Spis treści

1. Wstęp ... 3
Plan pracy ... 4
2. Analiza tematu ... 5
2.1. Istniejące rozwiązania ... 5
2.2. Wymagania ... 7
2.3. Język SQL ... 8
2.3.1. Elementy języka ... 8
2.3.2. Rodzaje instrukcji ... 10
2.3.3. Funkcje ... 12
2.3.4. Operatory ... 13
2.4. Logika rozmyta ... 15
3. Przedmiot pracy ... 19
3.1. Wykorzystane technologie ... 20
3.1.1. PostgreSQL ... 20
3.1.2. JavaScript/ECMAScript ... 20
3.1.3. TypeScript ... 22
3.1.4. Webpack ... 24
3.1.5. Electron ... 25
3.1.6. Architektura Flux, React i Redux ... 26
3.2. Projekt ... 30
3.2.1. Projekt rozszerzenia dla elementów rozmytych w systemie PostgreSQL ... 30
3.2.2. Architektura aplikacji klienckiej ... 33
3.2.3. Edytor kodu SQL ... 35
3.2.4. Działanie systemu dydaktycznego na wielu platformach ... 37
3.2.5. Interfejs użytkownika ... 40
4. Badania ... 42
4.1. Metodyka badań wydajności operatorów ... 43
4.2. Zbiór danych testowych ... 45
4.3. Wyniki pomiaru wydajności operatorów rozmytych ... 48
4.4. Metodyka badań pomiaru wydajności aplikacji Fuzzybase ... 54

4.5. Zbiór danych testowych do pomiaru wydajności aplikacji Fuzzybase 54
4.6. Wyniki pomiaru wydajności aplikacji Fuzzybase 55
4.7. Wpływ wykorzystania zapytań rozmytych na ilość zwracanych danych 58
4.8. Problemy napotkane w pracy 60
4.8.1. Dostosowanie projektu rozszerzenia do obsługi wielu systemów operacyjnych 61
4.8.2. Kompilacja rozszerzenia dla elementów rozmytych w systemie PostgreSQL 62
4.8.3. Rozwiązanie pozwalające na edycję kodu 64
4.8.4. Płynność działania aplikacji pod obciążeniem 64
5. Podsumowanie 66
Bibliografia i
Spis skrótów i symboli iv
Zawartość dołączonej płyty v
Spis rysunków vi
Spis tabel viii
Spis listingów ix

1. Wstęp

Celem niniejszej pracy dyplomowej jest realizacja nowego narzędzia dydaktycznego umożliwiającego tworzenie zapytań SQL zawierających elementy rozmyte z wykorzystaniem nowych technologii i zbadanie wydajności takiego rozwiązania. Nowe narzędzie ma bazować na operatorach rozmytych opracowanych przez dr inż. Bożenę Małysiak-Mrozek w rozprawie doktorskiej „Metody aproksymacyjnego wyszukiwania obiektów w bazach danych" [1] oraz narzędzia dydaktycznego FuzzyQ, stworzonego w 2005r. przez mgr Bartosza Dziedzica w jego pracy dyplomowej [2]. 13-letnia aplikacja, wykorzystująca połączenie technologii Flash oraz C#, skompilowana jako samodzielna aplikacja działająca w środowisku Windows, jest na dzień dzisiejszy przestarzała technologicznie. Nowe narzędzie ma wprowadzić funkcjonalność tworzenia zapytań z użyciem podpowiedzi kontekstowych, a także umożliwić automatyczne dodawanie operatorów do bazy. Ponadto, dzięki zastosowaniu wieloplatformowego środowiska Electron, ma być możliwe uruchomienie aplikacji na systemach Linux, Windows oraz macOS. Podczas pracy nad rozwiązaniem zostanie przebadana wydajność utworzonego rozszerzenia dla bazy danych PostgreSQL (porównanie wydajności funkcji natywnych i pisanych w języku PL/pgSQL na dużych zbiorach). Poruszony zostanie również aspekt wydajności środowiska Electron i przedstawione zostaną sposoby na jej poprawienie.

Plan pracy

Praca składa się z pięciu rozdziałów. W rozdziale pierwszym opisany jest cel, motywacja i plan pracy dyplomowej. W rozdziale drugim są określone wymagania funkcjonalne systemu i usprawnienia względem FuzzyQ. W jego podrozdziałach opisany jest język SQL, na którym opiera się system dydaktyczny oraz omawiane jest zagadnienie logiki rozmytej i podstawowych operatorów w analogii do logiki boolowskiej. Rozdział trzeci przybliża technologie wykorzystane do zbudowania systemu, uzasadniając ich wybór. Druga część rozdziału opisuje architekturę i projekt aplikacji. W rozdziale czwartym (Badania) opisana jest architektura całego systemu, składającego się z oddzielnie przygotowanych operatorów rozmytych i aplikacji do zarządzania bazami danych ze wsparciem zapytań rozmytych, działającej w środowisku Electron. Opisane są rozwiązania technicznych, takich jak budowa komponentów aplikacji, komunikacja między wątkami, czy dodawanie operatorów do dowolnej bazy relacyjnej. Poruszone zostały problemy napotkane w pracy oraz wyjaśniono ich rozwiązania. Trzon rozdziału czwartego stanowią testy wydajności sprawdzające z jednej strony rozszerzenie do systemu zarządzania bazą danych PostgreSQL, a z drugiej szybkość prezentacji danych w środowisku przeglądarkowym jakim jest Electron. Rozdział piąty zawiera podsumowanie prac nad systemem, wnioski płynące z wyników badań i przedstawienie możliwości jego rozwoju.

2. Analiza tematu

Obecną wersję systemu dydaktycznego FuzzyQ wyróżnia prosty interfejs, możliwość tworzenia zapytań do bazy i graficzny kreator zapytań [2]. Aplikacja jest jednak przestarzała pod względem technologicznym, gdyż wykorzystuje porzucone środowisko Flash i nie zadziała bezpośrednio na systemie Ubuntu, wykorzystywanym w większości stanowisk laboratoryjnych. FuzzyQ ma także braki pod względem interfejsu, nie spełniając wysokich standardów dzisiejszych narzędzi programistycznych i trendów z zakresu *user experience* wyznaczanych przez firmy JetBrains, Microsoft, czy Google. Za przykład może posłużyć brak kolorowania składni, wsparcia dla wcięć i podpowiedzi kontekstowych w polu edycji zapytania. Nowy system dydaktyczny nie będzie zatem prostym przepisaniem istniejącego narzędzia na inny język programowania, lecz nową implementacją pomysłu. System, z nowymi funkcjonalnościami, zostanie zrealizowany w sposób otwarty na rozszerzanie i zgodny ze współczesnymi standardami projektowania interfejsów.

2.1. Istniejące rozwiązania

Poza aplikacją FuzzyQ zrealizowaną na Politechnice Śląskiej, na Politechnice Poznańskiej w 2006 roku powstało narzędzie SQLf_j oparte o bazę danych MySQL, dokonujące translacji z języka SQLf (SQL z rozszerzeniami rozmytymi) na SQL przy użyciu tabeli pomocniczych [3]. Tabela „possibility” zawiera definicje funkcji trapezowych dla danych liczbowych, a tabela „similarity” opisuje podobieństwa między parami ciągów znaków. Obsługiwany jest również minimalny próg przynależności wyników i ograniczenie ich liczby. Jest to podejście, które pozwala na dodawanie definicji i zależności w prosty sposób, lecz całość wymaga korzystania

z ograniczonego interfejsu konsolowego, który służy do tłumaczenia poleceń, pozwalającego jedynie na wykonanie instrukcji SELECT [4].
W sieci obecna jest również strona opisująca język FSQL (Fuzzy SQL) wraz z serwerem FSQL dla nieprecyzyjnej relacyjnej bazy danych (FSQL Server for a Fuzzy Relational Database) bazująca na serwerze Oracle [5] [6]. Rozwiązanie pozwala na definiowanie etykiet dla zakresów danych i definicję progu wyników oraz posiada standardowy zestaw komparatorów rozmytych.
Poza wymienionymi narzędziami nadającymi się wyłącznie do celów akademickich, nie ma rozpowszechnionych i nadających się do wykorzystania w praktyce systemów wyszukiwania z numerycznymi elementami rozmytymi. Istnieją natomiast przykłady pokazujące jak rozszerzyć bazę danych o odpowiednie funkcje i operatory, które można zastosować w medycynie czy ekonomii [7] [8] [9] [10]. Popularnością cieszą się natomiast funkcje do wyszukiwania rozmytego w ciągach znaków. PostgreSQL zawiera implementację metod ułatwiających użytkownikowi wyszukanie tekstu nawet, jeśli został on wpisany z błędem: obliczającą odległość Levenshteina między wyszukiwaną frazą a przeszukiwanymi danymi [11], *metaphone* – określającą przybliżony zapis fonetyczny [12]. Za przykład może posłużyć również pełnotekstowy silnik wyszukiwania Elasticsearch, mogący przeszukiwać tekst licząc odległość Levenshteina [13].

2.2. Wymagania

Podstawowym wymaganiem stawianym projektowanemu systemowi jest współczesny interfejs umożliwiający tworzenie zapytań SQL ze wsparciem dla elementów rozmytych. Przyjmując, że użytkownik systemu zna język SQL na poziomie podstawowym, w odróżnieniu od FuzzyQ, tworzony system nie będzie miał graficznej reprezentacji zapytań, która w FuzzyQ sprowadzała się do prostego grafu umożliwiającego wybór rodzaju elementów w tworzonym zapytaniu [2]. Zamiast tego edytor zapytań SQL ma być wzbogacony o kolorowanie składni oraz podpowiedzi kontekstowe, takie jakie występują we wszystkich popularnych środowiskach programistycznych po naciśnięciu skrótu klawiszowego (np. IntelliJ, VS Code). Podpowiedzi kontekstowe mają zawierać zarówno standardowe elementy języka SQL, jak i rozszerzenie w postaci elementów rozmytych. System ma wspierać tworzenie połączeń z bazami danych i przechowywanie ich. Interfejs aplikacji ma umożliwiać wyświetlanie tabel bazy i swobodne przeglądanie wyników zapytań. System powinien działać na wszystkich popularnych systemach operacyjnych: Windows, Linux (Ubuntu) oraz macOS.

System ma posiadać zintegrowany zestaw operatorów, które będzie można dodać do dowolnej bazy. Operatory w systemie będą bazować na operatorach będących częścią rozprawy doktorskiej dr inż. Bożeny Małysiak-Mrozek, zostaną jednak zrefaktoryzowane i przystosowane do systemu tak, aby można było rozszerzyć o nie dowolną bazę danych. Atutem byłaby możliwość samodzielnego tworzenia nowych operatorów.

2.3. Język SQL

Język SQL został zaproponowany przez Donalda Chamberlina i Raymonda Boyce'a dwa lata po tym, jak Edgar „Ted" Codd zaprezentował koncepcję relacyjnego modelu danych na sympozjum w 1972 roku [14]. Twórcy języka SQL zauważyli, że matematyczną notację w językach Codda – algebrze relacji (*relational algebra*) i rachunku relacji (*relational calculus*) – ciężko pojąć osobie bez wykształcenia matematycznego, a także jej zapis na klawiaturze będzie problematyczny. Tak powstał SEQUEL (*A Structured English Query Language*) - język pozwalający na tworzenie złożonych zapytań do relacyjnych baz danych w czytelny i prosty sposób, a także obejmujący operacje modyfikacji danych i administracji bazą. Po rozpowszechnieniu języka SQL wśród twórców oprogramowania konieczna była standaryzacji. Na przestrzeni lat powstało 9 kolejnych wersji standardu, dodające nowe typy danych, rodzaje dozwolonych operacji i zabezpieczenia. Najnowsza wersja standardu, SQL:2016, została wydana w grudniu 2016 roku, m.in. dodając wsparcie dla dokumentów w notacji JSON, formatowanie i parsowanie daty i czasu oraz rozpoznawanie wzorców w wierszach [15]. Niestety, żadna z istniejących implementacji nie jest zgodna ze standardami opracowanymi 2 lata temu. Nie zawsze oznacza to brakujące funkcjonalności, lecz ich realizację przy użyciu składni niezgodnej ze standardem. Dla przykładu, baza Postgres posiada wsparcie dla obiektów JSON już od wersji 9.2 z roku 2012.

2.3.1. Elementy języka

Język SQL posiada zestaw podstawowych typów danych, które mogą być składowane w tabelach i muszą być podane podczas tworzenia kolumn. Najważniejsze to: `CHARACTER`, `BINARY`, `BOOLEAN`, `INTEGER`, `FLOAT`, `REAL`, `DATE`, `TIME`, `YEAR`, `MONTH`, `DAY`, `HOUR` i `MINUTE`. Ciągi, takie jak `CHARACTER` i `BINARY` przyjmują domyślnie stałą długość, gdzie puste miejsce jest wypełniane znakami spacji w przypadku ciągu znaków, a zerami w przypadku ciągu binarnego. Dzięki opcji `VARYING` można przechowywać ciągi zmiennej długości. Drugim standardowym elementem języka są ope-

ratory: porównania (`=`, `<>`, `<`, `<=`, `=>`, `>`), zawierania w zakresie liczb (`BETWEEN`), zawierania w zbiorze (`IN`), podobieństwa ciągów znaków (`LIKE`), porównania do wartości `NULL`/`TRUE`/`FALSE` (`IS`, `IS NOT`), porównania do innej wartości z uwzględnieniem `NULL` (`IS NOT DISTINCT FROM`) oraz operator przemianowania zwracanej kolumny (`AS`). Podobnie jak w zdecydowanej większości języków, nadmiarowe białe znaki są ignorowane, a komentarze mogą być zawarte wewnątrz znaków `/* */`. Jednoliniowe komentarze rozpoczyna się znakami `--`. Przechodząc do składni języka, można z niej wyodrębnić:

- wyrażenia (ang. *expressions*) – mogą zwracać tabele bądź wartości skalarne, np. `age + 1`.
- predykaty (ang. *predicates*) – warunki, składające się z operatorów i wyrażeń, przeliczane na wartości prawda/fałsz/nieznane z użyciem logiki trójwartościowej; służą do filtrowania wyników zapytań i instrukcji, bądź zmiany przebiegu programu, np. `sex='M'`.
- klauzule (ang. *clauses*) – słowa kluczowe, z których zbudowane są zapytania i instrukcje, np. `SELECT name, last_name`, `WHERE age>20`, `SET value=2`.
- instrukcje (ang. *statements*) – pozwalają na odczyt i zapis danych, transakcjami, połączeniami i przebiegiem wykonania programu, są szerzej opisane w podrozdziale 0. Instrukcje muszą być oddzielone od siebie średnikiem.
- zapytania (ang. *queries*) – instrukcje zwracające dane na podstawie podanych kryteriów, np. SELECT * FROM employees WHERE salary<3000; inne instrukcje zwracają najczęściej liczbę zmodyfikowanych (`UPDATE`) lub usuniętych (`DELETE`) wierszy.

