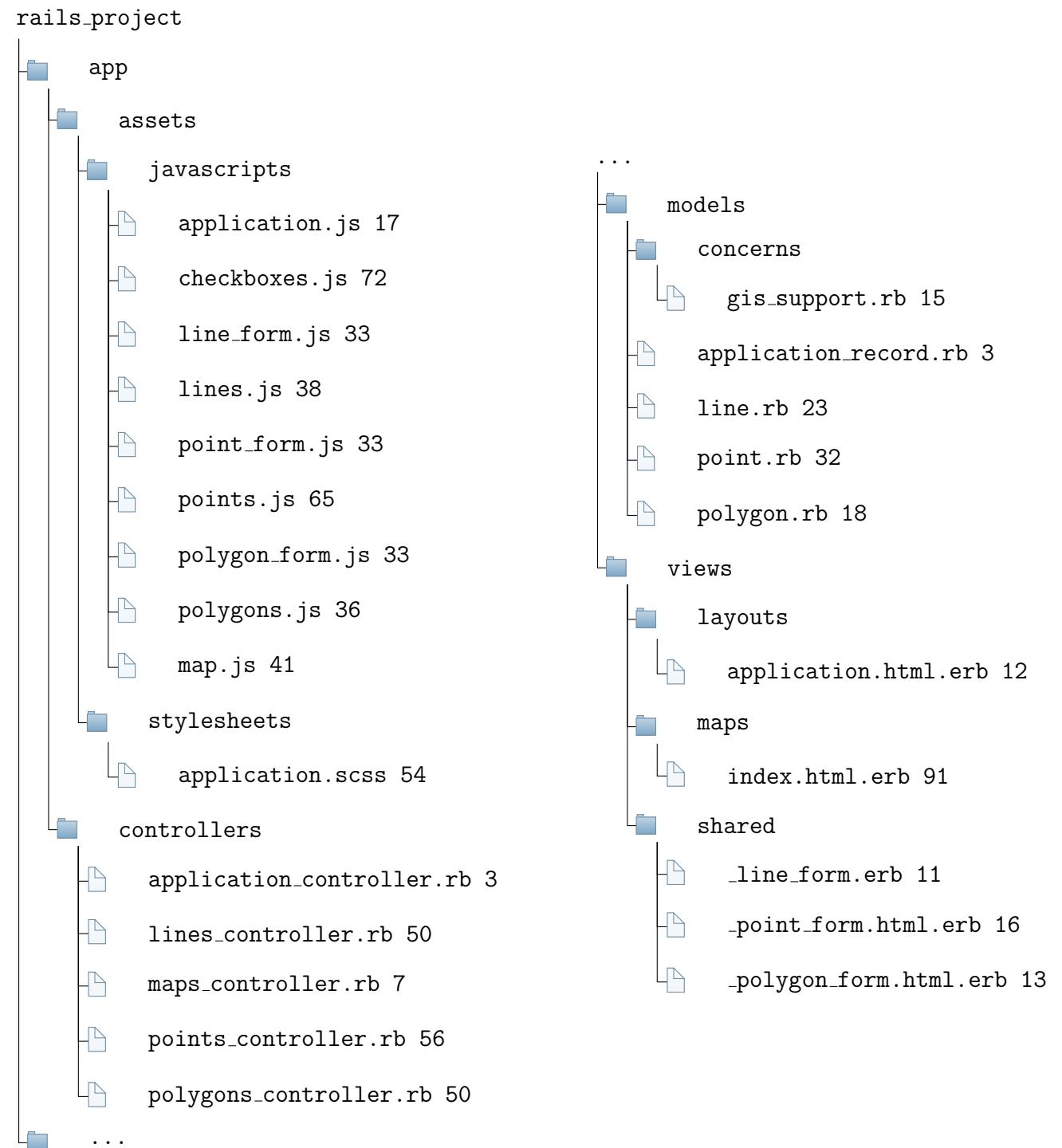
    property :id
    property :name
    property :object_type
    property :object_class
    property :voivodeship
    property :county
    property :commune
    property :terc
    property :coordinates_text
    property :color
  end
end
```

#### 1.2.4 Podsumowanie

Ruby on Rails posiada zdecydowanie najwięcej gotowych narzędzi do pracy z danymi geograficznymi z rozpatrywanych frameworków. Tworzenie aplikacji dzięki temu trwa mniej czasu. Roda mimo mniejszego wsparcia dla danych przestrzennych w przeciwieństwie do Hanami pozwala na ich używanie bez rozszerzania funkcjonalności frameworka. Hanami pozwala zbudować prosty system GIS wykorzystujący podstawowe typy przestrzenne dostępne w bazie danych PostgreSQL, ale implementacja systemu informacji geograficznej wykorzystującego inną bazę danych za pomocą frameworku Hanami wymaga od programisty znacznie więcej pracy i większych umiejętności programistycznych aby rozszerzyć możliwości frameworka.

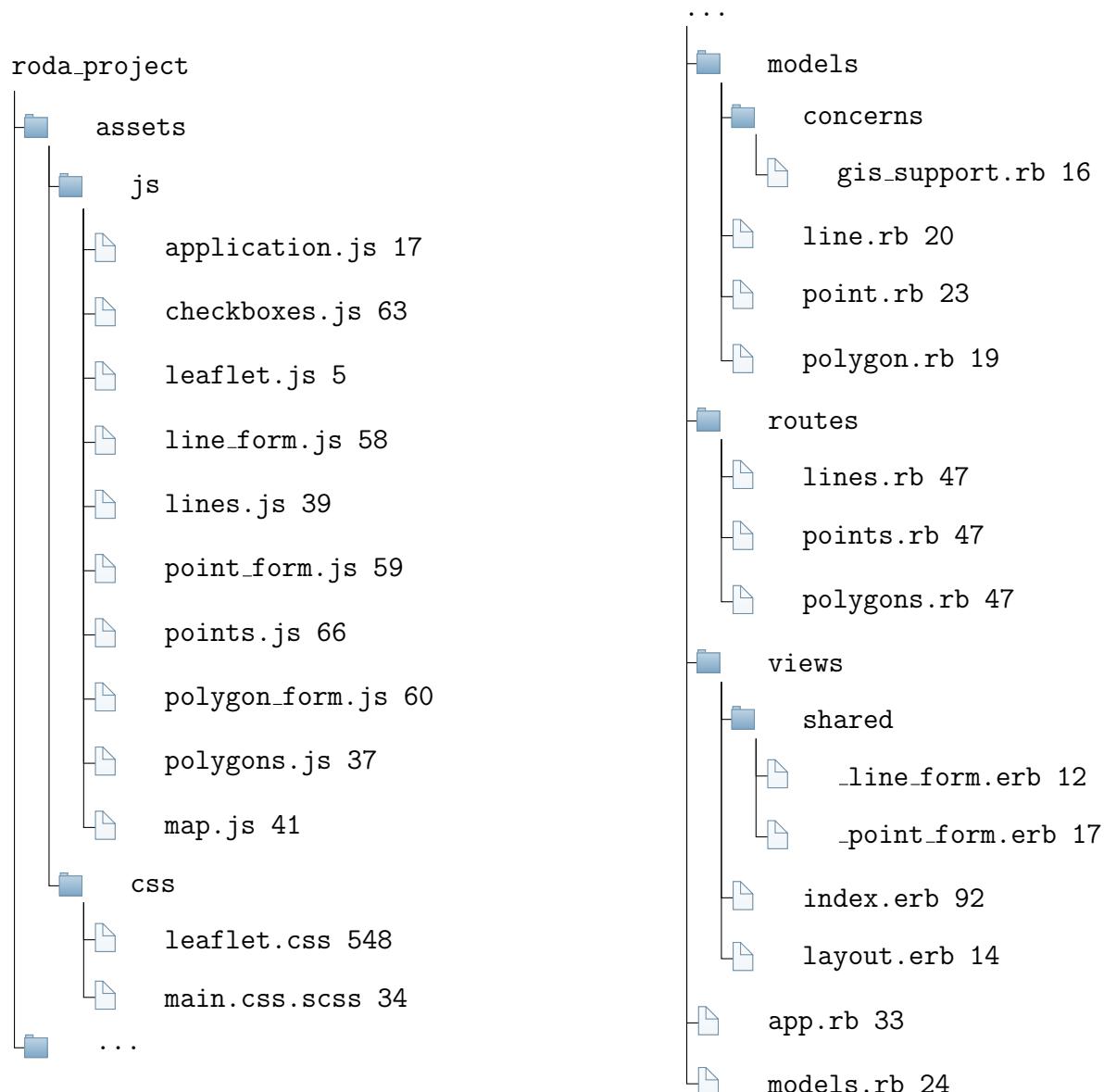
### 1.3 Porównanie struktur wykonanych projektów

Struktura zaimplementowanego projektu przy pomocy Ruby on Rails bezpośrednio odzwierciedla architekturę frameworku. W projekcie mamy jasny podział na Model-Widok-Kontroler.



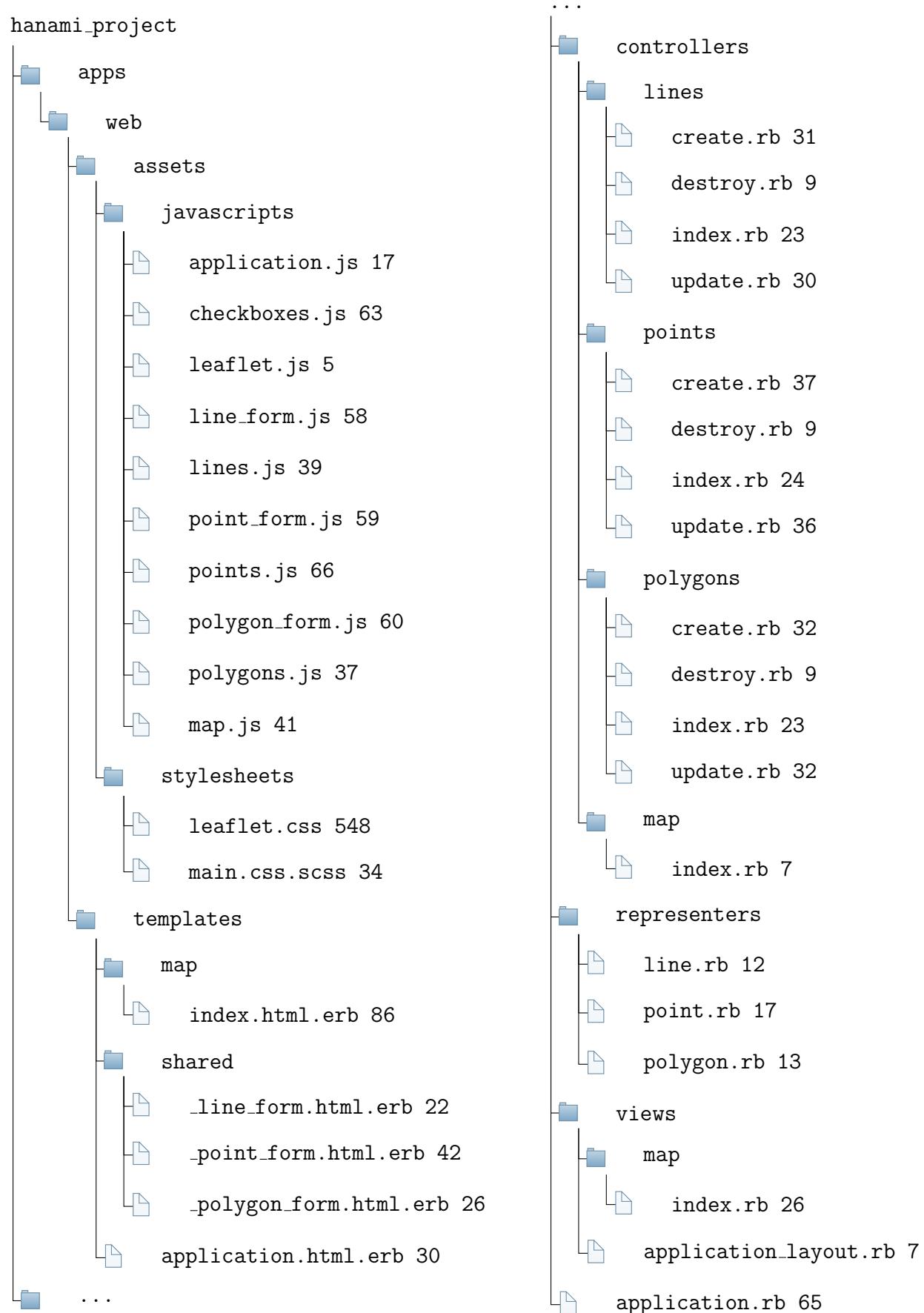
Rysunek 1.4 Struktura projektu Ruby on Rails

Filozofia prostoty w frameworku Roda ma swoje odzwierciedlenie w strukturze projektu, która nie posiada za wiele rozgałęzień.