Listing 2.1 obrazuje elementy języka SQL. W przykładzie najpierw następuje dodanie wierszy do tabeli ze studentami, a następnie jest wykonane zapytanie zwracające grupy wraz z liczbą studentów, którzy ukończyli pierwszy semestr:

```
INSERT INTO students (name, faculty, group, semester)
  VALUES
    („Krzysztof Miemiec”, „AEI”, „ISMiP”, 3),
    („Jan Kowalski”, „AEI”, „ISMiP”, 1),
    („Adam Nowak”, „AEI”, „BDiS”, 2);
```

```
SELECT group, COUNT(*)
  FROM students
  WHERE semester>=2
  GROUP BY group;
```

Listing 2.1 Przykładowe zapytanie SQL

2.3.2. Rodzaje instrukcji

Z języka SQL można wydzielić kilka podzbiorów zajmujących się osobnymi rodzajami zadań.

- DQL – *Data Query Language* (z ang. język zapytań do danych); zawiera polecenie SELECT zwracające dane z podanej tabeli.
- DML – *Data Manipulation Language* (z ang. język manipulacji danymi); zawiera polecenia INSERT, UPDATE i DELETE, które odpowiednio wstawiają, modyfikują i usuwają dane z tabeli.
- DDL – *Data Definition Language* (z ang. język definiowania danych); składa się z poleceń CREATE, ALTER i DROP (TABLE/INDEX/VIEW), służące do tworzenia, modyfikacji i usuwania tabel, indeksów i widoków.
- DCL – *Data Control Language* (z ang. język kontroli nad danymi); obejmuje polecenia CRAETE USER, GRANT, DENY i REVOKE, pozwalające na nadanie uprawnień, zabronienie wykonania operacji lub odebranie uprawnień.
- TCL – *Transactional Control Language* (z ang. język kontroli nad transakcjami); składa się z poleceń COMMIT i ROLLBACK służących do zatwierdzania i wycofywania zmian z transakcji.

Choć operatory mają zastosowanie w dowolnej instrukcji, najczęściej wykorzystywaną w pracy instrukcją będzie SELECT, w związku z tym zostanie ona opisana szerzej. Przedstawiona składnia zapytania z użyciem instrukcji SELECT jest oparta o dokumentację bazy PostgreSQL:

```
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    [ * | expression [ [ AS ] output_name ] [, ...] ]
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY grouping_element [, ...] ]
    [ HAVING condition [, ...] ]
```

```
    [ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS {
FIRST | LAST } ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start [ ROW | ROWS ] ]
```

Klauzula `SELECT` przyjmuje najpierw ciąg wyrażeń do zwrócenia w wyniku. W celu zwrócenia wszystkich kolumn można wykorzystać znak `*`. Domyślnie uzupełniane słowo kluczowe `ALL` powoduje zwrócenie wszystkich wyników zapytania, wliczając duplikaty. Po dodaniu słowa kluczowego `DISTINCT`, duplikaty nie są zwracane w wyniku zapytania, a `DISTINCT ON` usuwa zduplikowane wiersze odpowiadające podanym wyrażeniom [7].

Następnie obliczany jest wynik elementów z klauzuli `FROM`. Może być to jeden lub więcej elementów, na które składają się: nazwy tabel, zagnieżdżone instrukcje, nazwy funkcji i złączenia (`JOIN`). W przypadku występowania wielu elementów, zawartość ich wierszy jest łączona. Po słowie kluczowym `ON` wymagane jest podanie predykatu określającego warunek złączenia tabel. Klauzula `JOIN` pozwala na złączenie dwóch elementów przyjmowanych przez klauzulę `FROM`. `CROSS` oraz `INNER JOIN` zwracają proste złączenie, takie jakie powstaje przy podaniu wielu elementów do klauzuli `FROM`, ograniczone jednak warunkiem następującym po słowie kluczowym `ON`. Można je zastąpić przy użyciu klauzul `FROM` i `WHERE`. `LEFT OUTER JOIN` zwraca natomiast złączone elementy obu tabel, wraz z elementami tabeli po lewej stronie, które nie mogły zostać dopasowane przy użyciu warunku i w których brakujące pola zostały zastąpione wartościami `NULL`. Analogicznie, `RIGHT OUTER JOIN` zwraca złączone elementy obu tabel wraz z niedopasowanymi elementami prawej tabeli, a `FULL OUTER JOIN` zwraca złączone i niedopasowane elementy obu tabel, w których brakujące pola przyjmują wartość `NULL`.

Jeśli zostanie podana klauzula `WHERE`, wynik będzie zawierał jedynie wiersze spełniające podane w niej predykaty, które mogą być połączone operatorami logicznymi `AND` i `OR`.

Kolejna, opcjonalna klauzula `GROUP BY` pozwala na złączenie wielu wierszy, które spełniają warunki grupowania, w pojedyncze wiersze dzielące pola z tymi samymi wartościami. Funkcje agregujące (np. `COUNT`, `AVG`, `SUM`, `MAX`, `MIN`) zostaną wykonane na wierszach grup, tworząc oddzielne wyniki dla każdej z nich. Klauzula `HAVING`,

podobnie jak `WHERE`, przyjmuje predykaty sprowadzane do wartości logicznej, jednak nie mogą być użyte kolumny niezgrupowane bądź niebędące funkcjami agregującymi. Klauzula `HAVING` jest aplikowana na zapytania zawierające wiersze zgrupowane, a zatem po zgrupowaniu ich przy użyciu `GROUP BY`. Jeżeli klauzula `GROUP BY` nie występuje w zapytaniu, jest ono automatycznie przekształcane w zapytanie zgrupowane.
Klauzula `ORDER BY` przyjmuje listę wyrażeń w postaci nazw lub liczb porządkowych kolumn wynikowych lub wyrażeń utworzonych z kolumn wejściowych. Liczbę porządkową kolumny można użyć, gdy nazwy kolumn wynikowych powtarzają się, czego można jednak uniknąć stosując operator `AS`. Do każdego wyrażenia można dodać słowo kluczowe `DESC`, aby posortować wyniki malejąco. Domyślnie przyjmowana jest wartość `ASC` – sortowanie rosnące. Dopuszczalne jest również użycie własnych operatorów sortowania. Ostatnim elementem wyrażenia są słowa kluczowe `NULLS LAST` i `NULLS FIRST`. Oznaczają one odpowiednio posortowanie wierszy z wartościami `NULL` jako ostatnie i jako pierwsze. Domyślnie przyjęto, że w przypadku sortowania rosnącego wartości `NULL` są wartościami ostatnimi, a w przypadku sortowania malejącego, wartości `NULL` znajdują się na początku.
Ostatnimi klauzulami są `LIMIT` oraz `OFFSET`. Ograniczają one wyniki do zadanej liczby wierszy, a także pozwalają wyświetlić je z przesunięciem. W przypadku wykorzystywania tych klauzul ważne jest użycie klauzuli `ORDER BY`, gdyż bez jej podania standard SQL nie gwarantuje zwracania wyników w tej samej kolejności i w rezultacie można otrzymać różne wyniki dla tego samego zapytania.

2.3.3. Funkcje

Funkcje pozawalają na wyodrębnienie zestawu instrukcji do wykonania pod określoną nazwą. Każda funkcja musi mieć zdefiniowane argumenty, a wykorzystać można wszystkie typy zarejestrowane wcześniej w bazie. Funkcja może przyjmować również zmienną liczbę argumentów tego samego typu (zawsze umieszczonych na końcu). Wspierane jest przeciążanie funkcji, a zatem można zdefiniować wiele funkcji o tej samej nazwie, ale o różnych typach argumentów [16, 17].
PostgreSQL pozwala na tworzenie funkcji w języku SQL, PL/pgSQL i C. Funkcje w języku SQL zawsze zwracają w wyniku ostatnie wyrażenie. Język PL/pgSQL (Procedural Language/PostgreSQL) wzbogaca język SQL o elementy sterujące (pętle, instrukcje warunkowe) i zmienne. W celu zwiększenia wydajności, w języku C

warto implementować funkcje, które będą wykonywane często i dla dużej liczby elementów, np. funkcje agregujące [18].

Przykładowa instrukcja wykorzystywana w pracy do dodawania typów rozmytych do tabeli, została przedstawiona na listingu 2.2.

```
CREATE OR REPLACE FUNCTION add_fuzzy_type(
  type_name VARCHAR(64)
) RETURNS fuzzy.types.id%TYPE AS $$
  INSERT INTO fuzzy.types (name) VALUES (type_name) ON CONFLICT DO
NOTHING;
  SELECT id FROM fuzzy.types WHERE name=type_name;
$$ LANGUAGE sql VOLATILE;
```

Listing 2.2 Funkcja w języku SQL dodająca typy rozmyte do bazy

2.3.4. Operatory

Operatory są niczym innym, jak skrótowym zapisem funkcji wykonywanych na jednym lub dwóch argumentach, określane są najczęściej mianem „lukru składniowego”. Nazwy operatorów składają się od 1 do 63 znaków ze zbioru `+-*/<>=~!@#%^&|`?`. Standard języka SQL ogranicza jednak dozwolone nazwy, dla przykładu, nie jest dozwolone zastosowanie dwóch znaków – obok siebie, gdyż zostałyby one sparsowane jako oznaczenie komentarza. Do tego operator `!=` jest automatycznie przetwarzany do formy `<>`. Do zdefiniowania operatora wymagane jest podanie typów operandów (`LEFTARG` i `RIGHTARG`) i funkcji wykonującej funkcję. Operatory mogą zawierać również dodatkowe elementy optymalizujące wykonanie zapytań, takie jak komutatory (operatory, które po zmianie kolejności argumentów dają ten sam wynik, jak np. `+` dla dodawania) i negatory (operatory, które po zmianie kolejności argumentów dają wynik odwrotny, tak jak operatory porównania `>` i `<=`) [17].

Przykładowa instrukcja tworząca operator pobrania odpowiedniej nazwy dla liczby w danym typie rozmytym przedstawiona została na listingu 2.3.

```
CREATE OPERATOR ~> (
  PROCEDURE = get_fuzzy_name,
  LEFTARG = FLOAT8,
  RIGHTARG = VARCHAR(64)
);
```

Listing 2.3 Operator w języku SQL

2.4. Logika rozmyta

Twórcą teorii zbiorów rozmytych i logiki rozmytej jest Lotfi Zadeh, który zaproponował wykorzystanie funkcji przynależności przyjmującej wartości z przedziału <0, 1> [9]. Zbiór rozmyty A w przestrzeni X można zdefiniować jako zbiór par [1]:

$A = \{(\mu_A(x), x)\}$, dla każdego $x \epsilon X$

gdzie:

μ_A – funkcja przynależności zbioru rozmytego A, przypisująca każdemu elementowi zbioru $x \epsilon X$ stopień jego przynależności $\mu_A(x) \epsilon [0, 1]$ do zbioru A.

Logika rozmyta to wielowartościowa odmiana logiki, która w odróżnieniu od klasycznej logiki boolowskiej pozwala na zapisanie wartości między prawdą (1 - przynależność do zbioru), a fałszem (0 – brak przynależności do zbioru). Proces pozwalający na przekształcenie wartości z dziedziny liczb rzeczywistych na wartość rozmytą nazywany jest fuzzyfikacją (rozmywaniem). Proces odwrotny nazywany jest defuzzyfikacją (wyostrzaniem). Wartość rozmyta może być przetworzona na wartość logiczną trójwartościową (prawda, fałsz, niezdefiniowana). Wartość niezdefiniowana zostanie zwrócona w przypadku braku funkcji przynależności lub wartości liczbowej. Odpowiedniki podstawowych boolowskich operacji logicznych w logice rozmytej zostały przedstawione w tabeli 2.1.

Logika boolowska	Logika rozmyta
X AND Y	MIN(X, Y)
X OR Y	MAX(X, Y)
NOT X	1 - X

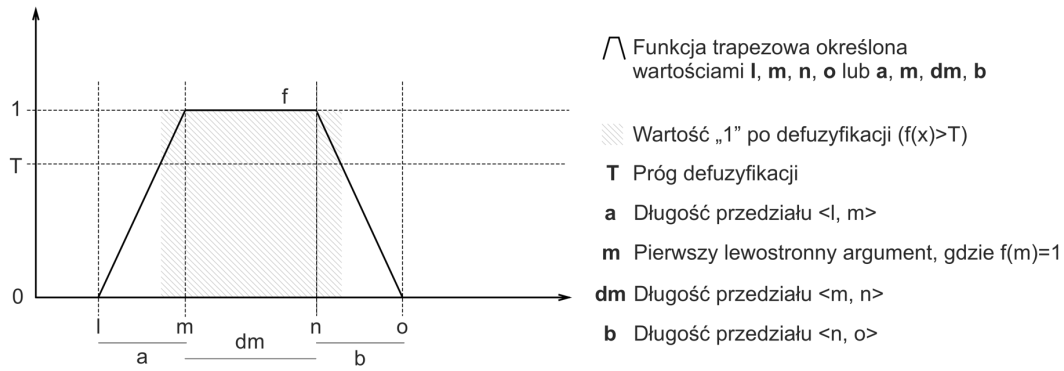
Tabela 2.1 Operatory logiczne

Stopień przynależności wartości numerycznej do zbioru rozmytego jest określany przez odpowiednią funkcję rozmytą. Najczęściej funkcje te należą do jednej z klas

funkcji: trójkątnych, trapezowych, Gaussa, dzwonowych czy sigmoidalnych. W niniejszej pracy funkcje przynależności są funkcjami trapezowymi. Funkcja trapezowa może zostać opisana za pomocą wartości granicznych przedziałów: lewostronnego (zwartościami rosnącymi od 0 do 1), środkowego (ze stałą wartością 1) i prawostronnego (z wartościami malejącymi od 1 do 0) i określona jest wzorem:

$$\mu_A(x; l, m, n, o) = \begin{cases} 0, & x \leq a \\ \dfrac{x-a}{b-a}, & a < x \leq b \\ 1, & b < x \leq c \\ \dfrac{d-x}{d-c}, & c < x \leq d \\ 0, & x > d \end{cases}$$

Istnieje druga możliwość reprezentacji funkcji, w postaci lewostronnego brzegu przedziału środkowego oraz długości wszystkich wyżej wymienionych przedziałów. Wykorzystanie przedziałów w wewnętrznej strukturze funkcji pozwala na uproszczenie operacji arytmetycznych, gdyż większość z nich dotyczy przedziałów. Na rysunku 2.1 są opisane parametry funkcji trapezowej wykorzystywane w pracy oraz przykład defuzyfikacji. Do defuzyfikacji wartości rozmytej wykorzystywana jest wartość progowa (ang. *threshold*), a następnie na podstawie operatora porównania (równy, nie równy, mniejszy, większy, mniejszy lub równy, większy lub równy) wyznaczana jest wartość boolowska.