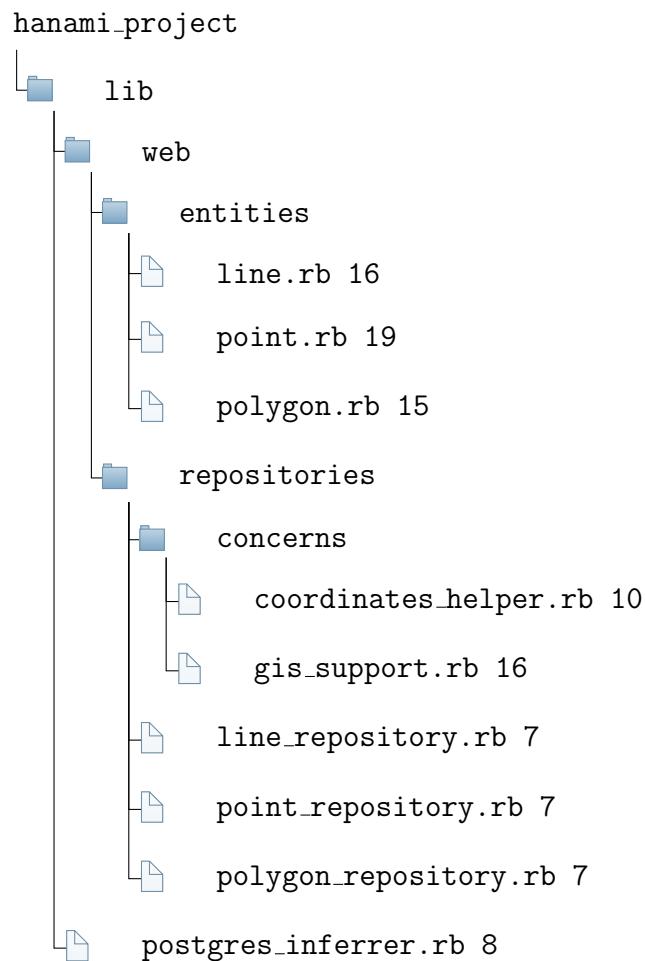


Rysunek 1.5 Struktura projektu Roda

Dodatkowe warstwy abstrakcji w architekturze projektu Hanami przekładają się na bardziej rozbudowaną strukturę projektu.

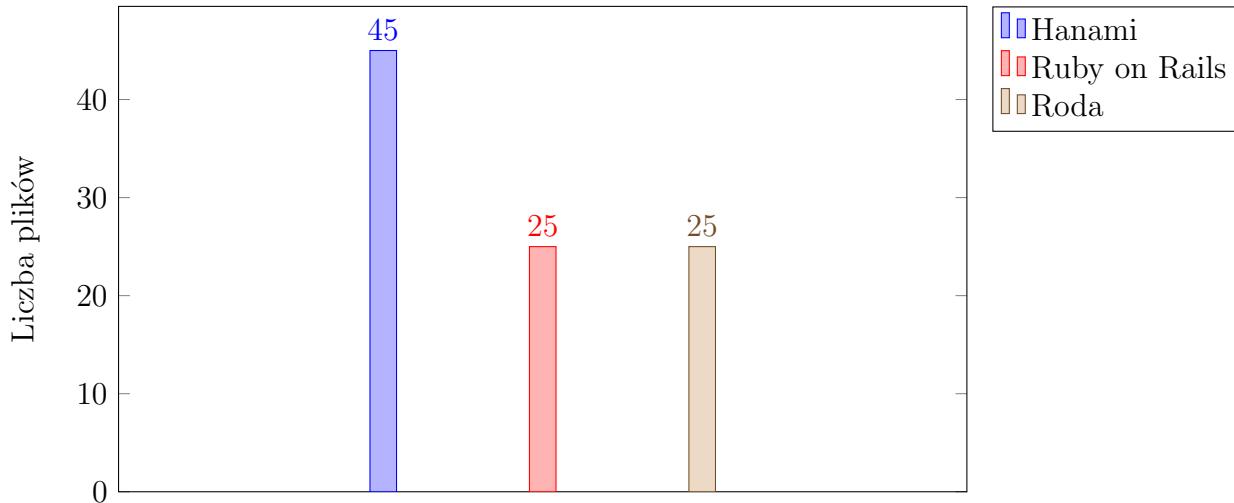


Rysunek 1.6 Struktura projektu Hanami - część I

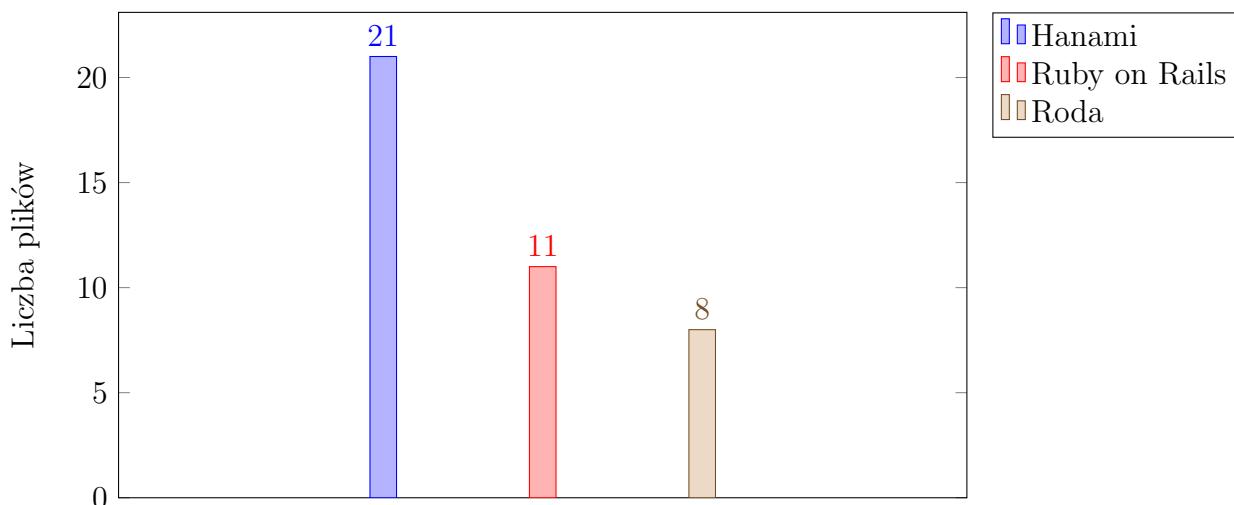


Rysunek 1.7 Struktura projektu Hanami - część II

Na rysunkach 1.8 i 1.9 można zauważyc, że projekt stworzony przy pomocy Hanami zawiera znacznie więcej plików i folderów od dwóch pozostałych projektów. Dodatkowe warstwy abstrakcji w architekturze frameworka przekładają się na bardziej skomplikowaną strukturę projektu. Projekty zrealizowane przy pomocy Ruby on Rails i Roda zawierają dokładnie tyle samo plików, ale Roda ma nieco mniej folderów. Na rysunku 1.5 można zauważyc, że struktura projektu Roda jest bardziej płaska niż struktura Ruby on Rails zaprezentowana na rysunku 1.4.

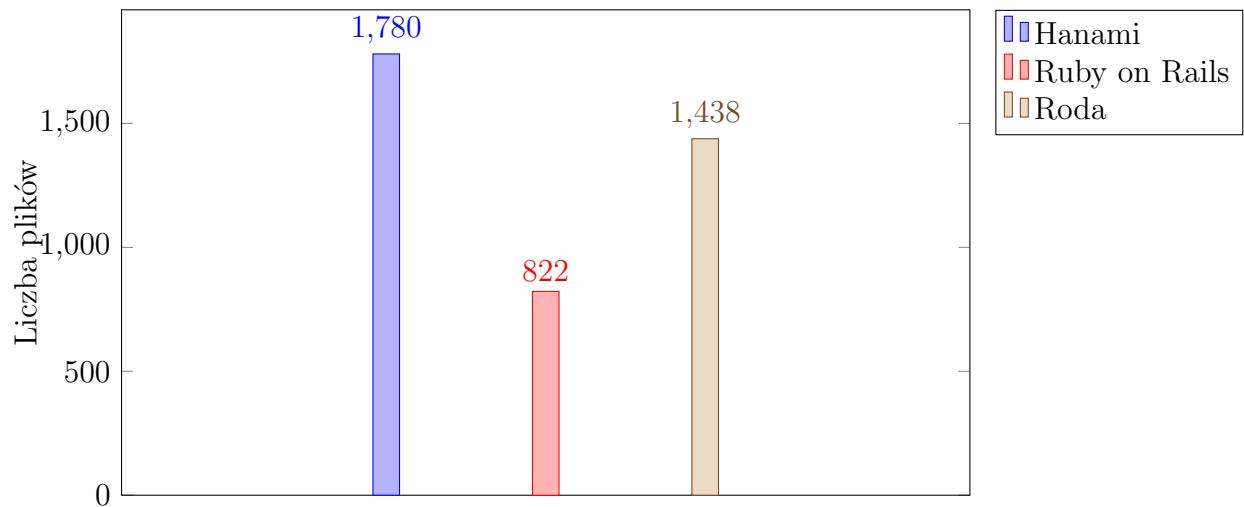


Rysunek 1.8 Liczba plików w projekcie w zależności od użytego frameworku



Rysunek 1.9 Liczba folderów w projekcie w zależności od użytego frameworku

Na poniższym wykresie można zauważyc, że projekt Ruby on Rails zawiera najmniej linii kodu. Framework posiada wiele gotowych, wbudowanych funkcjonalności co ułatwia pracę programistę. Projekt Hanami zawiera najwięcej linii kodu co spowodowane jest skomplikowaną architekturą aplikacji, każda kolejna warstwa abstrakcji narzuca na programistę obowiązek napisania dodatkowego kodu realizującego komunikacje między warstwami.



Rysunek 1.10 Liczba linii kodu w projekcie w zależności od użytego frameworku

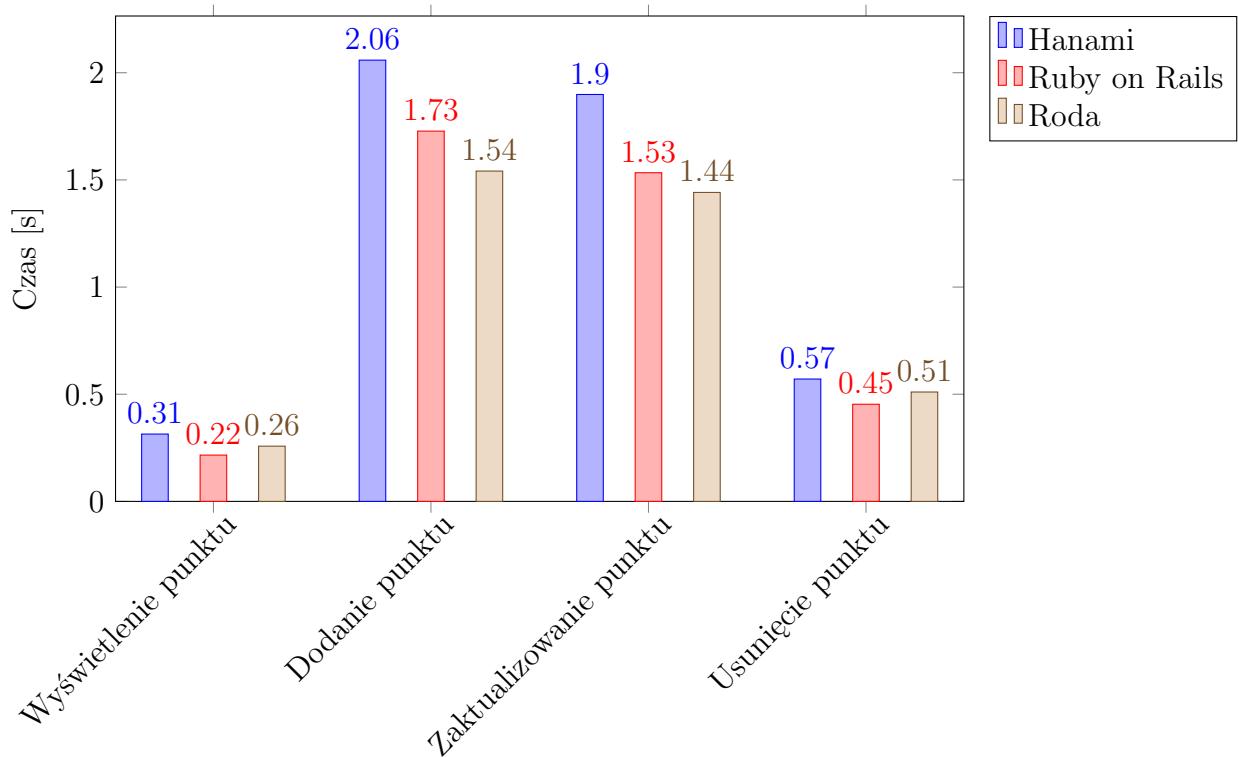
## 1.4 Porównanie wydajności zaimplementowanych aplikacji

Wszystkie wyniki badań wydajnościowych zostały uśrednione po 30 przebiegach.