Rys. 2.1 Funkcja trapezowa

Poza samą klasą funkcji należy zdefiniować operacje, jakie można na niej wykonać. Są to: negacja, podstawowe operacje arytmetyczne (dodawanie, odejmowanie, mnożenie, dzielenie) oraz wyznaczanie stopnia przynależności. Każda z tych operacji może przyjąć za argumenty zarówno funkcje trapezowe, jak i liczby (Tabela 2.2).

Operacja	**Zapis**	**Opis**
Negacja	-f	Wynikiem jest funkcja, której parametry (l, m, n, o) zostały przekształcone symetrycznie względem 0.
Dodawanie liczby	f+r r+f	Wynikiem jest funkcja f, w której do każdego z parametrów (l, m, n, o) dodano liczbę r.
Dodawanie funkcji	f1+f2	Wynikiem jest funkcja f, której argumenty (m, dm, a, b) są sumami parametrów (m, dm, a, b) funkcji f1 i f2.
Odejmowanie liczby	f-r	Wynikiem jest funkcja f, w której do każdego z parametrów (l, m, n, o) odjęto liczbę r.
Odejmowanie funkcji od liczby	r-f	Wynikiem jest zanegowana funkcja f, w której wcześniej od każdego z parametrów (l, m, n, o) odjęto liczbę r.
Odejmowanie funkcji	f1-f2	Wynikiem jest funkcja f, której parametry (m, dm, a, b) są różnicami parametrów (m, dm, a, b) funkcji f1 i f2.
Mnożenie liczby	f*r r*f	Wynikiem jest funkcja f, której parametry (m, dm, a, b) zostały pomnożone przez liczbę r.
Mnożenie funkcji	f1*f2	Wynikiem jest funkcja f, której parametry (l, m, n, o) są iloczynami parametrów funkcji f1 i f2.
Dzielenie przez liczbę	f/r	Wynikiem jest funkcja f, której parametry (m, dm, a, b) zostały podzielone przez liczbę r.
Dzielenie liczby	r/f	Wynikiem jest funkcja f, której parametry (m, dm, a, b) są wynikiem dzielenia liczby r przez parametry (m, dm, a, b) funkcji f.
Dzielenie przez funkcję	f1/f2	Wynikiem jest funkcja f, której parametry (l, m, n, o) są ilorazami parametrów funkcji f1 i f2.

Tabela 2.2 Wykaz operacji na trapezowych funkcjach przynależności

Operacja	Zapis	Opis
Przynależność liczby	r~=f	Wynikiem jest wartość funkcji dla argumentu r.
Przynależność funkcji	f1~=f2	Wynik to 1, jeżeli przedziały <m, n> funkcji f1 i f2 nachodzą na siebie; w przeciwnym wypadku wynikiem jest maksymalna wartość w punkcie przecięcia się krawędzi z przedziałów <l, m> i <n, o> obu funkcji.

Tabela 2.2 Wykaz operacji na trapezowych funkcjach przynależności

Praktycznym zastosowaniem zbiorów rozmytych są zmienne lingwistyczne będące wielkością określoną poprzez wartości lingwistyczne. Zmienna lingwistyczna składa się z wartości lingwistycznych, z których każdej może zostać przyporządkowany zbiór rozmyty. Przykładem zmiennej lingwistycznej jest wielkość *temperatura*, przyjmująca wartości *lodowato*, *zimno*, *ciepło*, *gorąco*. Każda z wartości może zostać określona za pomocą funkcji trapezowych:

- lodowato: $\mu_{temperatura}(x;\ -100, -100, -10,\ 5)$
- zimno: $\mu_{temperatura}(x;\ -10,\ 0,\ 15,\ 20)$
- ciepło: $\mu_{temperatura}(x;\ 18,\ 22,\ 28,\ 35)$
- gorąco: $\mu_{temperatura}(x;\ 28,\ 32,\ 60,\ 60)$

Temperatura w powyższym przypadku odnosi się do temperatury odczuwalnej przez człowieka i jest wyrażona w stopniach Celsjusza. Nie można jej odnieść np. do temperatury pracy silnika. Obrazuje to, że tworząc zmienne lingwistyczne odnosimy się najczęściej do specyficznej dziedziny, gdyż wykorzystanie jednej zmiennej do różnych dziedzin byłoby niepraktyczne.

3. Przedmiot pracy

Celem niniejszej pracy jest zbudowanie systemu dydaktycznego, służącego do obsługi baz danych i nauki tworzenia zapytań rozmytych. Na system składają się dwie części – bazodanowa oraz aplikacyjna. Część bazodanowa jest rozszerzeniem funkcjonalności systemu zarządzania bazą danych PostgreSQL, składającym się z typów, tabel, funkcji SQL i funkcji natywnych uruchamianych z poziomu zapytań SQL, upraszczających budowanie zapytań rozmytych. Część aplikacyjna to program komunikujący się z bazą danych, który docelowo powinien być uruchamialny na platformach Windows, Linux i macOS. Aplikacja ma umożliwiać wyświetlanie wyników zapytań w wydajny i przejrzysty sposób, a także tworzenie zapytań przy użyciu edytora kodu. Wydajność obu części rozwiązania zostanie przebadana na dużych zbiorach danych podczas testów. Elementem badań będzie też wybór technologicznych rozwiązań i zastosowanej architektury, które z uwagi na szybki rozwój i częste zmiany nie zostały jeszcze dobrze opisane w pracach akademickich. Wszystkie elementy aplikacji powinny być osadzone w interfejsie użytkownika o niskim poziomie skomplikowania, lecz dużej użyteczności. Za nazwę systemu przyjęto złączenie słów w języku angielskim oznaczających rozmyty (ang. *fuzzy*) i baza (ang. *base*) - Fuzzybase.

3.1. Wykorzystane technologie

Do zbudowania systemu dydaktycznego postanowiono wykorzystać jedno z najpopularniejszych w ostatnich latach środowisk do tworzenia aplikacji wieloplatformowych – Electron. Środowisko to jest wykorzystywane m.in. przez komunikatory Skype i Slack, środowisko programistyczne VS Code czy narzędzie do zarządzania repozytoriami z kodem GitKraken. Aplikacja, oparta o frameworki React i Redux, zostanie napisana w języku TypeScript, który jest następnie tłumaczony do języka JavaScript. Całość jest pakowana do skompresowanej i zoptymalizowanej paczki z kodem przy użyciu narzędzia Webpack, a następnie zamknięta w pliku natywnej aplikacji (np. *.exe dla Windows, *.app dla macOS) poprzez wspomniane środowisko Electron. Uzasadnieniem powyższych wyborów jest uniwersalność, prostota działania, a także znajomość technologii przez autora pracy.
Poza samą aplikacją, system dydaktyczny ma zawierać również zestaw funkcji rozmytych stanowiących rozszerzenie dla baz danych. Podobnie jak w poprzednim rozwiązaniu, zostanie użyta baza PostgreSQL, a funkcje pozostaną w języku C, jednak zostaną poddane refaktoryzacji [2].

3.1.1. PostgreSQL

Postgres to jeden z powszechnie używanych systemów obiektowo-relacyjnych baz danych. Umożliwia dodawanie funkcjonalności poprzez rozszerzenia, czego popularnym przykładem jest rozszerzenie umożliwiające przechowywanie informacji geograficznych PostGIS. Szczegółowy opis tworzenia rozszerzeń dla systemu Postgres został przedstawiony w rozdziale 4.8.

3.1.2. JavaScript/ECMAScript

JavaScript to wysokopoziomowy, dynamicznie typowany język programowania, który powstał w latach 90. JavaScript został przekazany organizacji Ecma International w celu ustalenia wspólnego standardu dla wszystkich producentów oprogramowania. Został określony w standardzie ECMA-262 i jest obecnie znany jako EC-

MAScript (skrótowo ES) [19]. Jako język wieloparadygmatowy umożliwia podejścia: obiektowe (poprzez tworzenie klas określanych prototypami), zdarzeniowe (ang. event-driven), funkcyjne i imperatywne.

W kilka lat po debiucie środowiska uruchomieniowego Node.js w 2009 i udostępnienia oficjalnej platformy do udostępniania pakietów Node Package Manager nastąpił gwałtowny wzrost popularności języka, który wraz z dostępnymi pakietami umożliwia obecnie tworzenie aplikacji przeglądarkowych, mobilnych, serwerowych i desktopowych.

Działanie języka JavaScript jest ograniczone do wykonywania kolejnych operacji w jednym wątku w tzw. pętli zdarzeń [20]. Dzięki temu maszyna wirtualna nie musi zarządzać współdzieleniem pamięci między wątkami, a także zarządzać nimi, co pozwala na osiągnięcie bardzo dobrych wyników wydajnościowych. Pętla zdarzeń pozwala na asynchroniczność działania aplikacji, jednak nie oznacza to równoległości. W obrębie jednego procesu długie obliczenia nie mogą zostać oddelegowane do osobnych wątków, a w związku z tym kolejne zdarzenia w pętli nie zostaną wykonane aż do momentu wykonania obliczeń. Istnieje kilka sposobów na poradzenie sobie z problemem blokowania pętli zdarzeń. W przypadku aplikacji serwerowych zbudowanych w oparciu o platformę Node.js operacje wejścia/wyjścia są wykonywane w osobnym wątku, co pozwala na użycie jednego z mechanizmów asynchroniczności (*callback*, *Promise*, *async*/*await*, *generator*) i obsługiwanie kolejnych żądań, a następnie powrót do obsługi poprzedniego żądania po uzyskaniu danych. Drugim sposobem jest wydzielenie osobnych procesów służących do równoległej obsługi żądań (najoptymalniej, gdy ich liczba pokrywa się z liczbą wątków, które sprzętowo obsługuje procesor). Można pokusić się nawet o utworzenie dedykowanych procesów służących wyłącznie do długotrwałych operacji i korzystanie z mechanizmów IPC (ang. *inter-process communication*). Jednak nawet jeśli utworzymy dziesiątki procesów na wydajnej maszynie, wystarczy tyle samo długo trwających zadań, aby skutecznie zablokować działanie kolejek. Jedynym sposobem jest wykorzystanie operacji `setImmediate`, która opóźni wykonanie kolejnej części kodu przenosząc je na koniec kolejki. Jest to z jednej strony wada języka JavaScript - programista musi sam zarządzać „przełączaniem wątków”, a z drugiej zaleta, pozwalająca na w pełni atomowe wykonanie fragmentów kodu i uniknięcie hazardu, do rozwiązania którego najczęściej potrzebne są semafory.

Drugą ważną cechą JavaScriptu jest dynamiczne typowanie, dające nieograniczone możliwości w manipulacji danymi – nie tylko przypisywanie wartości różnych typów do zmiennych, ale także modyfikację typów (klas, obiektów, tablic) w czasie trwania programu. Do pisania trywialnych aplikacji nie jest wymagane korzystanie ze środowiska programistycznego, jednak bez korzystania ze środowiska umożliwiającego sprawdzanie i podpowiadanie typów, wraz z rosnącą złożonością aplikacji, maleje szansa na napisanie bezbłędnie działającego programu.
Aby uniknąć problemów wynikających z dynamicznego typowania (brak błędów zgłaszanych przez kompilator), wykorzystywane są narzędzia sprawdzające poprawność kodu (ang. *linter*). Popularnym wyborem jest narzędzie ESLint, które pozwala na znalezienie błędów dotyczących składni i utrzymanie spójnego stylu programowania [21].
Każdy współczesny projekt pisany z użyciem języka JavaScript posiada plik `package.json`, w którym zdefiniowane są: wersja, nazwa projektu, zależności projektowe (lista projektów zewnętrznych, z których korzysta aplikacja), zależności deweloperskie (lista projektów zewnętrznych, które wykorzystywane są do budowania projektu) i komendy uruchomieniowe.

3.1.3. TypeScript

TypeScript jest rozszerzeniem języka JavaScript, dokładającym do niego możliwość typowania [22]. Kod w TypeScripcie jest transkompilowany do JavaScriptu. Transkompilacja polega na przetworzeniu kodu w języku TS, obliczeniu typów, sprawdzeniu ich, a następnie zwróceniu wyniku w postaci kodu JS lub podsumowania błędów. Zastosowanie TypeScriptu rozwiązuje problem dynamicznego typowania opisany w poprzednim rozdziale. To, że TypeScript jest rozszerzeniem języka JavaScript oznacza całkowitą zgodność ze standardem ECMAScript – kod napisany przy użyciu czystego języka JavaScript zostanie przepuszczony przez kompilator z opcjonalnym sprawdzeniem oczywistych typów. Zmienne, stałe oraz funkcje, które nie są obłożone typami, a które kompilator języka TypeScript jest w stanie wywnioskować (ang. *infer*) z kontekstu kodu, zostaną sprawdzone pod względem poprawności. Dla przykładu, podczas sprawdzania poniższego wyrażenia zostanie zgłoszony błąd `TS2339: Property 'match' does not exist on type 'string[]'`, gdyż funkcja `match` nie istnieje w prototypie tablicy ciągów znaków, który został wywnioskowany (Listing 3.1).

```
const x = (a) => [a + 'Test'];
x('TypeScript').match(/^Type/);
```

Listing 3.1 Przykład niepoprawnego wyrażenia w języku TypeScript

Oczywistym jest, że podstawowe typy języka TypeScript pokrywają się z typami dostępnymi w języku JavaScript. Są to:

- Boolean – przechowuje wartości prawda/fałsz,
- Number - brak jest osobnych reprezentacji liczb zmienno- i stałoprzecinkowych, gdyż wszystkie liczby przechowywane są jako liczby zmiennoprzecinkowe podwójnej precyzji,
- String – ciąg znaków jest typem podstawowym, a JavaScript nie posiada typu znakowego,
- Array – tablica o nieokreślonej liczbie elementów,
- Tuple – tablica o określonej liczbie i typach elementów,
- Enum – typ wyliczeniowy, pozwalający nadać przyjazne nazwy wartościom liczbowym lub ciągom znaków,
- Any – reprezentuje dowolny typ; do zmiennych tego typu można przypisać dowolną wartość,
- Void – reprezentuje brak typu, często zwracany przez funkcje; do zmiennych tego typu można przypisać jedynie wartości `undefined` i `null`,
- Undefined i Null – typy oznaczające brak wartości, które domyślnie można przypisać do dowolnego typu (np. zmienna typu `number` może w rzeczywistości przyjąć wartość `null`); istnieje jednak możliwość włączenia opcji `strictNullChecks`, dzięki której żadna zmienna nie określona jako `undefined`, bądź `null`, nie może przyjąć tych wartości,
- Never – typ `never` określa wartości, które nigdy nie wystąpią; przykładem może być typ zwracany przez funkcję, która zawsze rzuca wyjątek przed zakończeniem swojego przebiegu,
- Object – zgodnie ze specyfikacją, jest to typ, który nie zalicza się do typów `number`, `string`, `boolean`, `null` lub `undefined`; typ `Array` zalicza się więc do typu `Object`, który sam jest w stanie przechowywać dane w postaci rekordów.