### 1.4.1 Interfejs użytkownika

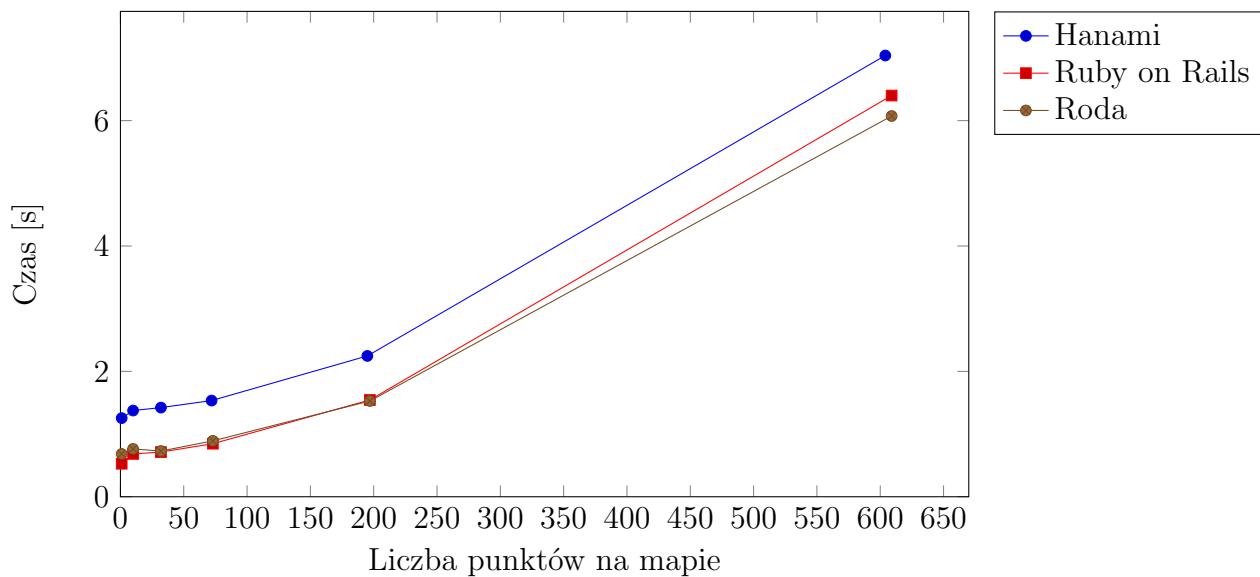
Badania na poziomie interfejsu użytkownika wykonano za pomocą Capybary, Selenium i Ruby Benchmark. Każdy scenariusz obejmował kroki, które użytkownik musi wykonać w aplikacji aby wykonać daną akcję. Badanie danego scenariusza, kończono w chwili pojawienia się efektów wykonanej akcji w interfejsie graficznym.

#### Punkty



Rysunek 1.11 Czas wykonania operacji CRUD dla punktu

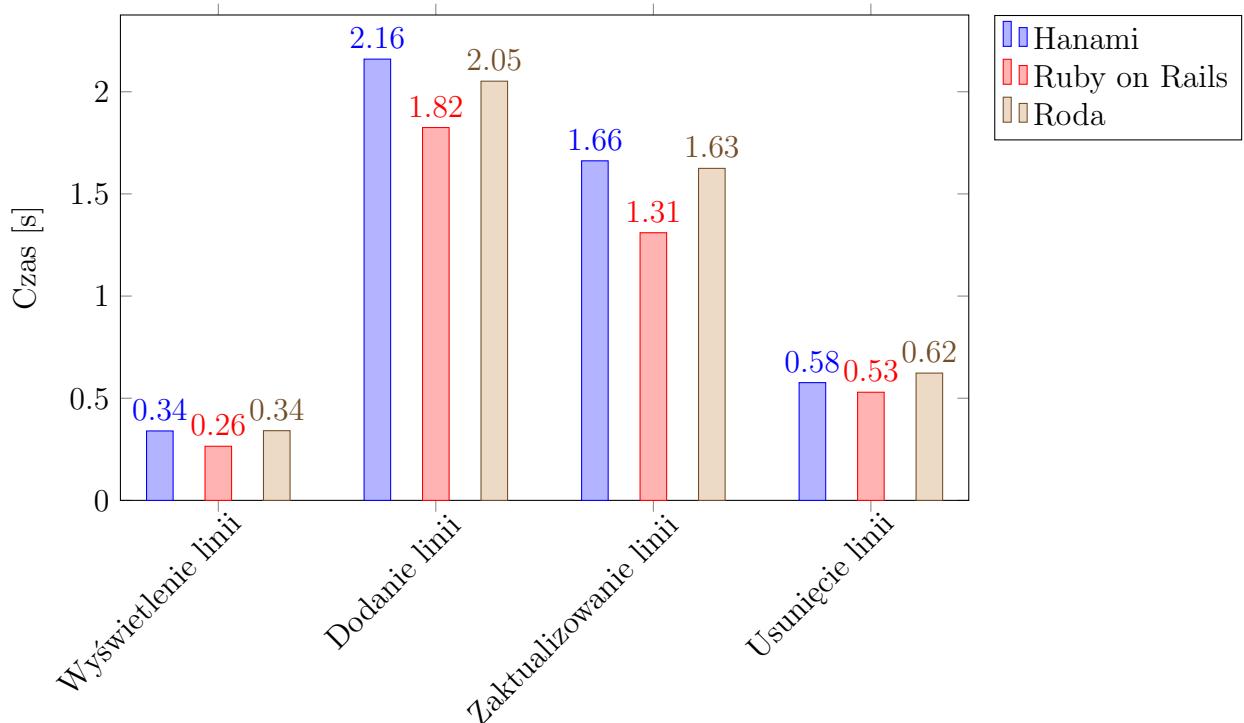
Framework Hanami okazał się najwolniejszy przy pojedynczych operacjach na obiektach typu punkt. Roda jest bardziej wydajna w akcjach dodawania i aktualizowania punktów, natomiast Ruby on Rails ma przewagę przy wyświetlaniu i usuwaniu punktu.



Rysunek 1.12 Czas ładowania widoku w zależności od liczby punktów

Na powyższym wykresie można zauważyć, że przewaga Rody i Ruby on Rails nad Hanami jest stała i nie zależy od liczby punktów.

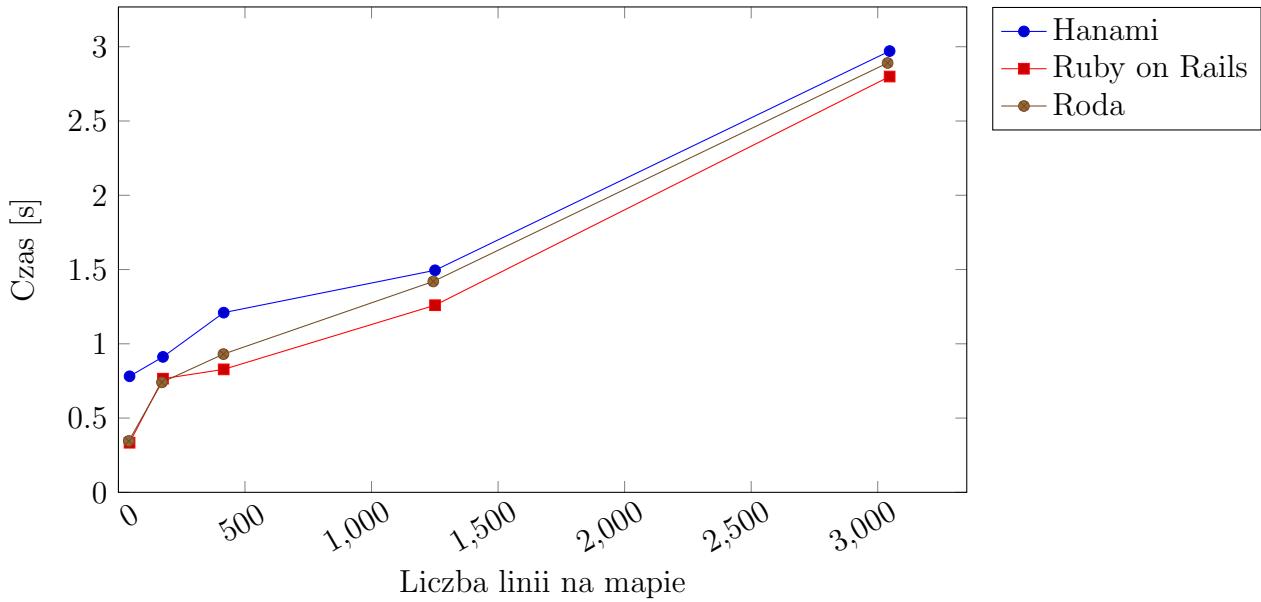
### Linie



Rysunek 1.13 Czas wykonania operacji CRUD dla linii

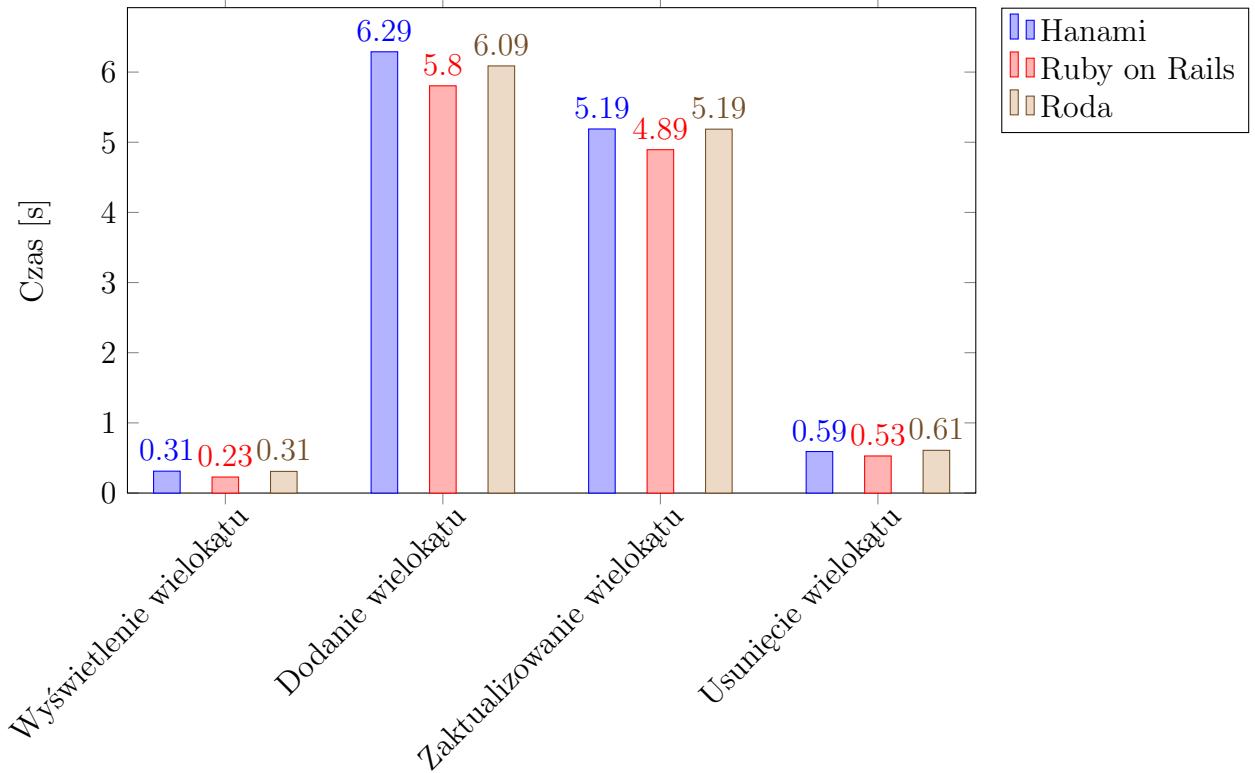
W operacjach na liniach można zauważyć, że Roda nie radzi sobie już tak dobrze jak z obiektami punktowymi. Najlepiej wypada Ruby on Rails, dwa pozostałe frameworki uzyskały porównywalne wyniki.

## Wielokąty



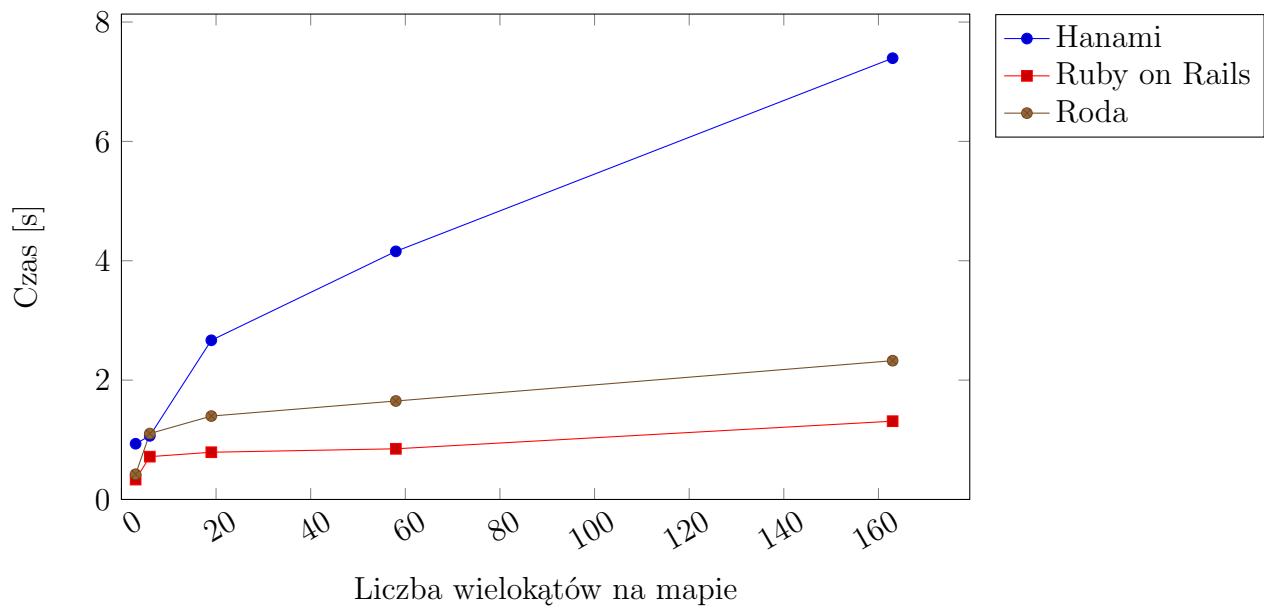
Rysunek 1.14 Czas ładowania widoku w zależności od liczby linii

Przy ładowaniu widoku zawierającego mniej niż 200 linii Roda i Ruby on Rails dają podobne rezultaty. Przy większej liczbie linii Ruby on Rails zyskuje przewagę. Hanami wypada najgorzej, ale różnica między nim, a dwoma pozostałymi frameworkami spada wraz ze wzrostem liczby linii.



Rysunek 1.15 Czas wykonania operacji CRUD dla wielokątów

Podczas operacji na wielokątach Roda i Hanami wypadły porównywalnie, obydwa frameworki dały gorsze rezultaty niż Ruby on Rails.



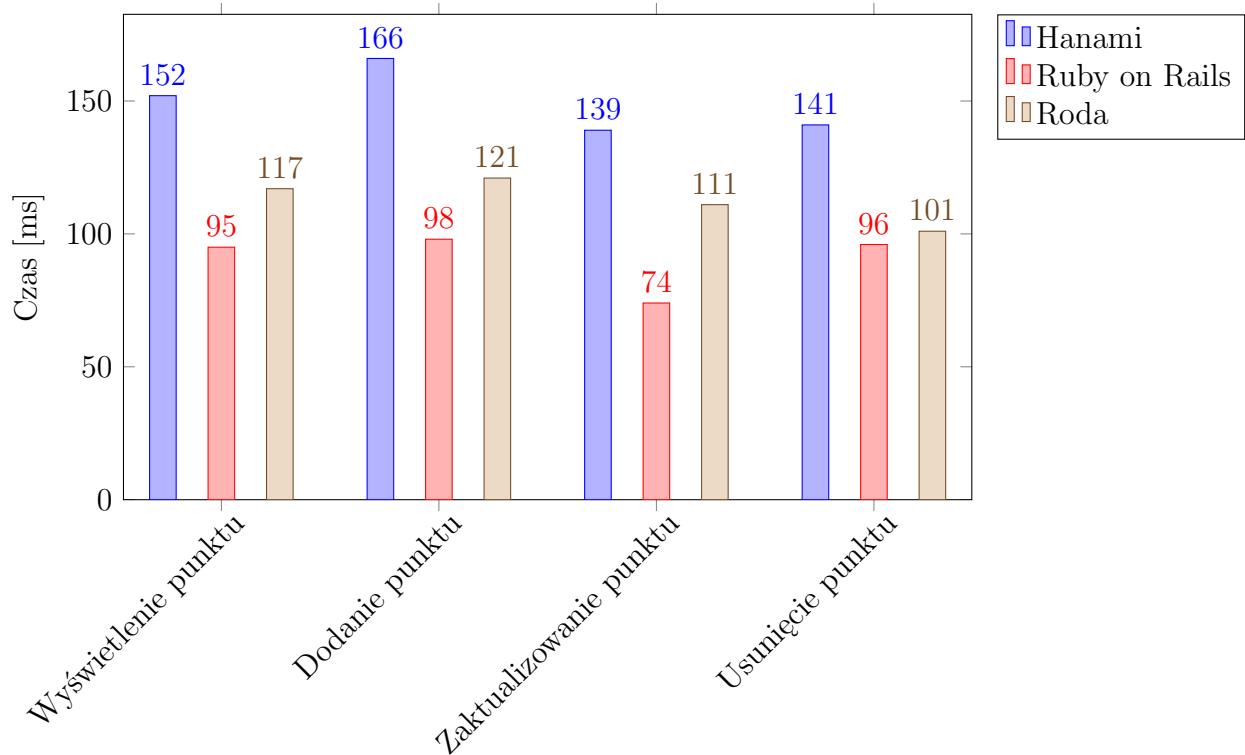
Rysunek 1.16 Czas ładowania widoku w zależności od liczby wielokątów

Podczas ładowania większej liczby wielokątów, najszybciej zwiększał się czas wykonania akcji dla frameworku Hanami. Drugie miejsce zajęła Roda, najlepsze wyniki zarejestrowano z użyciem Ruby on Rails.

### 1.4.2 Zapytania HTTP

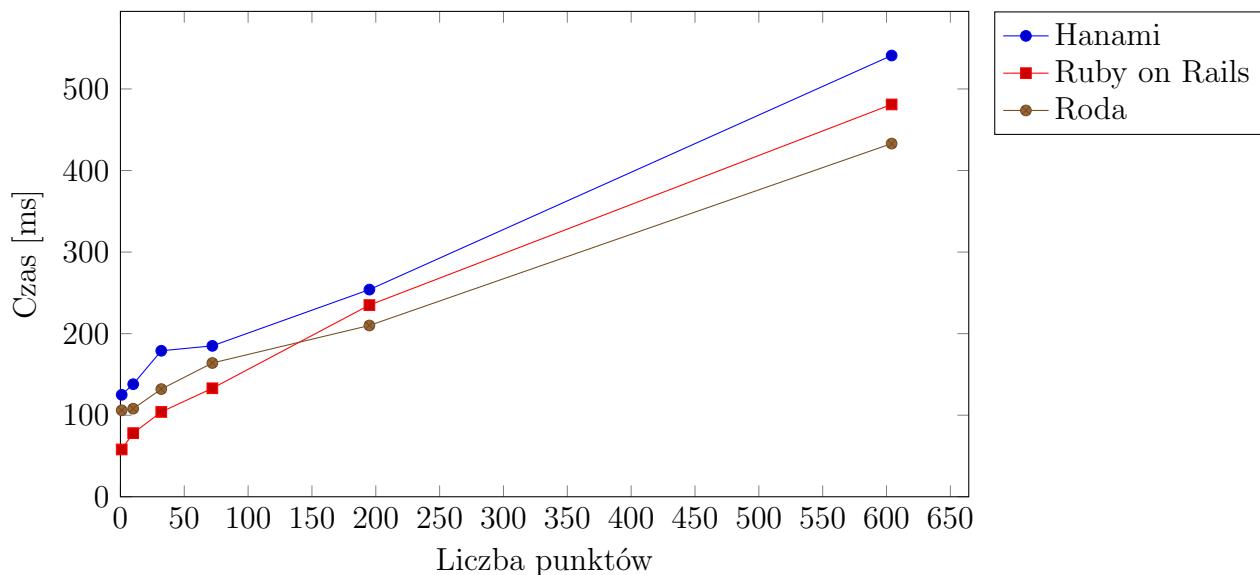
Wszystkie badania w tym rozdziale wykonano przy pomocy aplikacji JMeter.

#### Punkty



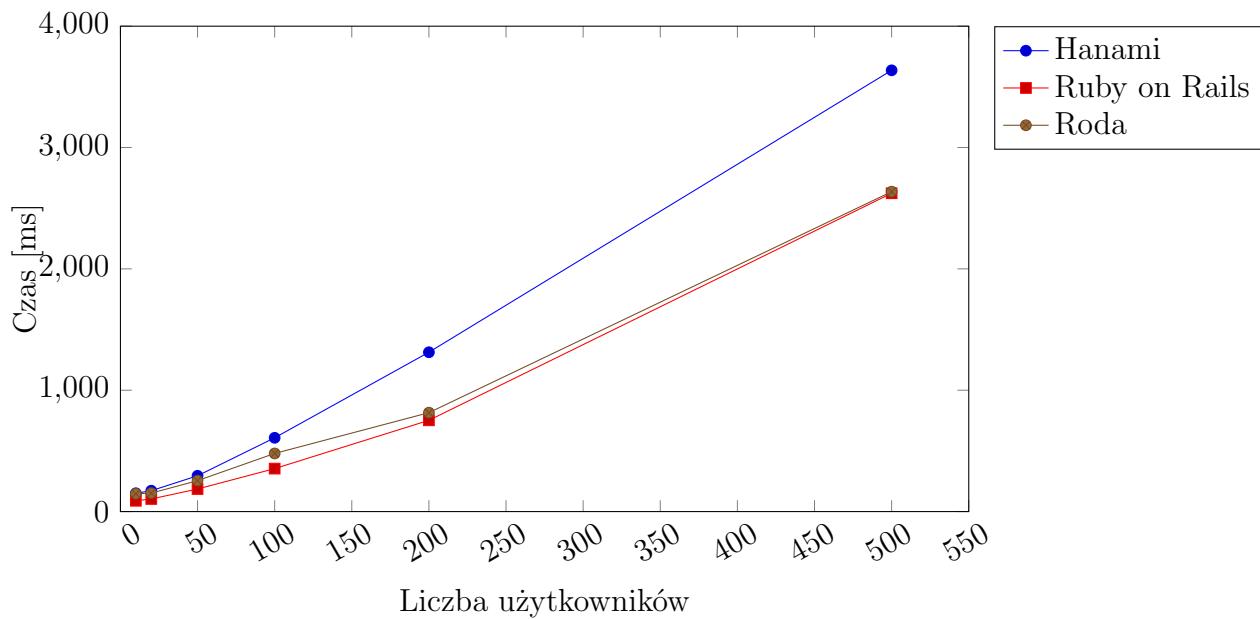
Rysunek 1.17 Czas wykonania zapytania HTTP dla punktu

Czas pojedynczych operacji HTTP dla obiektów punktowych jednoznacznie pokazuje, że najbardziej wydajna jest aplikacja napisana z użyciem Ruby on Rails, następnie Roda, a najmniej wydajny okazał się framework Hanami.