Bytami wprowadzonymi dopiero w języku TypeScipt są interfejsy oraz klasy. Interfejsy (dostępne jako `interface`) pozwalają na określenie tego, jak ma wyglądać dany obiekt lub funkcja i są sposobem na ustalenie jednolitego kontraktu między fragmentami kodu. Klasy (dostępne jako `class`) zostały wprowadzone do języka

JavaScript w standardzie ECMAScript 6. Pozwalają one na definicję fragmentów programu zgodnie z obiektowym paradygmatem programowania, z wydzieleniem konstruktora, pól i metod. TypeScript pozwala również na definiowanie aliasów dla typów złożonych (dostępne poprzez słowo kluczowe `type`). Typ może być przecięciem (częścią wspólną, działającą niczym operacja AND) lub unią (działającą tak jak operacja OR) innych typów. Możliwe jest rozszerzanie typów innymi typami, które zostały utworzone wcześniej (słowo kluczowe `extends` lub wspomniane przecięcie) oraz tworzenie typów generycznych, a więc takich, które przyjmują inne typy jako swoje parametry.
Obecnie wiele modułów dostępnych za pośrednictwem repozytorium NPM posiada wbudowane definicje typów, umieszczane w plikach z rozszerzeniem `d.ts`. Definicje typów to plikami zawierające jedynie typy i nie znajduje się w nich żaden kod wykonywalny. Część modułów nie posiada wbudowanych definicji. W odpowiedzi na potrzeby społeczności TypeScript powstało monolityczne repozytorium DefinitelyTyped, przechowujące tysiące definicji typów dla modułów bez wsparcia języka TypeScript.

3.1.4. Webpack

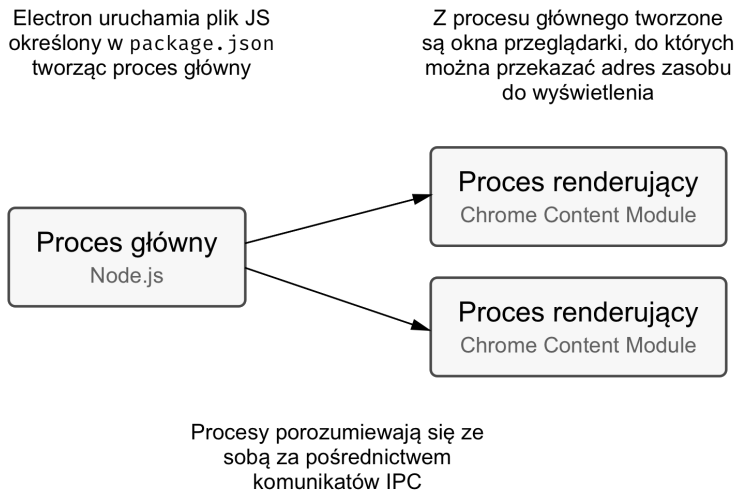
Webpack jest narzędziem służącym do automatyzacji złożonego procesu budowania aplikacji JavaScriptowych [23]. Dzięki narzędziom ładującym (ang. *loaders*) umożliwia przetworzenie wszystkich plików projektu i wyłapanie zależności między nimi i wykorzystywanych zasobów. Pozwala to na udostępnienie w pakiecie z aplikacją jedynie tych zasobów (plików graficznych, fontów i innych plików nie będących kodem) oraz fragmentów kodu, które zostały zaimportowane w punkcie wejścia aplikacji. Proces można zilustrować jako rosnące drzewo składające się z zależności w aplikacji. Optymalizacja stosowana przez narzędzie WebPack, polega na pozbyciu się nieużywanych fragmentów kodu to potrząsanie drzewem (ang. *tree-shaking*), co oznacza pozbycie się z naszego drzewa zależności, które nie mają żadnego połączenia z innymi zależnościami. Dla języka TypeScript przygotowano dedykowany *loader*, który umożliwia przetworzenie wszystkich plików tego języka w locie. Podobnie jest w przypadku zasobów graficznych. W pliku konfiguracyjnym narzędzia WebPack, wykorzystywanym również do budowy stron internetowych, najczęściej znajdują się *loadery* umożliwiające ponowną kompresję, skalowanie i optymalizację niemalże dowolnych zasobów.

3.1.5. Electron

Electron to środowisko uruchomieniowe pozwalające na tworzenie aplikacji przy użyciu języków HTML, CSS i JavaScript. Pierwotnie środowisko Electron było rozwijane pod nazwą Atom Shell przez GitHub, będąc środowiskiem stworzonym na potrzeby środowiska programistycznego Atom [24].

Platforma Electron składa się z dwóch środowisk opartych na języku JavaScript. Są to Node.js oraz Chrome Content Module. Część aplikacji działająca w Node.js to proces główny, który posiada pełny dostęp do systemu, w którym działa. Proces główny jest oczywiście ograniczony uprawnieniami użytkownika lub innego procesu, który go uruchomił. Część aplikacji działająca jako Chrome Content Module to fragment przeglądarki Chrome, umożliwiający wyświetlanie stron internetowych [25]. Zawiera wszystkie funkcjonalności niezbędne do renderowania zawartości w przeglądarce (np. HTML5) i wsparcie akceleracji graficznej. Pozbawiona jest natomiast rozszerzeń Chrome, automatycznego wypełniania formularzy, autokorekty itp. Dzięki połączeniu wyżej opisanych części składowych środowiska Electron, z poziomu aplikacji rezydującej w procesie przeglądarki, można uzyskać dostęp do interfejsów programowania aplikacji dostępnych wcześniej tylko dla aplikacji w Node.js. Ponadto w Node.js można wykorzystać natywne biblioteki programistyczne, które mogą służyć np. do przetwarzania obrazu.

Electron wykorzystuje format plików ASAR, który służy do archiwizacji wszystkich zasobów oraz kodu aplikacji. Podczas uruchamiania aplikacji w platformie Electron, najpierw uruchamiany jest proces główny, którego kod źródłowy jest lokalizowany w paczce ASAR na podstawie definicji zawartej w pliku `package.json`. Proces główny może tworzyć obiekty `BrowserWindow`, które tworzą nowe procesy renderujące. Komunikacja między procesami odbywa się za pomocą wbudowanego mechanizmu IPC (Rys. 3.1). IPC pozwala na wysyłanie komunikatów tekstowych do konkretnych procesów. Należy zatem zadbać, aby wszystkie dane były serializowalne, gdyż np. prototypy klas oraz funkcje nie są przekazywane.



Rys. 3.1 Schemat działania procesów w platformie Electron

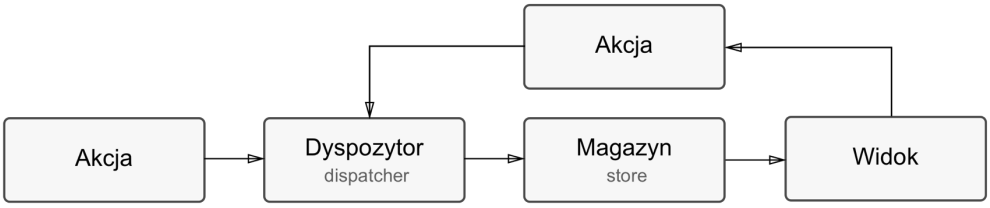
Poza udostępnianiem standardowych interfejsów programistycznych, Electron integruje się z systemami Windows, macOS i Linux, pozwalając na tworzenie aplikacji rezydujących w zasobniku systemowym, przechwytywanie zdarzeń myszy i klawiatury i wykorzystywanie innych możliwości systemów, np. wyświetlanie powiadomień, czy obsługę paska Touch Bar w komputerach z systemem macOS.

3.1.6. Architektura Flux, React i Redux

Flux to architektura służąca do tworzenia aplikacji internetowych opracowana przez zespół Facebooka. Architektura Flux nie jest odgórnie narzuconą strukturą, można ją traktować jako nieobowiązkowy wzorzec. Polega na wykorzystaniu komponentów interfejsu użytkownika w taki sposób, aby zachować jednokierunkowy przepływ danych. Komponenty mogą pobierać dane od komponentów nadrzędnych lub od komponentów dostarczających dane (komponentów typu `Provider`). W tej architekturze dane zawsze są przekazywane od korzenia drzewa do jego liści. Informacje zwrotne z komponentów zagnieżdżonych są natomiast przekazywane poprzez mechanizmy zdarzeń (ang. *callbacks*) oraz akcji.

Aplikacje realizowane w architekturze Flux najczęściej posiadają wspólną strukturę przechowującą dane, nazywaną magazynem (ang. *store*). Magazyn przechowuje stan aplikacji oraz zawiera logikę odpowiedzialną za aktualizowanie stanu na podstawie akcji. Z pojęciem *store* związany jest dyspozytor akcji (ang. *dispatcher*), który pobiera. Dyspozytor jest odpowiedzialny za zarządzanie przepływem wszyst-

kich danych w aplikacji Flux (Rys. 3.2). Sam w sobie jest jedynie rejestrem odwołań do poszczególnych magazynów i mechanizmem przekazywania do nich akcji. W przypadku aplikacji z wieloma magazynami, dyspozytor zyskuje na znaczeniu, gdyż odpowiada on za rozwiązanie zależności między nimi, wykonując akcje w odpowiedniej kolejności.



Rys. 3.2 Przepływ danych w architekturze Flux

Biblioteka Redux to implementacja mechanizmów magazynu oraz dyspozytora. Programista ma za zadanie zdefiniowanie części logicznej magazynu, czyli tzw. *reducerów*. Akcje definiują jedynie to, co się wydarzyło, natomiast *reducer* definiuje, jak powinien zmienić się stan magazynu aplikacji po otrzymaniu akcji. Nazwa *reducer* pochodzi od operacji `Array.reduce` w języku JavaScript, która redukuje tablicę do jednej wartości. Operacja ta przyjmuje początkową wartość i funkcję zwracającą nową wartość na podstawie poprzedniej wartości oraz elementu tablicy. Następnie przyjęta funkcja jest wykonywana po kolei na każdym elemencie tablicy zwracając to coraz pełniejszą wartość zredukowaną. *Reducer* działa na identycznej zasadzie, zwracając nowy stan na podstawie poprzedniego oraz wchodzącej akcji. Jeżeli akcja nie ma wpływu na dany *reducer*, wtedy, aby uniknąć kopiowania fragmentu stanu należy zwrócić stary stan. Wyżej opisany mechanizm zarządzania stanem aplikacji skutkuje utworzeniem kopii stanu po każdej zmianie (stan nie jest mutowany), z zastrzeżeniem, że nienaruszone elementy drzewa stanu nie są kopiowane bez potrzeby, a jedynie tworzone jest do nich odniesienie. Oszczędza to czas na kopiowaniu danych oraz umożliwia zachowanie w pamięci wiele stanów poprzednich. Możliwe są także „podróże w czasie”, czyli przeniesienie się do dowolnego z poprzednich stanów aplikacji, co ułatwia debugowanie oraz upraszcza implementację operacji cofnij/powtórz.

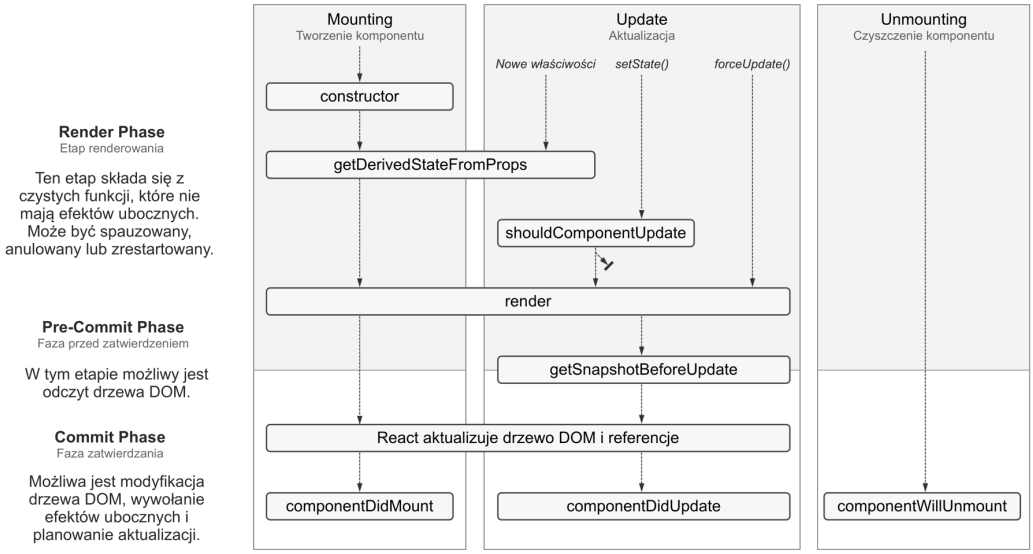
Drugą biblioteką, wykorzystywaną do prezentowania treści użytkownikowi, jest React. Ta biblioteka, opracowana przez inżynierów Facebooka, pozwala na deklaratywne budowanie interfejsu użytkownika na bazie komponentów, które posiadają

właściwości nadawane przez komponenty je okalające. Opcjonalnie komponenty mogą posiadać swój stan, np. reflektując wartości wpisane do formularza przez użytkownika. W odróżnieniu od innych bibliotek dostępnych na rynku (Angular, Vue) szablony komponentów umieszcza się bezpośrednio w nich, co pozwala na szybkie i czytelne połączenie logiki z widokiem. Dodatkowo czytelność można zwiększyć poprzez wykorzystania rozszerzenia składni JSX, które pozwala na wykorzystanie fragmentów składni XML wewnątrz kodu w języku JavaScript. Komponenty mogą być implementowane zarówno jako klasy, jak i funkcje (*stateless functional components*). Proste komponenty, wyświetlające jedynie fragment stanu aplikacji, można zrealizować poprzez funkcje, które są odpowiednikiem komponentu implementującego jedynie metodę *render* (Listing 3.2).

```
interface Props {
  id: string;
  name: string;
  avatar: string;
}

const User: React.SFC<Props> = ({
  id, name, avatar,
}) => (
  <div className={styles.container}>
    <Tooltip title={id}>
      <Avatar alt={name} src={avatar}/>
    </Tooltip>
    <Typography variant="body1">{name}</Typography>
  </div>
);
```

Listing 3.2 Przykładowy bezstanowy komponent funkcyjny, wyświetlający informacje o użytkowniku

Złożone komponenty najczęściej wymagają użycia klas, które pozwalają na przechowanie stanu i obsłużenie cyklu życia komponentu, który został zaprezentowany na Rys. 3.3.