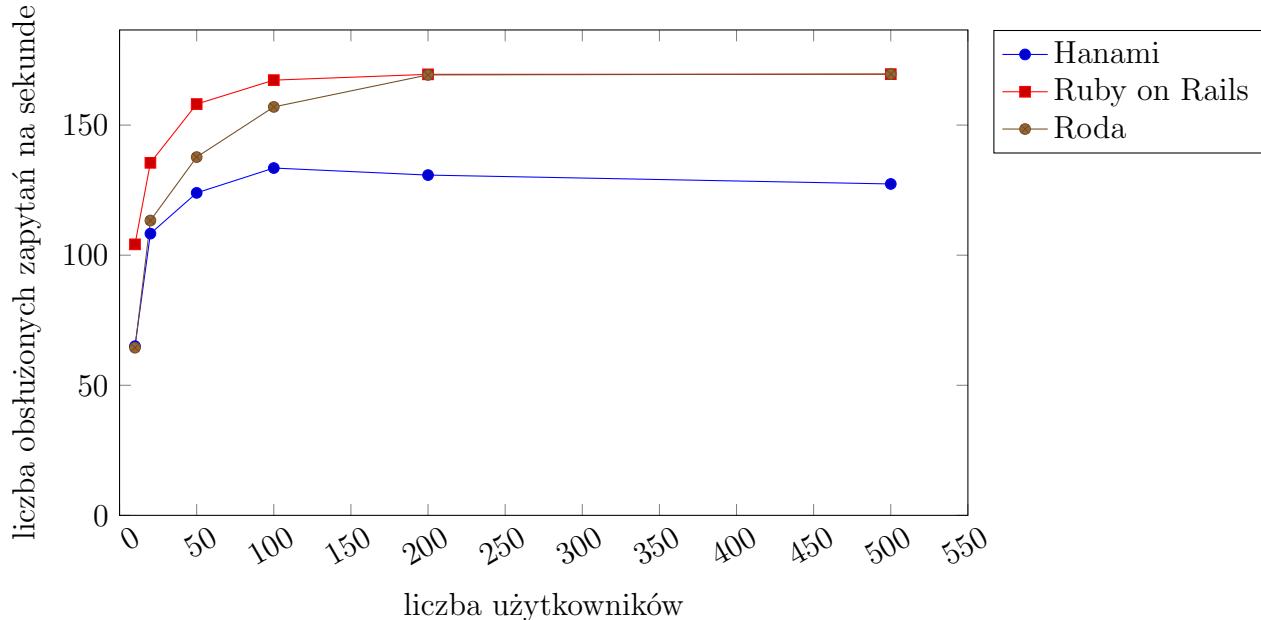
Rysunek 1.18 Czas ładowania danych w zależności od liczby punktów

Hanami bez względu na liczbę punktów wypada gorzej od dwóch pozostałych framework'ów. Jeśli punktów jest mniej niż 40, Ruby on Rails wczytuje dane nieco szybciej niż Roda. Powyżej 70 punktów, lepsze wyniki uzyskano dla frameworku Roda.



Rysunek 1.19 Czas ładowania danych w zależności od liczby użytkowników

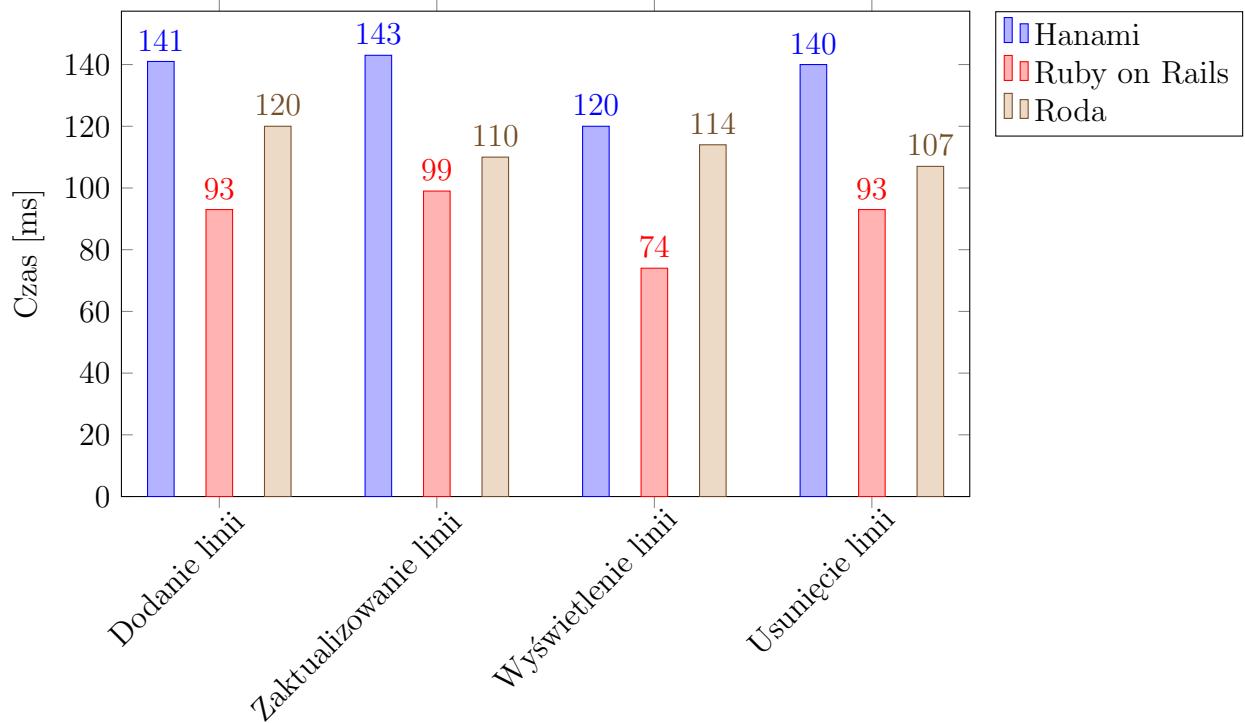
Przy zwiększającej się liczbie użytkowników aplikacja wykorzystująca framework Hanami szybciej zwiększa swój czas odpowiedzi. Pozostałe dwie aplikacje zachowują się podobnie.



Rysunek 1.20 liczba obsłużonych zapytań na sekundę w zależności od liczby użytkowników

Początkowo Ruby on Rails obsługuje najwięcej użytkowników. Powyżej 200 użytkowników, Roda wyrównuje wynik framework'u Ruby on Rails - 169 użytkowników na sekundę.

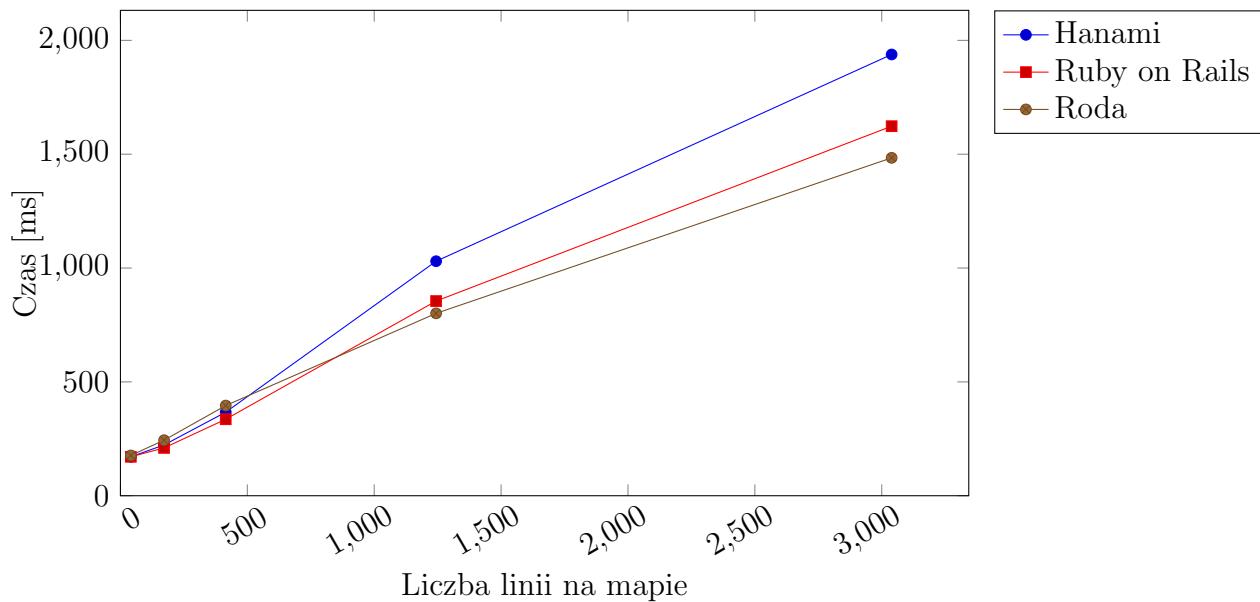
### Linie



Rysunek 1.21 Czas wykonania operacji CRUD dla linii

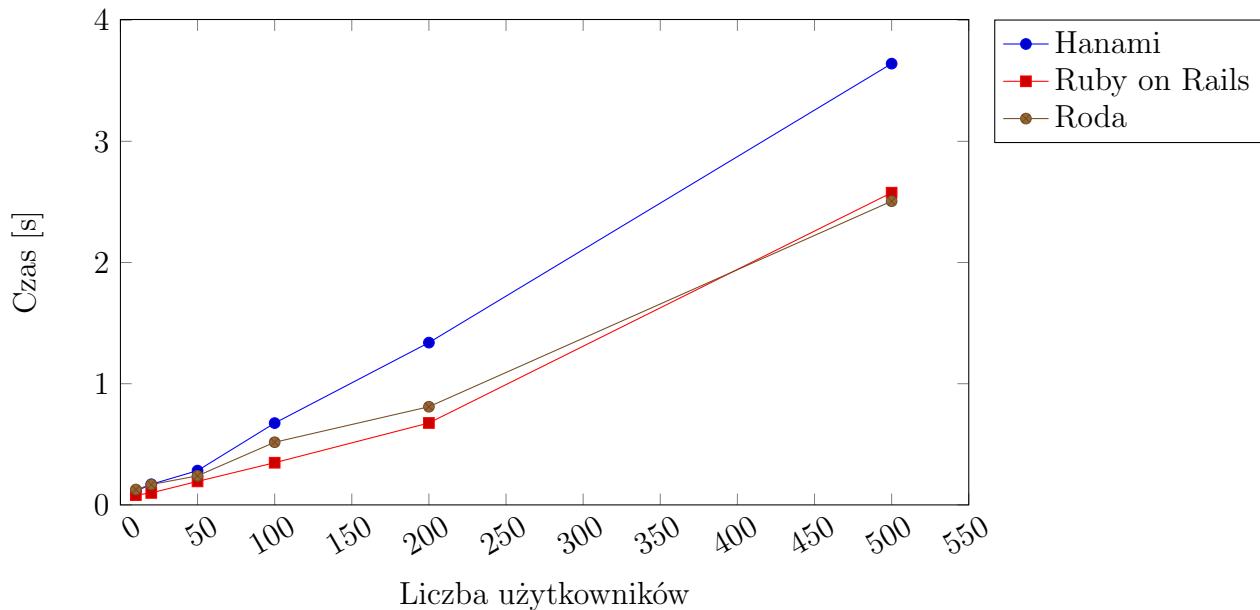
Tak samo jak dla punktów, najkrótszy czas operacji na liniach otrzymano w badaniach aplikacji opartej na frameworku Ruby on Rails. Następne miejsce zajął framework Roda.

Najdłuższy czas odpowiedzi otrzymano wykorzystując Hanami.



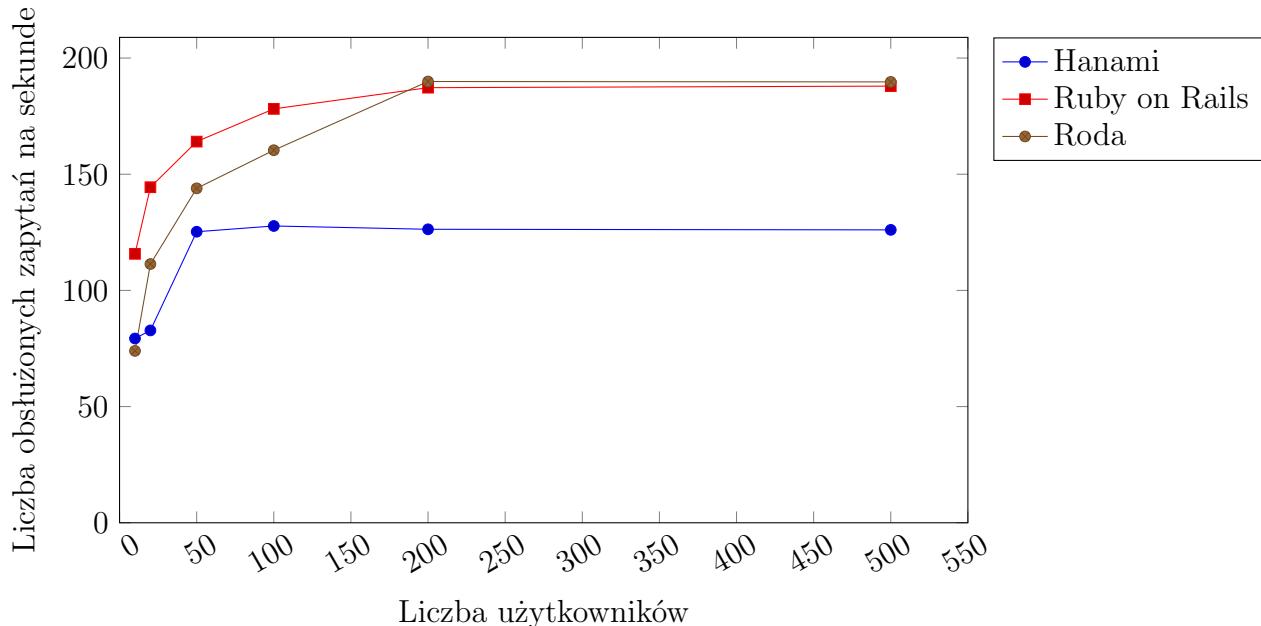
Rysunek 1.22 Czas ładowania danych w zależności od liczby linii

Poniżej 500 linii na mapie wszystkie 3 aplikacje odpowiadały w podobnym czasie. Przy większej liczbie linii, czas odpowiedzi Hanami rósł szybciej niż pozostałych dwóch frameworków. Najkrótszy czas odpowiedzi dla danych liczących ponad 1000 linii zanotowano dla aplikacji wykorzystującej framework Roda.



Rysunek 1.23 Czas ładowania danych w zależności od liczby użytkowników

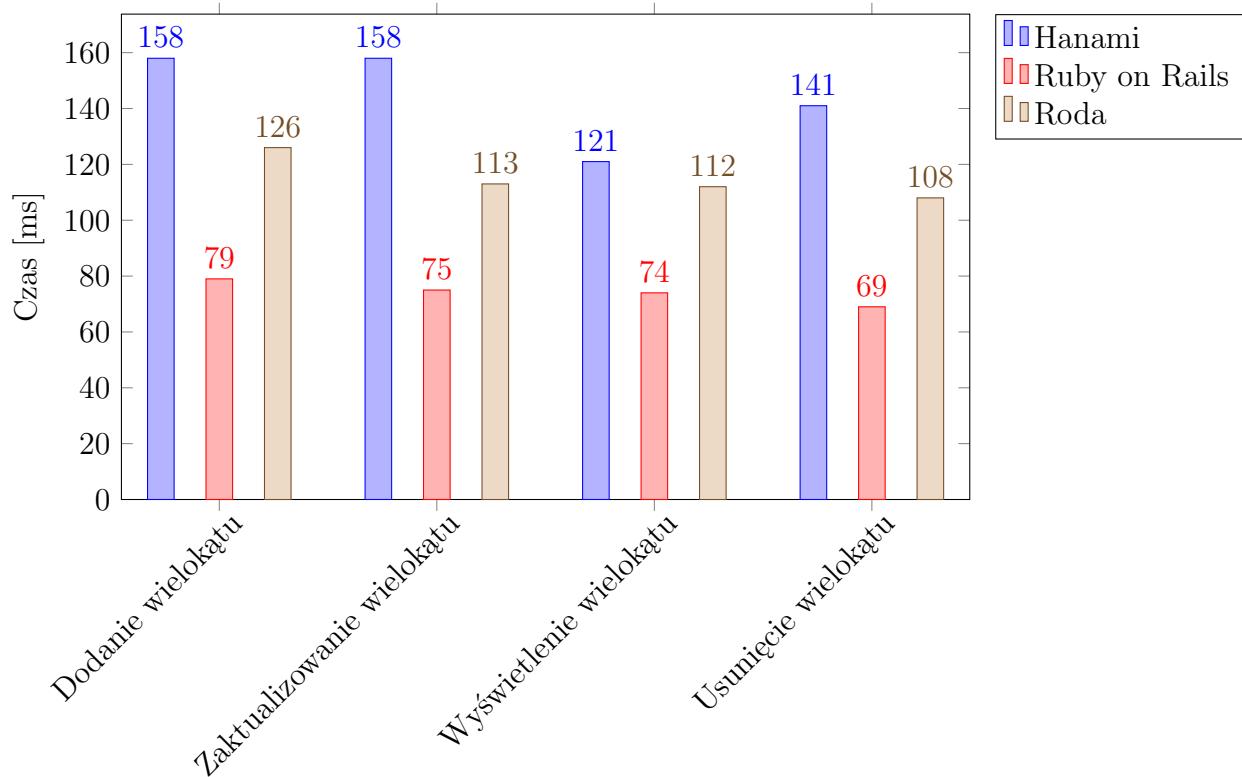
Wraz ze wzrostem liczby użytkowników czas odpowiedzi aplikacji zbudowanej w oparciu o framework Hanami, rósł szybciej niż pozostałych dwóch aplikacji.



Rysunek 1.24 Liczba obsłużonych zapytań na sekundę w zależności od liczby użytkowników

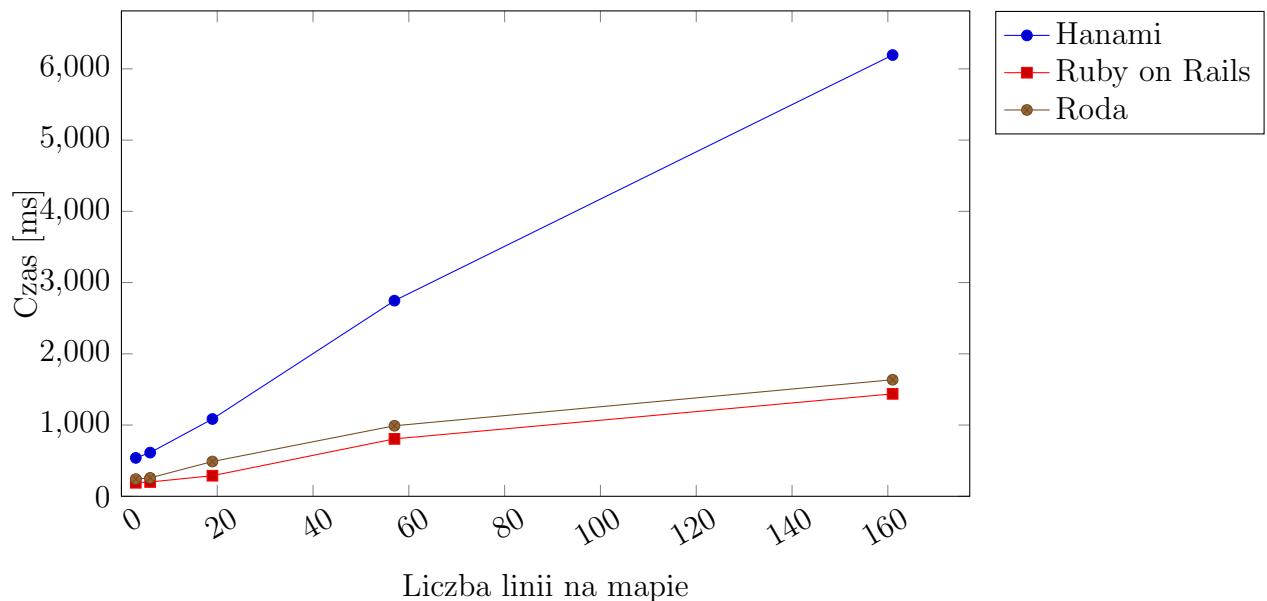
Powyżej 200 użytkowników jednocześnie korzystających z aplikacji, Roda i Ruby on Rails w ciągu sekundy obsługiwały około 190 użytkowników na sekundę. Przy mniejszej liczbie użytkowników Ruby on Rails obsługiwał więcej użytkowników w ciągu sekundy od Roda. Hanami w ciągu całego badania prezentował się znacznie gorzej od dwóch pozostałych framework'ów i maksymalnie w ciągu sekundy odpowiedział na zapytanie 115 użytkowników.

## Wielokąty



Rysunek 1.25 Czas wykonania operacji CRUD dla wielokątów

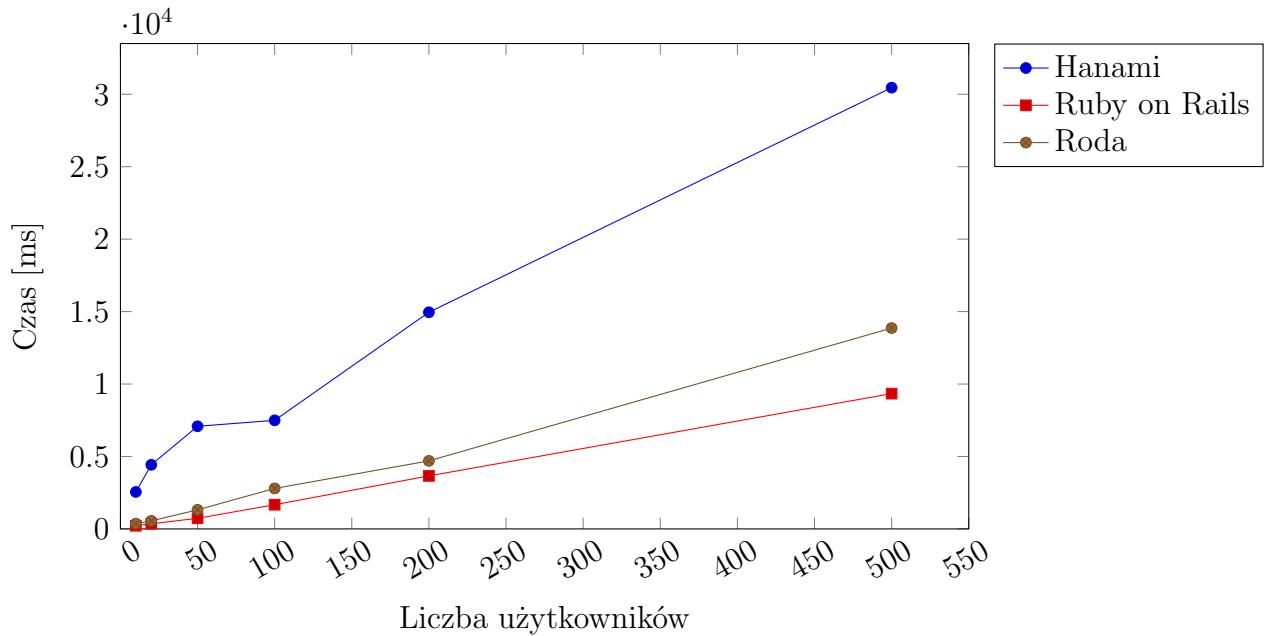
W podstawowych operacjach na wielokątach najlepsze czasy uzyskał framework Ruby on Rails, które były średnio dwukrotnie lepsze od wyników najwolniejszego frameworku Hanami.



Rysunek 1.26 Czas ładowania danych w zależności od liczby wielokątów

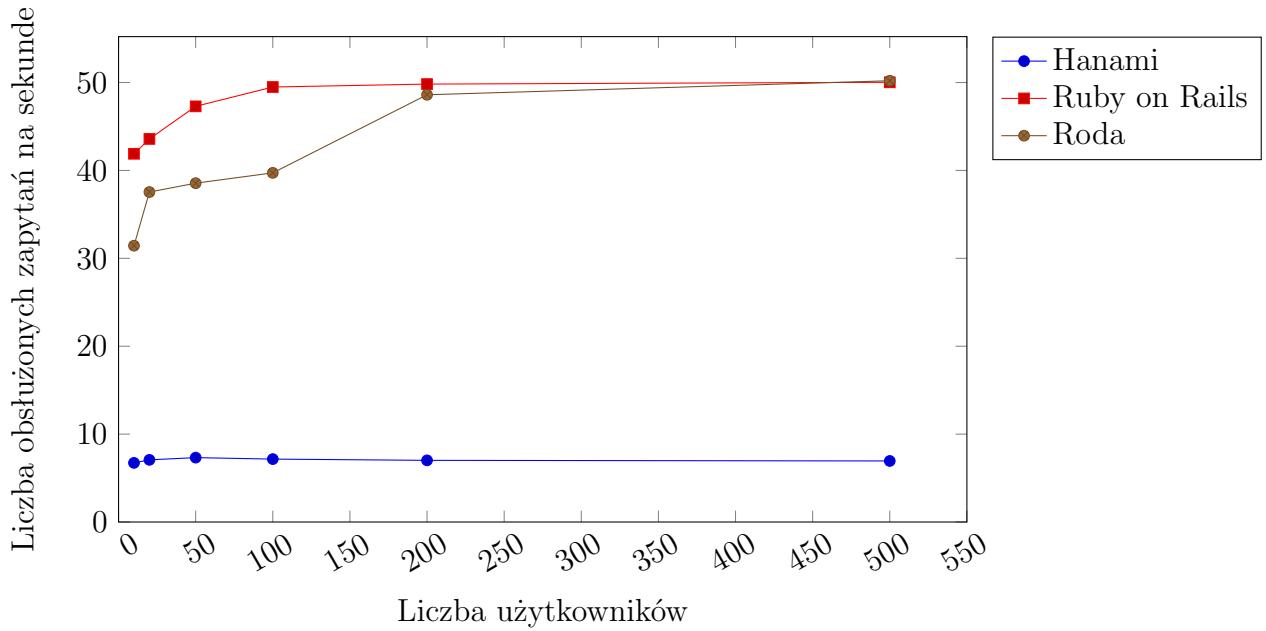
Czas odpowiedzi aplikacji zbudowanej w oparciu o Hanami znacznie szybciej rósł w po-

równaniu do dwóch pozostałych aplikacji. Ruby on Rails i Roda uzyskały zbliżone czasy z małą, stałą przewagą na korzyść Ruby on Rails.