Rys. 3.3 Cykl życia komponentu React

Połączenie bibliotek React i Redux w architekturze Flux umożliwia tworzenie modułowych i rozszerzalnych aplikacji internetowych oraz mobilnych. Przykładami aplikacji zbudowanych przy użyciu wyżej opisanej architektury są Facebook, Airbnb, Instagram i Netflix

3.2. Projekt

Projekt systemu dydaktycznego składa się z rozszerzenia dla systemu PostgreSQL oraz wieloplatformowej aplikacji służącej do łączenia z bazą danych, automatycznej instalacji rozszerzenia oraz wykonywania zapytań. Do realizacji systemu wykorzystano technologie opisane w rozdziale 3.1. Przede wszystkim są to: baza danych PostgreSQL, platforma Node.js, język TypeScript, platforma Electron oraz frameworki React i Redux.
Do zbudowania całego rozwiązania niezbędne jest środowisko Node.js, baza danych PostgreSQL wraz z dodatkami deweloperskimi. Rozszerzenie musi być budowane wielokrotnie, na każdym z systemów operacyjnych. Dodatkowo, aby skompilować rozszerzenie na platformę Windows wymagane jest środowisko programistyczne Visual Studio.
Punktem wejścia całego projektu jest plik `package.json`, zawierający informacje o zależnościach aplikacji oraz komendy odpowiadające za poszczególne etapy pracy z projektem. Najważniejsze z nich to:

- `start` – uruchamia projekt w trybie deweloperskim,
- `build` – buduje projekt w czterech etapach: przygotowuje paczkę z kodem aplikacji, przygotowuje paczkę z kodem rozszerzenia, kompiluje rozszerzenie na obecną platformę oraz tworzy pakiety z aplikacją na wszystkie systemy operacyjne,
- `clean` – usuwa z projektu pozostałości po poprzednim procesie budowania.

Wynik procesu budowania projektu przedstawia Rys. 3.4.

Rys. 3.4 Schemat zawartości paczki z projektem

3.2.1. Projekt rozszerzenia dla elementów rozmytych w systemie PostgreSQL

Rozszerzenie składa się z trzech części: ogólnej, operatorowej oraz pomocniczej. Część ogólna to definicja typu `twoint`, który jest typem pomocniczym wykorzystywanym w agregacjach wraz z operatorami Zadeha, które działają bezpośrednio na

liczbach zmiennoprzecinkowych. Druga część, operatorowa, zawiera definicję typu funkcji trapezowej `trapezoidal_function`, a także typu rozszerzającego funkcję trapezową `trapezoidal_function_ext` o wartość progową oraz operator porównania względem wartości. W tej części znajduje się również implementacja wszystkich operacji, jakie można wykonać z użyciem funkcji trapezowych. Dwie pierwsze części są oparte na pierwotnym projekcie FuzzyQ. Trzecia część to autorskie rozszerzenie pozwalające na dodawanie własnych zmiennych lingwistycznych i ich wartości rozmytych. Wewnętrznie są one określone mianami `fuzzy_type` oraz `fuzzy_function`. W tej części zdefiniowano funkcje takie jak:

- `add_fuzzy_type`/`delete_fuzzy_type` – tworzy/usuwa zmienną lingwistyczną o podanej nazwie,
- `get_fuzzy_type` – zwraca identyfikator zmiennej lingwistycznej,
- `add_fuzzy_function`/`delete_fuzzy_function` – tworzy/usuwa wartość zmiennej lingwistycznej określoną nazwą i funkcją,
- `get_fuzzy_type_functions` – zwraca wszystkie wartości zmiennej lingwistycznej,
- `get_fuzzy_function` – zwraca funkcję na podstawie nazwy zmiennej lingwistycznej i nazwy wartości,
- `get_fuzzy_name` – zwraca nazwę najlepiej dopasowanej wartości zmiennej lingwistycznej na podstawie nazwy zmiennej lingwistycznej i wartości liczbowej,
- `get_fuzzy_types` – zwraca wszystkie zmienne lingwistyczne i nazwy ich wartości,
- `get_fuzzy_functions` – zwraca wartości zmiennej lingwistycznej na podstawie jej nazwy.

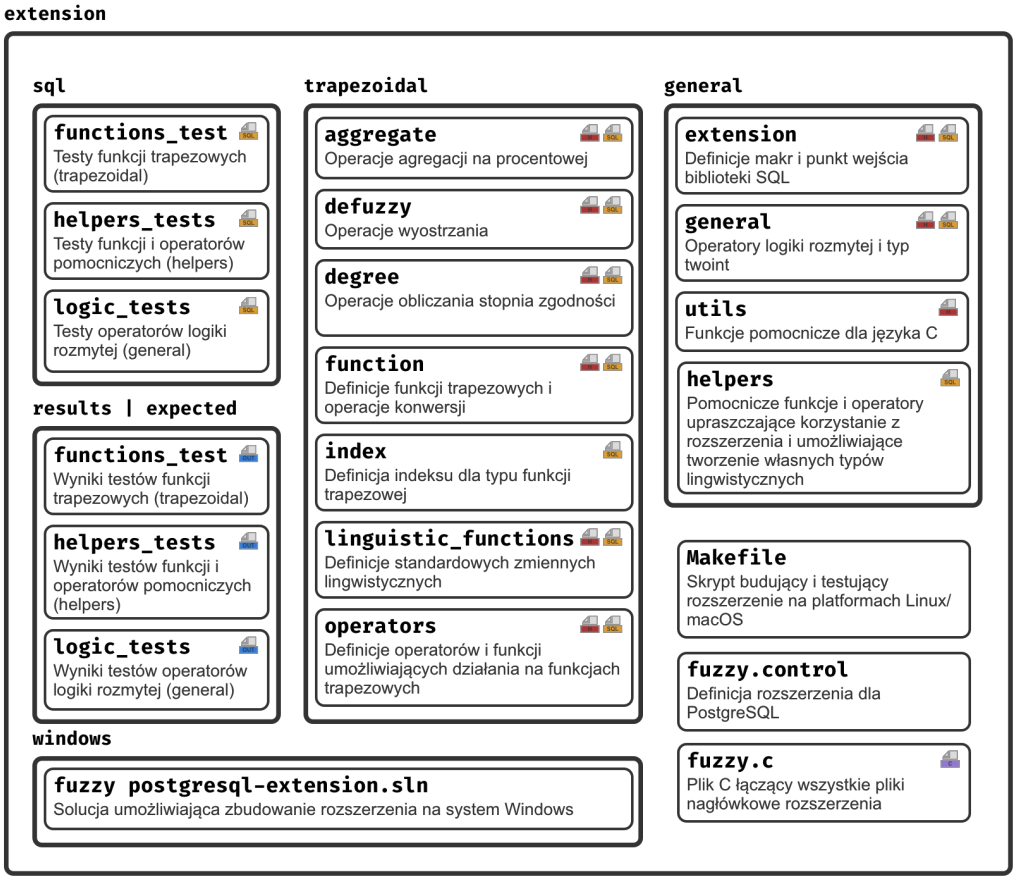
Dla uproszczenia składni poleceń, utworzono dwa operatory:

- `'type' |> 'function'` – wybiera funkcję wartości zmiennej lingwistycznej, korzystając z funkcji `get_fuzzy_function`,
- `numeric_value~>'type'` – wykorzystuje funkcję `get_fuzzy_name`, aby zwrócić najlepsze dopasowanie wartości zmiennej lingwistycznej.

Całe rozwiązanie jest spakowanym rozszerzeniem, które może być w prosty sposób zainstalowane i w założeniu ma być przenośne.

Do kompilacji rozszerzenia wykorzystano infrastrukturę PGXS – narzędzie dedykowane tworzeniu dodatków dla PostgreSQL [26]. Poza kompilacją, pakowaniem oraz instalacją rozszerzeń, PGXS umożliwia wykonywanie testów regresyjnych.

Dzięki tej funkcjonalności, całe rozszerzenie jest najpierw ręcznie sprawdzane pod kątem błędów, a następnie wyniki testów są zapisywane w projekcie. Po dokonaniu zmian w kodzie można mieć pewność, iż stare i zaktualizowane funkcje rozszerzenia działają tak samo jak poprzednio.
Rozszerzenie opiera się na kodzie biblioteki w języku C oraz skryptach SQL, które odwołują się do funkcji zdefiniowanych w bibliotece. Kod projektu został pogrupowany tak, aby logicznie spójne części znajdowały się w tych samych plikach. Niestety, z uwagi na utrudnioną konfigurację linkera w środowisku PGXS, można było wykorzystać tylko jeden plik C. Nie przeszkodziło to jednak w wydzieleniu kodu, który znajduje się w plikach nagłówkowych. Jest to antywzorzec, jednak znacząco upraszcza kompilację i pozwala na zachowanie wspomnianej logicznej separacji. Strukturę plików przedstawia Rys. 3.5.



Rys. 3.5 Struktura plików rozszerzenia Fuzzy dla PostgreSQL

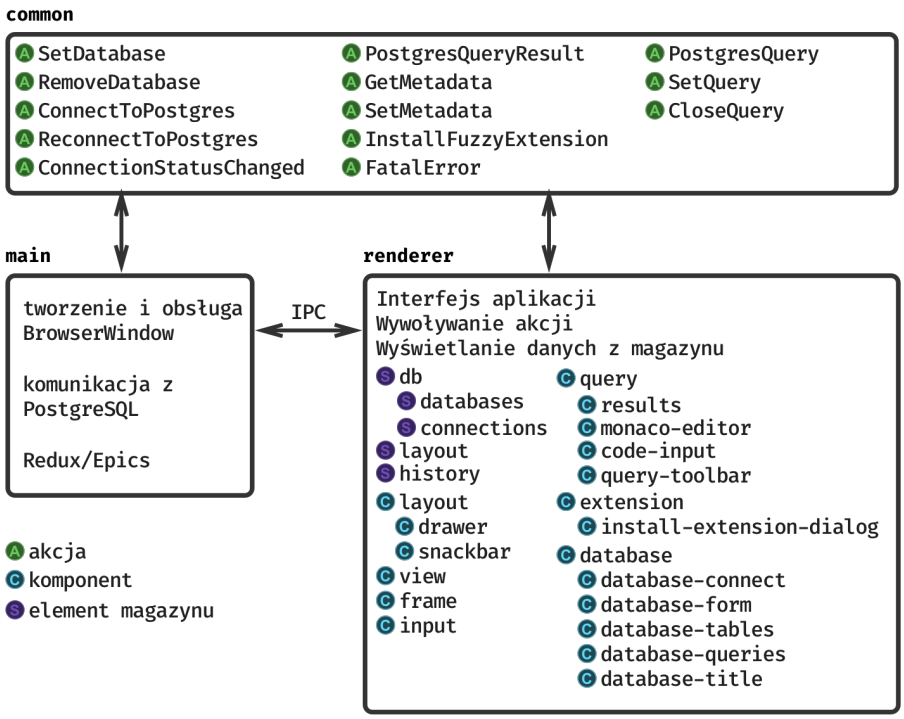
3.2.2. Architektura aplikacji klienckiej

Z uwagi na prostotę działania i przenośność, zdecydowano się skorzystać z platformy Electron, na której oparto aplikację Fuzzybase, zbudowaną w technologiach React i Redux. Mechanizm akcji Redux znalazł swoje zastosowanie w komunikacji międzyprocesowej. Część aplikacji komunikująca się z bazą danych znajduje się w osobnym procesie. W związku z tym proces przeglądarkowy współdzieli mechanizm dyspozytora i przekazuje wszystkie akcje do procesu głównego. Magazyn procesu główny nie przechowuje natomiast żadnych wartości. Wykorzystywany jest do konsumowania akcji związanych z bazą danych i wysyłania akcji zwrotnych z informacjami pochodzącymi z serwera.
Do magazynu jest wpięte oprogramowanie pośredniczące (ang. *middleware*) w postaci biblioteki `redux-observable`, która jest w stanie obsługiwać tzw. efekty uboczne (ang. *side-effects*). Biblioteka ta opiera swoje działanie na paradygmacie reaktywnego programowania, dostarczanego przez bibliotekę `rxjs`. Programowanie reaktywne charakteryzuje się podejściem funkcyjnym oraz wykorzystaniem danych wejściowych jako strumień. Wejściem w przypadku zastosowania biblioteki `redux-observable` są akcje przychodzące do magazynu, a funkcja przetwarzająca ich strumień to *Epic*. Każdy *Epic* odbiera wszystkie akcje wchodzące do magazynu, wobec czego należy skorzystać z operatora `ofType` lub `filter` i wyfiltrować akcję, która zostanie obsłużona. Podczas obsługi akcji powszechne jest korzystanie z operatorów: `map`, `switchMap`, `tap`, `concat` i `merge`, co ma na celu transformację akcji wejściowej w jedną lub więcej akcji wyjściowych. Przykładem może być *Epic* obsługujący zapytania do bazy danych (Listing 3.3), który wykonuje zapytanie korzystając z przekazanego identyfikatora połączenia oraz informacji dotyczących zapytania, a następnie obsługuje błędy.

```
const errorHandler = (err) => of(fatalError(err));
const handleEventualErrors = result => result.type ===
DB_ACTIONS.FATAL_ERROR
  ? of(result)
  : (of(result).pipe(ignoreElements()));

const query$ = (action$: ActionsObservable<DbAction>) => action$
  .ofType<PostgresQueryAction>(DB_ACTIONS.QUERY)
  .pipe(
    tap(action => ConnectionManager.getConnection(action.connectio-
nId).query(action.queryId, action.query)),
    catchError(errorHandler),
    switchMap(handleEventualErrors)
  );
```

Listing 3.3 Epic odpowiadający za wykonywanie zapytań do bazy danych

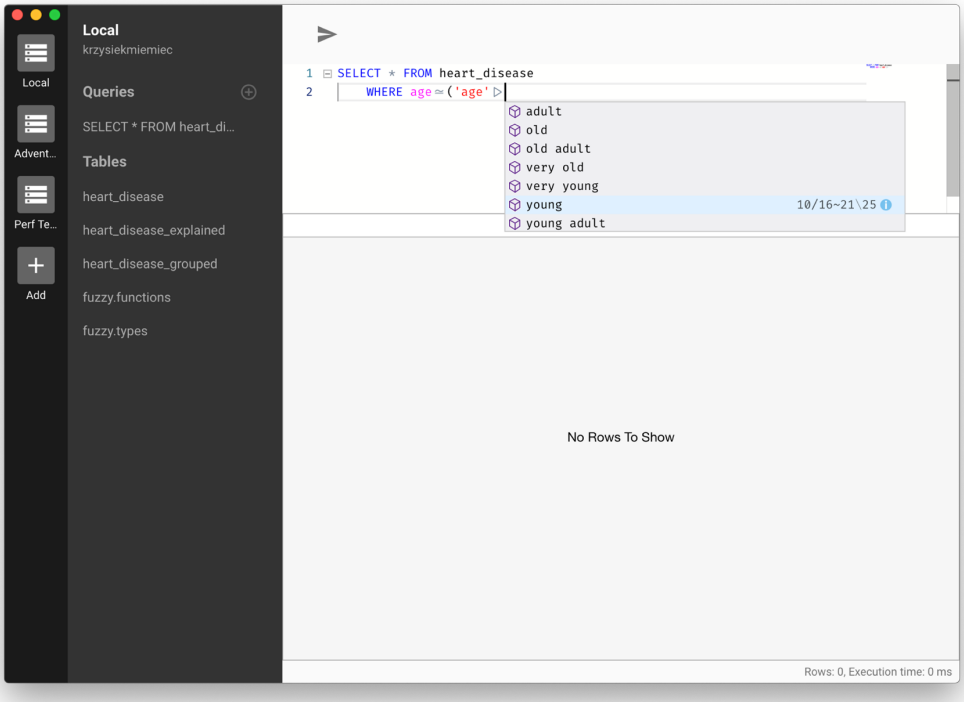
Schemat (Rys. 3.6) pokazuje architekturę rozwiązania, na którą składają się: magazyny, komponenty interfejsu użytkownika (w tym edytor kodu i komponent wyświetlający wyniki zapytania), akcje wraz z efektami ubocznymi i mechanizm komunikacji międzyprocesowej.



Rys. 3.6 Uproszczony schemat architektury aplikacji Fuzzybase

3.2.3. Edytor kodu SQL

Aplikacja Fuzzybase pozwala na tworzenie zapytań SQL z elementami rozmytymi, które są dodatkowo podpowiadane. Interfejs edytora oraz podpowiedzi został przedstawiony na Rys. 3.7. Komponent edytora kodu został oparty o edytor Monaco. Edytor Monaco jest biblioteką wygenerowaną wprost ze źródeł środowiska programistycznego VS Code. W edytorze dostępne są wszystkie mechanizmy edycji kodu znane ze środowiska VS Code, takie jak podpowiedzi kontekstowe, zwijanie kodu, zaawansowane formatowanie kodu (ligatury), podgląd kodu z prawej strony i obsługa wielu języków programowania. Rozszerzenia można tworzyć bez większych problemów, gdyż edytor został napisany w języku TypeScript i posiada pełne definicje wszystkich interfejsów.



Rys. 3.7 Ekran tworzenia zapytania i podpowiedzi kontekstowe

Najtrudniejszym zadaniem było zbudowanie podpowiedzi kontekstowych, które opierają się jedynie na pozycji kursora i dokumencie, w którym się on znajduje. Należało wstępnie wykonać serię zapytań do bazy danych, aby na podstawie ich

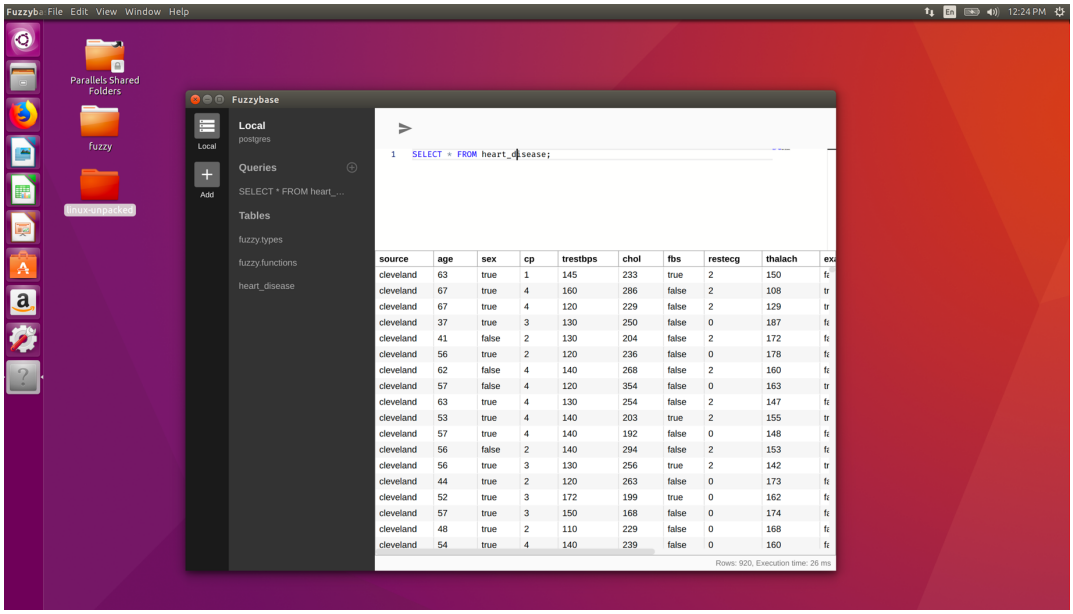
wyników utworzyć metadane bazy, zawierające typy rozmyte i ich funkcje lingwistyczne. Dzięki temu, gdy kursor występuje po operatorze określania przynależności (`~=`) można zaprezentować użytkownikowi listę typów. Generowana lista składa się z obiektów zawierających nazwę, typ, opis oraz zmianę, jaka ma zostać wstrzyknięta w miejsce kursora. Wspierane jest także podpowiadanie wyników częściowych, gdy użytkownik wpisze już część frazy. Gdy kursor występuje po operatorze wyboru funkcji lingwistycznej z typu (`|>`), algorytm podpowiedzi musi wyłuskać poprzedzającą operator nazwę typu, a następnie zwrócić listę podpowiedzi dotyczących typu (Rys. 3.7).

Aplikacja Fuzzybase może w przyszłości zostać poszerzona o podpowiedzi kontekstowe dotyczące wszystkich elementów języka. Wymagane do tego są metadane dotyczące nie tylko typów rozmytych, ale także wszystkich tabel, kolumn, funkcji, agregacji i operatorów dostępnych w bazie danych. Obecnie metadane są odświeżane po każdym zapytaniu do bazy danych, jednak nie będzie to możliwe, gdy zostaną poszerzone o kolejne zbiory wartości. Następnym krokiem byłoby napisanie analizatora języka, który utworzy drzewo składniowe (drzewo AST, ang. *abstract syntax tree*) na podstawie kodu znajdującego się w edytorze. Podczas tworzenia analizatora, dużym wyzwaniem jest optymalizacja jego działania. W przypadku wbudowanej analizy i podpowiedzi języka TypeScript, który obsługuje jednocześnie język JavaScript w edytorze Monaco, odpowiada za to zewnętrzny wątek roboczy, na którym działa silnik języka TypeScript. Zadanie analizy języka na podstawie ciągu znaków jest nietrywialne, jednak edytor Monaco posiada wbudowany mechanizm dzielenia tekstu na tokeny, który na podstawie wyrażeń regularnych zawartych w definicji języka, jest w stanie zwrócić słowa, wraz z przypisanym do nich typem. Tokenem może być np. operator, identyfikator, białe znaki lub słowo kluczowe. Dzięki tokenizacji, dopasowanie tekstu do klas reprezentujących składnię jest znacznie prostsze. Na podstawie utworzonego drzewa, złożonego z odpowiednich klas (np. klasa `SelectClause` zawierająca `FromClause` i `WhereClause`), można wydobyć kontekst, w jakim znajduje się kursor, a następnie zaproponować podpowiedzi na jego podstawie.

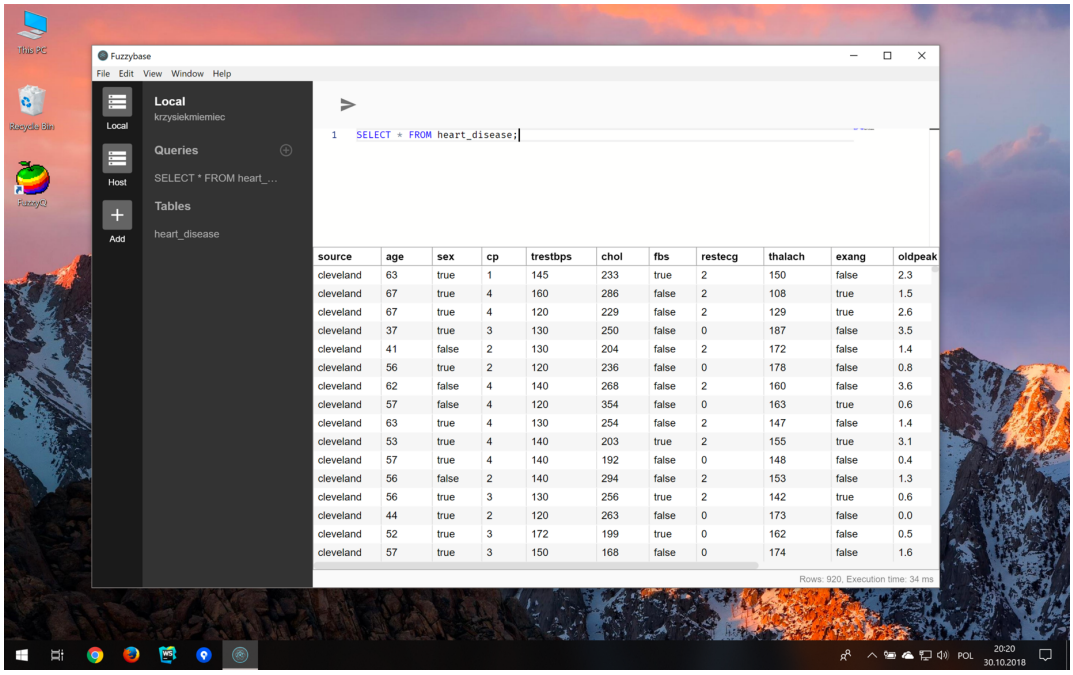
3.2.4. Działanie systemu dydaktycznego na wielu platformach

Jednym z założeń pracy było stworzenie rozwiązania, które jest przenośne i można z niego skorzystać na wszystkich popularnych systemach operacyjnych – Windows, Linux oraz macOS. Do testów wykorzystano system macOS Mojave wraz z systemami Windows 10 oraz Ubuntu 16.04, działającymi na maszynach wirtualnych Parallels.

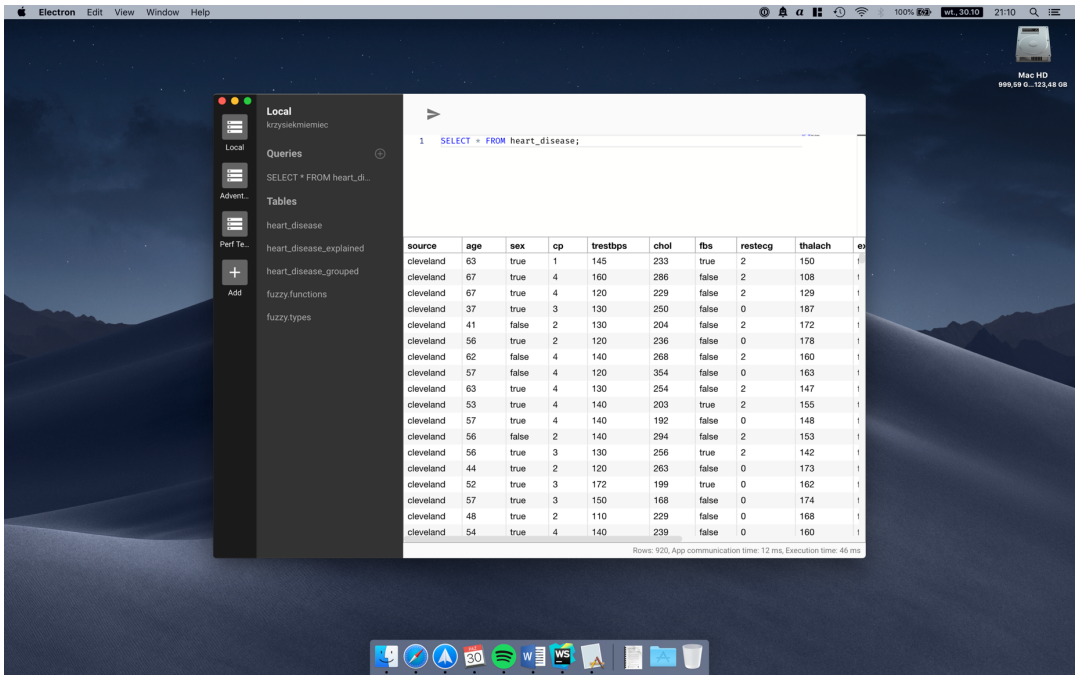
Do zbudowania wieloplatformowej aplikacji opartej o środowisko Electron wystarcza narzędzie `electron-builder` [27], które tworzy pliki wykonywalne (`*.exe` dla Windows, `*.app` dla macOS, plik binarny dla systemu Linux) zawierające wewnątrz (lub w tym samym folderze) paczki `*.asar` zawierające kod aplikacji w języku JavaScript oraz zasoby (kroje pisma, obrazy, załączone pliki statyczne). Narzędzie, po odpowiednim skonfigurowaniu, tworzy również pakiety instalatorów (obraz `*.dmg` dla macOS, pakiet `*.deb` dla systemów Debian/Ubuntu i instalator NSIS dla Windows). Interfejs aplikacji jest taki sam na każdej z platform, jednak samo okno i menu kontekstowe różnią się w zależności od wykorzystywanego środowiska graficznego. Różnice przedstawione są na zrzutach ekranu (Rys. 3.8, Rys. 3.9, Rys. 3.10).