Rysunek 1.27 Czas ładowania danych w zależności od liczby użytkowników

Dla danych typu wielokąt, Hanami znacznie wolniej obsługiwał zapytania przy wzrastającej liczbie użytkowników w porównaniu do pozostałych frameworków. Najlepiej z rosnącą liczbą użytkownik radził sobie framework Ruby on Rails.



Rysunek 1.28 Liczba obsłużonych zapytań na sekundę w zależności od liczby użytkowników

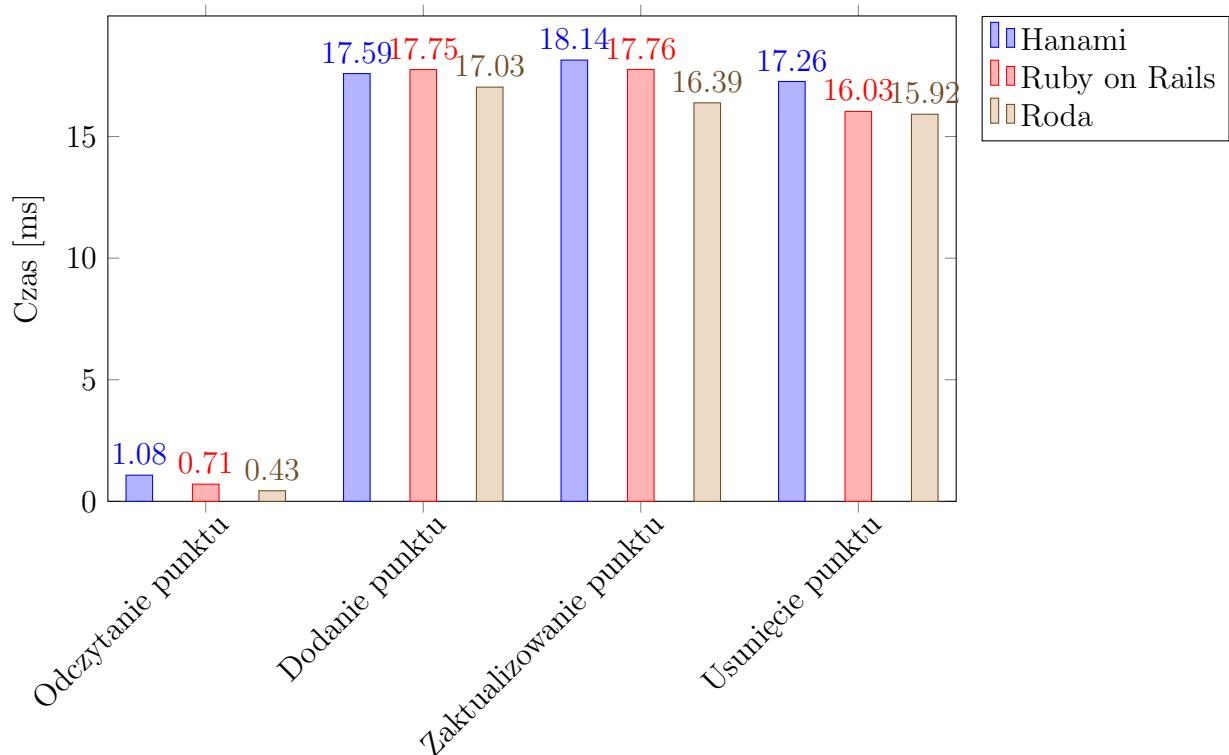
Hanami zdołał maksymalnie obsłużyć 7 użytkowników w ciągu sekundy i ta liczba nie zmieniała się przy rosnącej liczbie użytkowników. Ruby on Rails, podobnie jak dla

innych typów danych przy mniejszej liczbie użytkowników, mniej niż 200, obsługiwał więcej użytkowników niż Roda, jednak od 200 użytkowników jednocześnie korzystających z aplikacji, oba frameworki obsługiwały prawie 50 użytkowników w ciągu jednej sekundy.

### 1.4.3 Komunikacja z bazą danych

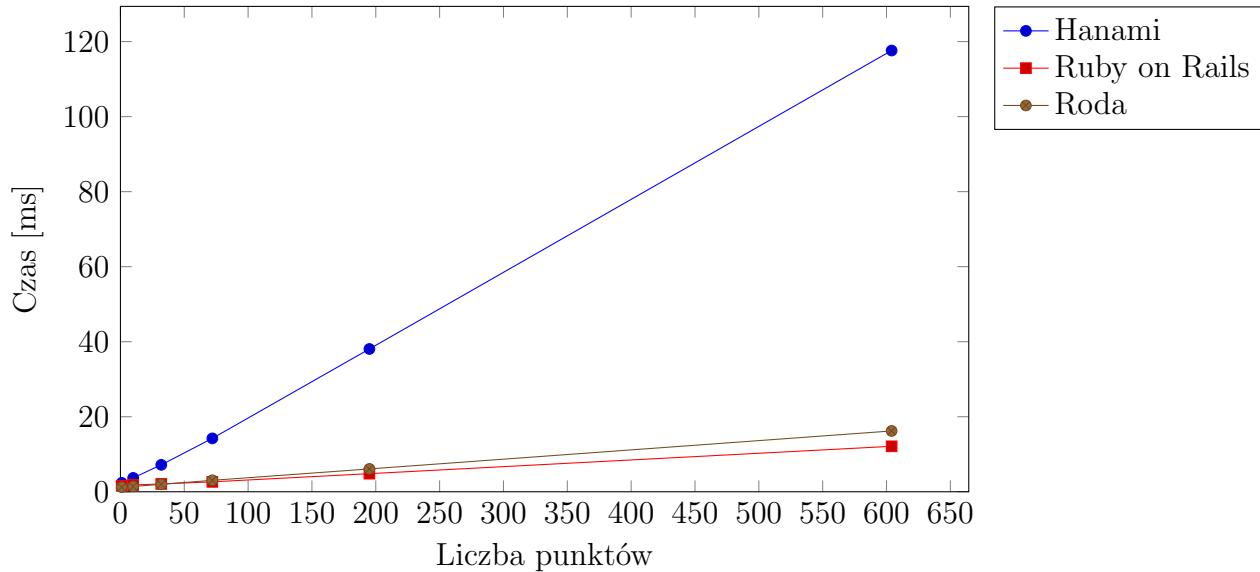
Badania wydajności komunikacji z bazą danych zrealizowano w warstwie modelu każdej z aplikacji. Czas wykonywania metod dla wybranych modeli zmierzono przy pomocy biblioteki Ruby Benchmark.

#### Punkty



Rysunek 1.29 Czas wykonania operacji CRUD dla punktu

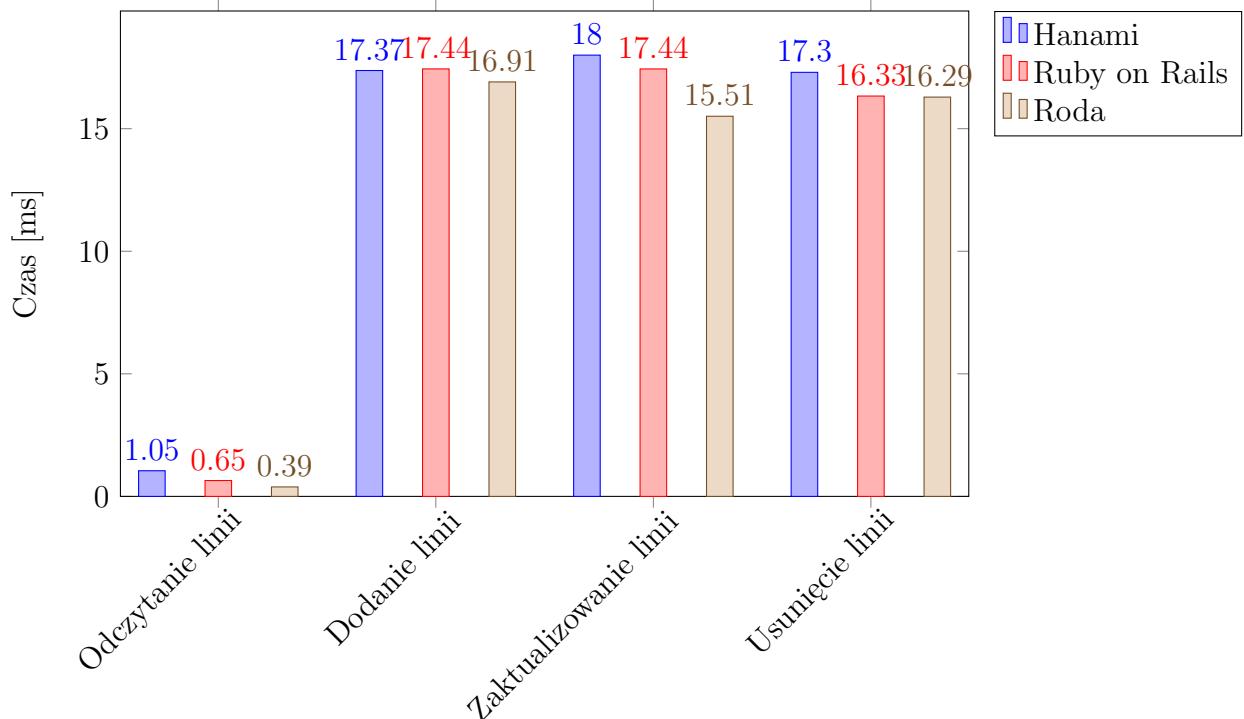
W pojedynczych operacjach w warstwie modelu czasy operacji na punktach były do siebie zbliżone dla 3 badanych frameworków. Najszybciej zadane operacje wykonał framework Roda.



Rysunek 1.30 Czas wczytywania danych z bazy w zależności od liczby punktów

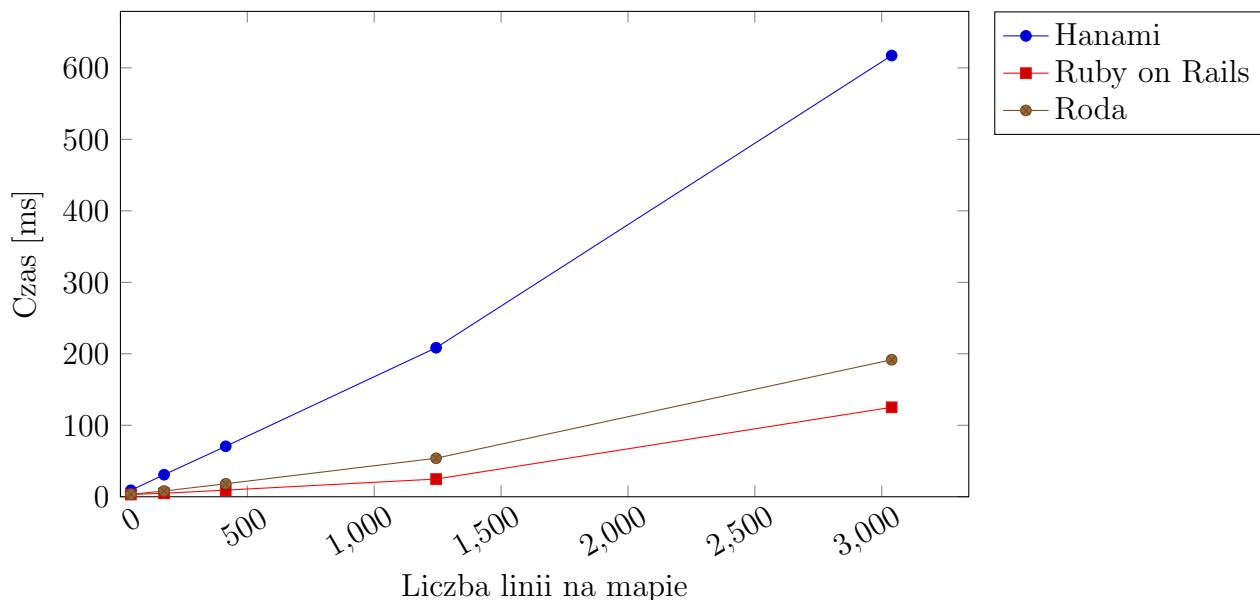
Przy rosnącej liczbie wczytywanych punktów Roda i Ruby on Rails uzyskały podobne wyniki, które były znacznie lepsze od Hanami.

### Linie



Rysunek 1.31 Czas wykonania operacji CRUD dla linii

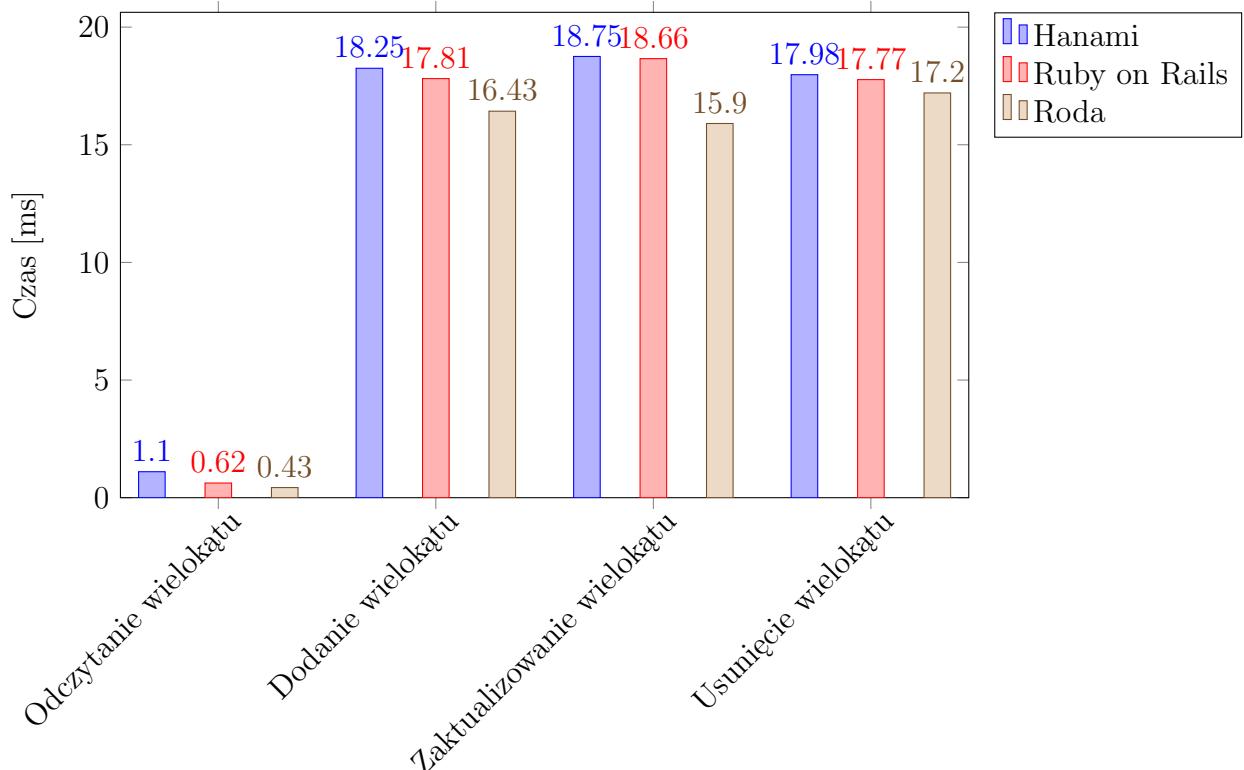
Wyniki dla danych liniowych są analogiczne jak dla punktów, Roda uzyskała najlepsze rezultaty, a najgorsze wyniki uzyskano dla Hanami.



Rysunek 1.32 Czas wczytywania danych z bazy w zależności od liczby linii

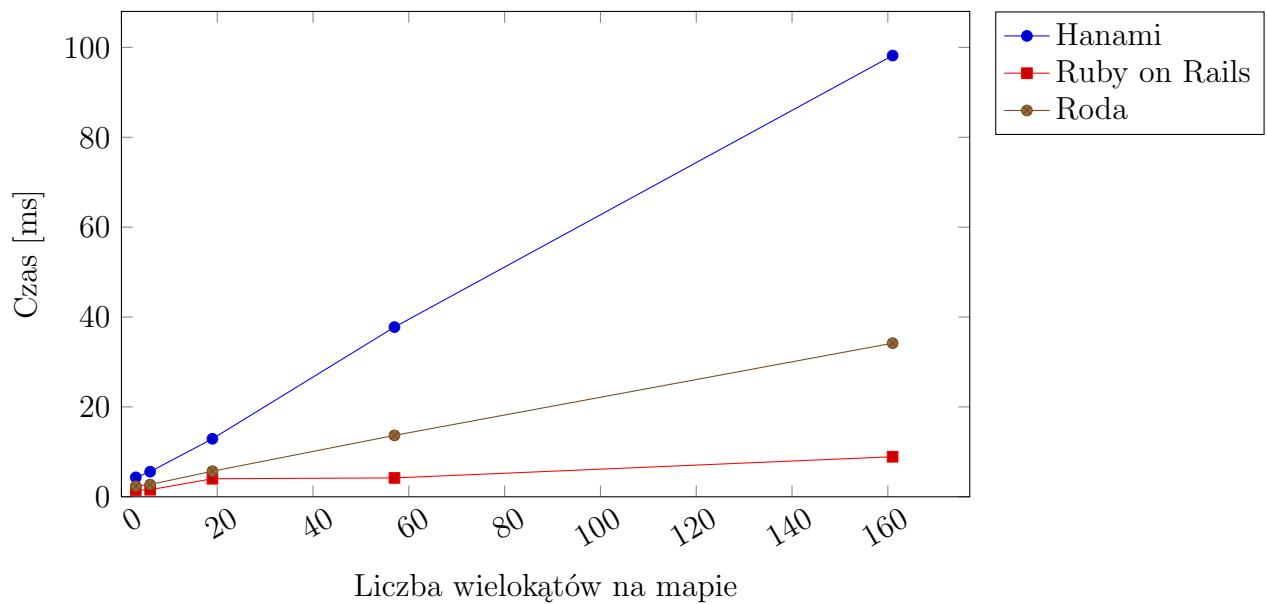
Przy danych liniowych nadal czas odpowiedzi Hanami rósł znacznie szybciej od dwóch pozostałych framework'ów. Można zauważyć, że różnica między Hanami, a Ruby on Rails jest bardziej zauważalna niż dla punktów, z korzyścią na stronę Ruby on Rails.

## Wielokąty



Rysunek 1.33 Czas wykonania operacji CRUD dla wielokątów

Badania dla wielokątów dały analogiczne wyniki jak dla dwóch poprzednich typów danych - Roda wykonywała operacje szybciej niż pozostałe frameworki. Najdłuższy czas otrzymano dla frameworku Hanami.



Rysunek 1.34 Czas wczytywania danych z bazy w zależności od liczby wielokątów

Podobnie jak dla pozostałych typów danych geograficznych, dla aplikacji wykorzystującej framework Hanami odnotowano znacznie gorsze wyniki dla rosnącej liczby wczytywanych wielokątów niż dla pozostałych dwóch aplikacji. Jednak dla wielokątów różnica między frameworkiem Roda, a Ruby on Rails jest bardziej zauważalna.

# Rozdział 2

## Podsumowanie

### 2.1 Analiza wyników badań

Badania wykazały, że przy wyborze frameworku internetowego języka Ruby do budowy systemu informacji geograficznej najgorszym wyborem jest framework Hanami. Jego jedynym atutem może być rozbudowana architektura, która ułatwia organizację kodu źródłowego w rozbudowanych projektach. Jednak w prostszych projektach, wiele warstw abstrakcji wymaga od programistów napisania więcej kodu źródłowego niż w innych frameworkach do zaimplementowania takich samych funkcjonalności. Hanami jest znacznie mniej wydajny w każdej z badanych płaszczyzn - interfejsu użytkownika, zapytań HTTP, komunikacji z bazą danych. Dodatkowo aby użyć innej przestrzennej bazy danych niż PostgreSQL trzeba zaimplementować wsparcie dla nowych typów danych, co nie jest konieczne przy korzystaniu z innych frameworków.

Pozostałe dwa frameworki Ruby on Rails i Roda podobnie wypadły w badaniach wydajnościowych. Roda w operacjach CRUD w warstwie modelu była szybsza dla wszystkich typów danych od frameworku Ruby on Rails, jednak przy wczytywaniu bardziej skomplikowanych typów - linii i wielokątów - dla Ruby on Rails zarejestrowano niższe czasy przy dużej liczbie danych. Na poziomie zapytań HTTP oraz interfejsu użytkownika, aplikacja wykorzystująca Ruby on Rails była bardziej wydajna zarówno dla operacji na pojedynczych obiektach geograficznych, jak i dla wczytywania większej liczby obiektów. Ruby on Rails posiada więcej bibliotek służących do pracy na danych geograficznych co ma przełożenie na mniejszą ilość napisanego kodu dzięki gotowym funkcjom. Struktura projektu Ruby on Rails i projektu Roda jest bardzo podobna, projekty zawierały tyle samo plików.

### 2.2 Realizacja celu projektu

Niniejsza praca udowodniła, że za pomocą frameworków internetowych języka Ruby z powodzeniem można stworzyć system informacji geograficznej. Najlepszym wyborem do budowy systemu GIS okazał się najbardziej popularny framework Ruby on Rails. Duża popularność przekłada się na duże wsparcie, wiele dostępnych bibliotek i narzędzi do pracy z danymi przestrzennymi. Ruby on Rails wymagał najmniej kodu źródłowego do zaimplementowania zaprojektowanego systemu, który wykazywał się największą wydajnością przy przetwarzaniu danych geograficznych.



# Literatura

- [1] *Dokumentacja biblioteki Capybara*, dostępna pod adresem:  
<https://github.com/teamcapybara/capybara>, aktualne na dzień 18.06.2017r.
- [2] *Dokumentacja biblioteki GeoRuby*, dostępna pod adresem:  
<https://github.com/nofxx/georuby>, aktualne na dzień 15.06.2017r.
- [3] *Dokumentacja biblioteki Google Maps JavaScript API*, dostępna pod adresem:  
<https://developers.google.com/maps/documentation/javascript/>, aktualne na dzień 15.06.2017r.
- [4] *Dokumentacja biblioteki Leaflet*, dostępna pod adresem:  
<http://leafletjs.com/reference-1.0.3.html>, aktualne na dzień 15.06.2017r.
- [5] *Dokumentacja biblioteki Rgeo*, dostępna pod adresem:  
<https://github.com/rgeo/rgeo>, aktualne na dzień 15.06.2017r.
- [6] *Dokumentacja biblioteki Ruby Benchmark*, dostępna pod adresem:  
<https://ruby-doc.org/stdlib-1.9.3/libdoc/benchmark/rdoc/Benchmark.html>, aktualne na dzień 18.06.2017r.
- [7] *Dokumentacja Hanami*, dostępna pod adresem:  
<http://hanamirb.org/guides/>, aktualne na dzień 08.03.2017r.
- [8] *Dokumentacja języka Ruby*, dostępna pod adresem:  
<https://www.ruby-lang.org/pl/documentation/>, aktualne na dzień 08.03.2017r.
- [9] *Dokumentacja JMeter*, dostępna pod adresem:  
<http://jmeter.apache.org/index.html>, aktualne na dzień 18.06.2017r.
- [10] *Dokumentacja MangoMap*, dostępna pod adresem:  
<http://help.mangomap.com/>, aktualne na dzień 22.04.2017r.
- [11] *Dokumentacja MySQL*, dostępna pod adresem:  
<https://dev.mysql.com/doc/refman/5.7/en/>, aktualne na dzień 15.06.2017r.
- [12] *Dokumentacja OpenStreetMap*, dostępna pod adresem:  
<http://wiki.openstreetmap.org/>, aktualne na dzień 08.03.2017r.
- [13] *Dokumentacja PostgreSQL*, dostępna pod adresem:  
<https://www.postgresql.org/docs/9.6/static/index.html>, aktualne na dzień 09.06.2017r.
- [14] *Dokumentacja PostGIS*, dostępna pod adresem:  
<http://postgis.net/documentation/>, aktualne na dzień 08.03.2017r.

- [15] *Dokumentacja Ruby on Rails*, dostępna pod adresem:  
<http://guides.rubyonrails.org/>, aktualne na dzień 08.03.2017r.
- [16] *Dokumentacja Roda*, dostępna pod adresem:  
<http://roda.jeremyevans.net/documentation.html>, aktualne na dzień 01.06.2017r.
- [17] *Dokumentacja SpatiaLite*, dostępna pod adresem:  
<https://www.gaia-gis.it/fossil/libspatialite/index>, aktualne na dzień 09.06.2017r.
- [18] Huisman Otto, By (de) Rolf A., *Principles of Geographic Information Systems*, ITC, 2009
- [19] Martin Robert, *The Clean Architecture*, dostępna pod adresem:  
<https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html>, aktualne na dzień 20.04.2017r.
- [20] Ruby Sam, Thomas Dave, Hansson Heinemeier David, *Agile Web Development with Rails 5*, Pragmatic Programmers, 2016
- [21] Schmandt Michael, *GIS Commons: An Introductory Textbook on Geographic Information Systems*, dostępne pod adresem:  
<http://giscommons.org/>, aktualne na dzień 07.04.2017r.
- [22] Smyrdek Przemysław, *Czym jest framework i po co go używać*, dostępne pod adresem:  
<http://poznajprogramowanie.pl/czym-jest-framework-i-po-co-go-uzywac/>, aktualne na dzień 20.04.2017r.