Rys. 3.8 Zrzut ekranu systemu dydaktycznego Fuzzybase w systemie Ubuntu 16.04



Rys. 3.9 Zrzut ekranu systemu dydaktycznego Fuzzybase w systemie Windows 10



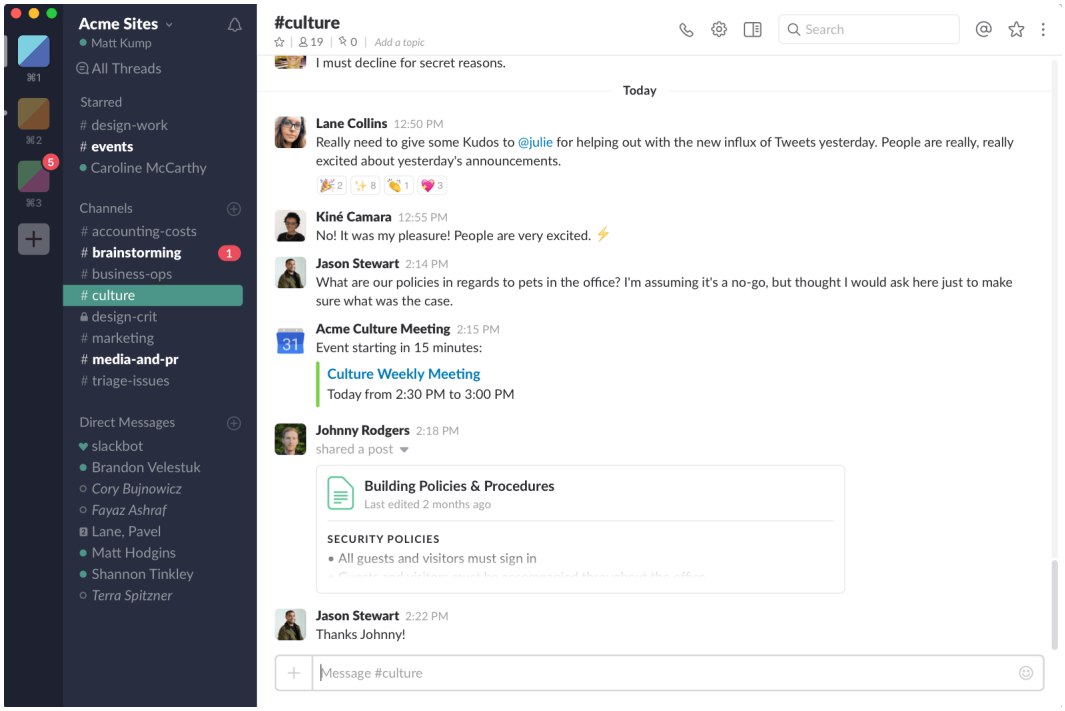
Rys. 3.10 Zrzut ekranu systemu dydaktycznego Fuzzybase w systemie macOS

Aplikacja Fuzzybase posiada wbudowany mechanizm instalacji rozszerzenia bazy danych PostgresSQL. Aplikacja podczas łączenia z bazą i wykonywania w niej zapytań, pobiera metadane dotyczące rozszerzenia i typów rozmytych. Jeżeli rozszerzenie nie zostanie znalezione, dostępna jest opcja automatycznej instalacji. Instalacja rozpoczyna się rozpoznaniem, czy baza danych jest dostępna lokalnie, czy użytkownik łączy się ze zdalnym hostem. W drugim przypadku oferowana jest jedynie paczka z rozszerzeniem, które można samodzielnie skompilować na docelowej maszynie, gdyż zdalne zarządzanie maszynami o nieznanej architekturze byłoby czasochłonne w implementacji. W przypadku odnalezienia bazy lokalnej, następuje próba instalacji rozszerzenia przy użyciu polecenia w języku SQL `CREATE EXTENSION fuzzy`. W razie niepowodzenia, podejmowana jest próba skopiowania plików o wybranej architekturze i platformie do folderu współdzielonego oraz folderu rozszerzeń bazy danych. Lokalizację tych folderów uzyskuje się poprzez wykonanie polecenia `pg_config` instalowanego razem z bazą. Polecenie `pg_config` może nie znajdować się pod ścieżkami w zmiennej środowiskowej `PATH`, w związku z czym wykorzystano uniwersalny, wieloplatformowy pakiet Node.js `which`, który działa tak jak standardowe polecenie Unixa o tej samej nazwie. Jeżeli pliki nie mogą zostać skopiowane, zakłada się, że problem wystąpił z powodu brakujących uprawnień administratora. Aplikacja Fuzzybase kopiuje więc pliki rozszerzenia do lokalizacji tymczasowej, po czym pyta użytkownika o hasło, wskazując na uruchamiany skrypt kopiujący rozszerzenie do lokalizacji docelowej. Po tej procedurze aplikacja ponawia próbę instalacji rozszerzenia i informuje użytkownika o rezultacie operacji.

Opisany proces instalacji pozwala na skrócenie czasu potrzebnego do przygotowania stanowiska do pracy z zapytaniami zawierającymi elementy rozmyte. W przypadku niewspieranej platformy (najczęściej odmiany systemu Linux, lub systemu o niewspieranej architekturze) oferowane są pliki źródłowe, dzięki którym użytkownik może ręcznie zbudować i zainstalować rozszerzenie.

3.2.5. Interfejs użytkownika

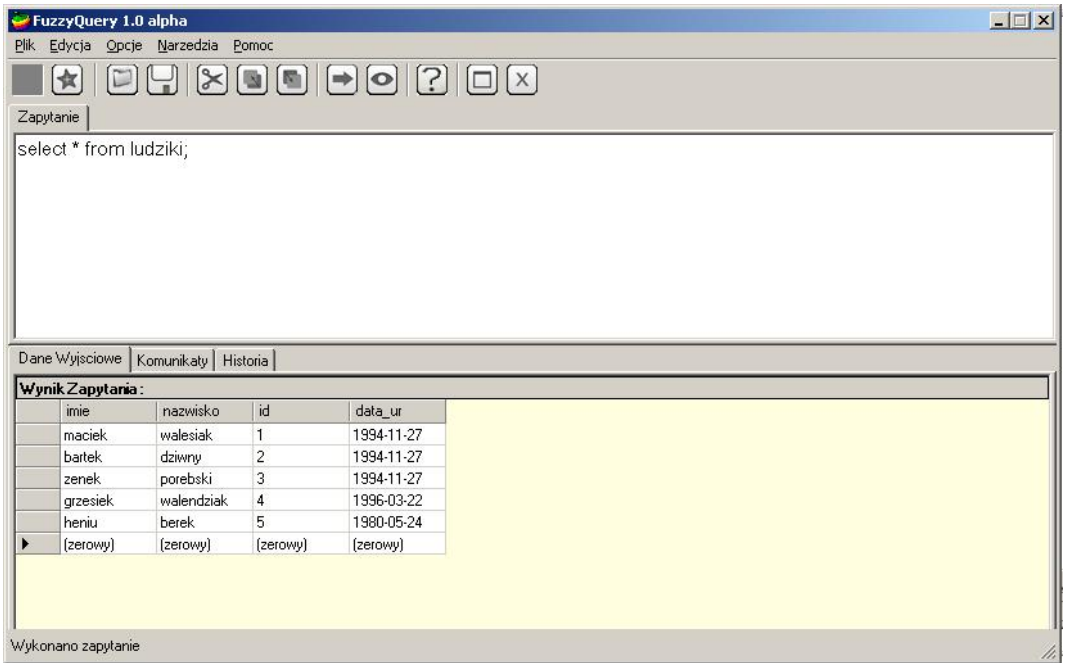
Doświadczenie użytkownika (UX, ang. *user experience*) jest całokształt wrażeń, jakich użytkownik doświadcza podczas korzystania z produktu interaktywnego. W ostatnich latach jest kładziony większy nacisk na wygodę korzystania z aplikacji, a dbanie o doświadczenie użytkownika stało się jednym z podstawowych elementów pracy nad oprogramowaniem. Postanowiono zatem, że produkt niniejszej pracy magisterskiej będzie się wyróżniał przejrzystością i prostotą działania dla grupy docelowej, jaką są studenci i prowadzący zajęcia laboratoryjne. Przyjęto, że użytkownicy z grupy docelowej są zaznajomieni z mechanizmem podpowiedzi kontekstowych oraz popularnymi komunikatorami tekstowymi. Biorąc pod uwagę owe założenia, opracowano interfejs podobny do tego, jaki można zaobserwować w komunikatorze biurowym Slack (Rys. 3.11) i zestawiono go z edytorem kodu wspierającym podkreślanie składni oraz rozbudowane skróty klawiszowe i podpowiedzi.



Rys. 3.11 Okno komunikatora biurowego Slack [28]

Wynikiem tego połączenia jest interfejs aplikacji widoczny na Rys. 3.8, Rys. 3.9 i Rys. 3.10. Umożliwia on skonfigurowanie wielu baz danych, do których można

być jednocześnie podłączonym. Można także edytować wiele zapytań do bazy jednocześnie. Konfiguracje baz danych są przechowywane na dysku, zatem wystarczy je utworzyć tylko raz. Dodatkowo, aplikacja umożliwia sortowanie i filtrowanie zwróconych wcześniej wyników zapytania. W zestawieniu z narzędziem FuzzyQ, ułożenie elementów wydaje się być podobne, jednak zaawansowany edytor kodu zastępujący graficzny kreator szablonu zapytania daje większą swobodę podczas tworzenia zapytań, a widok tabeli posiada szerszy wachlarz możliwości prezentacji danych. Ponadto program FuzzyQ nie jest zgodny z popularnym na uczelniach systemem Linux, a jego interfejs trudniej odnieść od realnych przykładów aplikacji wykorzystywanych na co dzień przez grupę docelową.



Rys. 3.12 Okno aplikacji FuzzyQ prezentujące widok zapytania i wyników

4. Badania

W niniejszym rozdziale przedstawiono badania wydajności utworzonego rozwiązania oraz opisano problemy, które wystąpiły podczas realizacji projektu. Testy i pomiary dotyczą wydajności operatorów zaimplementowanych w części bazodanowej (rozszerzenie PostgreSQL), a także części aplikacyjnej i są ustosunkowane do liczby przetwarzanych danych.

W pierwszej części są opisane testy wydajności rozszerzenia, których wyniki powinny być zadowalające pod względem czasu wykonania oraz zbliżone do zapytań ostrych, gdyż rozszerzenie zostało zrealizowane w niskopoziomowym języku C. Ze względu na fakt, że zapytania ostre zostały zoptymalizowane i uproszczone, a złożoność obliczeniowa elementów rozmytych w zapytaniach jest większa niż złożoność elementów ostrych, wyniki przemawiają na korzyść zapytań ostrych.

Kolejna część badań dotyczy aplikacji Fuzzybase. Jest to opracowany klient bazy danych PostgreSQL ze wsparciem dla elementów rozmytych w zapytaniach SQL. Aspekt wydajności aplikacji jest istotny ze względu na użyte technologie Electron i Node.js, które nie zostały dobrze zbadane w takich zastosowaniach. W sieci można znaleźć wiele opinii oraz testów syntetycznych udowadniających, że specyfika platformy JavaScript pozwala uzyskać przewagę w wydajności nad innymi platformami. Tyle samo jest testów negujących przewagę tej architektury, ze szczególnym uwzględnieniem platformy Electron, która jest piętnowana jako zużywająca wiele zasobów i niewydajna, co jest spowodowane brakiem pamięci współdzielonej oraz koniecznością tworzenia osobnych procesów. Celem badania jest sprawdzenie, który czynnik (komunikacja międzyprocesowa, odbiór danych, rendering) ma największy wpływ na wydajność aplikacji.

Dalsza część bada zasadność stosowania elementów rozmytych podczas pracy z dużą ilością danych i pokazuje wpływ ich wykorzystania na liczbę zwracanych wyników zapytania. Na podstawie tabeli zawierającej dane medyczne, pokazana jest różnica między zbiorami uzyskanymi poprzez zapytania ostre, a zbiorami zawierającymi dodatkowe informacje ze względu na rozmycie warunków.

Na koniec poruszono także problemy, które napotkano podczas rozwijania aplikacji klienckiej Fuzzybase oraz rozszerzenia Fuzzy dla systemu zarządzania bazą danych PostgreSQL. Opisane zostały wyzwania dotyczące tworzenia aplikacji wieloplatformowej, poprawiania wydajności oraz uproszczenia procesu kompilacji i instalacji rozszerzenia, które powinno działać na trzech, różniących się od siebie systemach operacyjnych. Dla każdej z kwestii zostało zaproponowane rozwiązanie techniczne, lecz zagadnienia dotyczące optymalizacji nie zostały zrealizowane w projekcie.

4.1. Metodyka badań wydajności operatorów

Do przeprowadzenia badań wydajności operatorów wykorzystano skrypt powłoki, który uruchamia polecenia w bazie PostgreSQL, a także skrypt w języku SQL przygotowujący bazę do testów. Listing 4.1 przedstawia bazową część skryptu powłoki.

```
#!/usr/bin/env bash
set -e

PGDATABASE=perf_test
PGPASSWORD=perf_test

sql="psql -d ${PGDATABASE}"

function time_fn() {
  echo "${1}"
  x=1
  while [ $x -le 20 ]; do
    $sql -c '\timing' -c "${1}" | grep -e '^Time: [0-9]' | sed 's/Time:
//'
    x=$(( $x + 1 ))
  done
}
```

Listing 4.1 Fragment skryptu powłoki wykonujący zapytania i mierzący ich czas

Poza ustawieniem zmiennych środowiskowych pozwalających na dostęp do bazy, w skrypcie zdefiniowano polecenie `sql` upraszczające wykonywanie poleceń na kon-

kretnej bazie. Funkcja `time_fn` odpowiada za wypisanie polecenia, które jest wykonywane, a następnie dokonuje 20 zapytań, z których wyłuskiwany jest czas wykonania. Przykładowe użycie funkcji wraz z wynikiem przedstawia listing 4.4.

```
time_fn "SELECT "'*'" FROM heart_disease WHERE age~=('age'|>'old')>0;"
358.661 ms # Pierwszy z pomiarów, po nim następuje ich jeszcze 19.
```

Listing 4.2 Przykładowe wywołanie funkcji pomiarowej

Badaniom poddano trzy rodzaje zapytań `SELECT`. Pierwsze zapytanie zawiera jeden prosty warunek rozmyty we frazie WHERE, który w celu sprawdzenia wydajności rozwiązań został zestawiony z wyrażeniami porównującymi przedział. Drugie zapytanie to instrukcja rozmyta grupująca zbiór na podstawie funkcji lingwistycznych, która tak samo jak w pierwszym teście jest porównywana z zapytaniem klasycznym. Trzecie zapytanie grupuje wyszukiwany zbiór danych na podstawie wyników konkretnej funkcji lingwistycznej i również jest ono zestawione ze swoim odpowiednikiem nie będącym wyrażeniem rozmytym. Platforma sprzętowa, na której uruchomiono testy to MacBook Pro 2016 z procesorem Intel Core i7 2,7 GHz i 16 GB pamięci RAM. Wyniki czasowe testów uzyskane na innych platformach mogą się różnić, jednak relatywne czasy powinny się pokrywać.

Pojedyncze pomiary czasu różnią się od siebie ze względu na to, że na platformie testowej działają inne, niezależne usługi, mogące wykorzystywać zasoby procesora lub pamięci spowalniając zapytania. Poza obciążeniem wynikającym z czynników zewnętrznych, rolę odgrywają też optymalizacje i zapisywanie części danych w pamięci podręcznej przez system PostgreSQL. Aby mierzona wydajność była możliwie zbliżona do tej w warunkach praktycznych, każdy wynik jest uśrednieniem 20 pomiarów i jest zestawiony z odchyleniem standardowym celem zobrazowania zakresu zmierzonych wartości. Pozwala to na określenie czy różnica wydajności jest znacząca, czy jedynie jest efektem wpływu czynników zewnętrznych i stopniowo dokonywanych optymalizacji systemu zarządzania bazą danych.

4.2. Zbiór danych testowych

Za dane testowe, wykorzystane do zapełnienia bazy, przyjęto dostępne publicznie wyniki badań dotyczące chorób serca [29]. Przed uruchomieniem testów baza testowa jest czyszczona i wypełniana danymi, które następnie są duplikowane, aż tabela w bazie osiągnie rozmiar około miliona rekordów. Taką ilość danych uznano za wystarczająco dużą, aby zaobserwować różnice w czasie wykonania zapytań (jeżeli takowe występują).

source	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	num
switzerland	51	t	4	110	0		0	92	f	0,0	2			4
cleveland	67	f	4	106	223	f	0	142	f	0,3	1	2	3	0
hungary	35	f	1	120	160	f	1	185	f	0,0				0
long_beach	55	t	4	150	160	f	1	150	f	0,0				0
cleveland	56	t	3	130	256	t	2	142	t	0,6	2	1	6	2
hungary	50	t	4	145	264	f	0	150	f	0,0				1

Tabela 4.1 Fragment zbioru danych heart_disease

Testowana tabela składa się z 15 kolumn opisujących pacjentów badanych pod kątem chorób serca (Tabela 4.1). Kolumny to:

- `source` – źródło informacji o pacjencie (lokalizacja placówki, w którym dokonano badań)
- `age` – wiek pacjenta
- `sex` – płeć pacjenta
- `cp` – rodzaj występującego bólu klatki piersiowej (1 – typowy duszący, 2 – atypowy duszący, 3 – inny, 4 – brak bólu)
- `trestbps` – ciśnienie krwi w stanie spoczynku
- `chol` –poziom cholesterolu we krwi
- `fbs` – zalecany poziom cukru we krwi określony jako przekroczony/nieprzekroczony

- `restecg` – wynik EKG, określony jako jedna z trzech wartości (0 - normalny, 1 - nieprawidłowy przebieg fali, 2 - hipertrofia)
- `thalach` – liczba uderzeń serca na minutę osiągnięta podczas wysiłku
- `exang` – ból po wysiłku (wartości prawda/fałsz)
- `oldpeak` – spadek fali EKG po wysiłku
- `slope` – kształt odcinka ST fali EKG po wysiłku (1 – rosnący, 2 – płaski, 3 – malejący)
- `ca` – liczba naczyń krwionośnych zaznaczonych podczas fluoroskopii
- `thal` – wada określana jako (3 – brak, 6 – nieodwracalna, 7 – odwracalna)
- `num` – diagnoza choroby serca (0 – brak, >1 występująca wada)

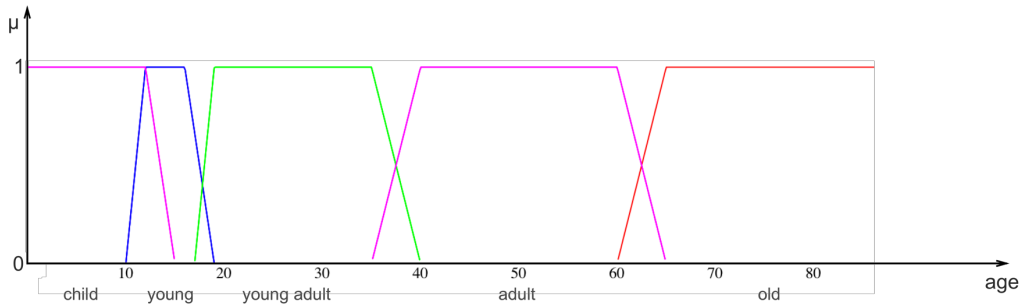
Z wymienionych wcześniej parametrów wyselekcjonowano trzy, do przetwarzania których można wykorzystać zmienne lingwistyczne i wartości rozmyte, są to: `age`, `trestbps`, `chol` oraz `thalach`.

Zmienne lingwistyczne i ich wartości można definiować działając bezpośrednio na bazie danych, bez konieczności rekompilacji rozwiązania, tak jak miało to miejsce w implementacji FuzzyQ. Przykład definicji zmiennej lingwistycznej znajduje się w listingu 4.3. Podczas wykonywania wielu funkcji, można skorzystać z klauzuli `UNION`, która złączy wyniki jako kolejne wiersze. Można również użyć krótszej składni i oddzielić funkcje przecinkami, jednak wyniki zostaną wtedy zwrócone jako kolumny. Zdefiniowano zatem następujące zmienne lingwistyczne i ich wartości na podstawie danych określonych przez ekspertów z dziedziny:

Zmienna lingwistyczna `age` (Listing 4.3, Rys. 4.1)

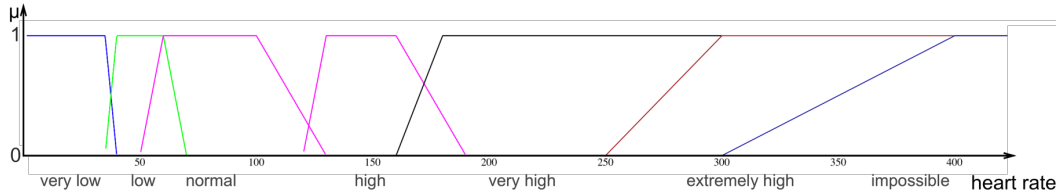
```
SELECT add_fuzzy_function('age', '0~16\19', 'child')
UNION SELECT add_fuzzy_function('age', '10/12~16\19', 'young')
UNION SELECT add_fuzzy_function('age', '17/19~39\45', 'young adult')
UNION SELECT add_fuzzy_function('age', '35/40~60\65', 'adult')
UNION SELECT add_fuzzy_function('age', '60/65~200', 'old');
```

Listing 4.3 Zapytanie języka SQL definiujące zmienną lingwistyczną age



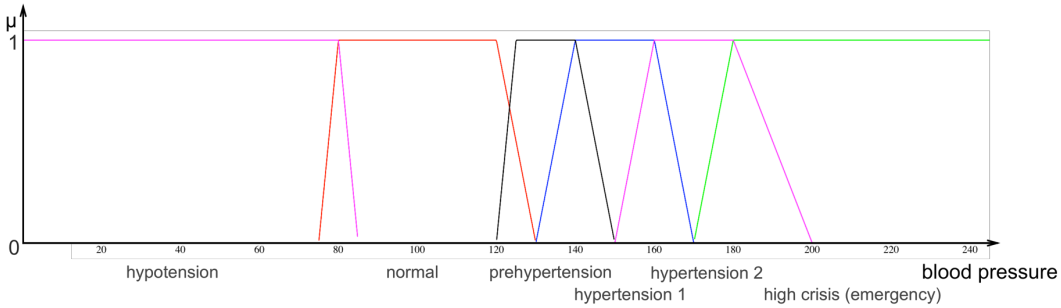
Rys. 4.1 Wykres prezentujący wartości zmiennej lingwistycznej age

Typ `heart_rate` (Rys. 4.2)



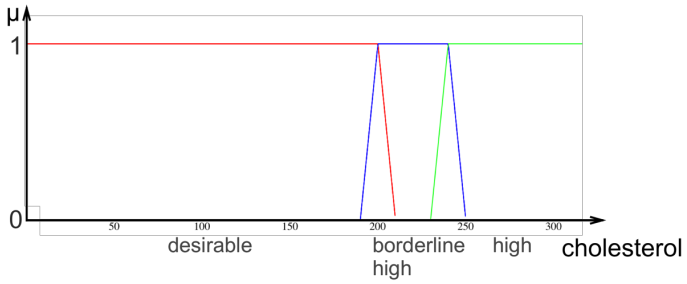
Rys. 4.2 Wykres prezentujący wartości zmiennej lingwistycznej `heart_rate`

Typ `blood_pressure` *(Rys. 4.3)*



Rys. 4.3 Wykres prezentujący wartości zmiennej lingwistycznej `blood_pressure`

Typ `cholesterol` (Rys. 4.4)



Rys. 4.4 Wykres prezentujący wartości zmiennej lingwistycznej `cholesterol`

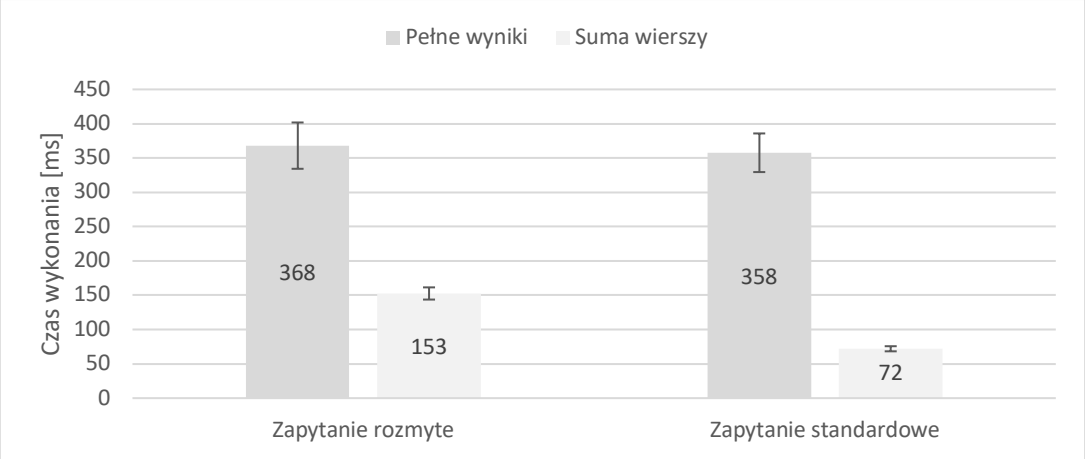
4.3. Wyniki pomiaru wydajności operatorów rozmytych

Kod pierwszych, prostych zapytań jest przedstawiony w listingu 4.4. Zapytanie ostre wykorzystuje możliwie prostą implementację zapytania rozproszonego, która obliczeniowo jest mniej złożona od operacji przeprowadzanych wewnętrznie podczas określania wartości zmiennej rozmytej oraz porównania jej. Można spodziewać się więc, że zapytanie rozmyte będzie wykonywało się dwukrotnie wolniej niż zapytanie klasyczne.

```
echo Fuzzy age query
time_fn "SELECT "'*'" FROM heart_disease WHERE age~=('age'|>'old')>0;"

echo Standard age query
time_fn "SELECT "'*'" FROM heart_disease WHERE age>60 AND age<120;"

# COUNT
echo Fuzzy age query
time_fn "SELECT COUNT("'*'") FROM heart_disease WHERE
age~=('age'|>'old')>0;"

echo Standard age query
time_fn "SELECT COUNT("'*'") FROM heart_disease WHERE age>60 AND
age<120;"
```

Listing 4.4 Kod prostego zapytania testowego

Na wykresie na Rys. 4.5, zostały przedstawione uśrednione czasy wykonania wraz z odchyleniem standardowym. W zależności od tego, czy zwracane były kompletne wyniki, czy też tylko liczba wyfiltrowanych wierszy, czasy wykonań znacząco różniły się od siebie. W przypadku zwracania kompletnych wyników (wszystkie wiersze w całości), różnica w czasie wykonania nie przekracza wartości odchylenia standardowego. Natomiast, gdy wynikiem jest jedynie suma znalezionych wierszy, czas wykonania jest ponad dwukrotnie krótszy w przypadku zapytania ostrego, co pokrywa się z przypuszczeniami. W tym miejscu warto przytoczyć kod funkcji `degree`, która odpowiada za obliczenie stopnia przynależności do funkcji (Listing 4.5).



Rys. 4.5 Wykres czasu wykonania prostych zapytań

```
float8 degree(float8 x, trapezoidal_function f) {
    float8 l = f.m - f.a;
    float8 m = f.m;
    float8 n = f.m + f.dm;
    float8 o = f.m + f.dm + f.b;

    if (x >= m && x <= n) {
        return 1;
    }
    if (x > l && x < m) {
        return (x - l) / (m - l);
    }
    if (x > n && x < o) {
        return (o - x) / (o - n);
    }
    return 0;
};
```

Listing 4.5 Kod funkcji obliczająca stopień przynależności

W optymistycznym przypadku, wykonane zostaną 4 operacje arytmetyczne oraz 2 operacje porównania, a w pesymistycznym 7 operacji arytmetycznych i 6 operacji porównania. Do tego wynik obliczenia stopnia zgodności musi zostać porównany z wartością z zapytania `SELECT`. W przypadku zapytania ostrego są to jedynie dwa porównania.

Na podstawie przeprowadzonego testu można wysnuć wniosek, że mimo dłuższego czasu wykonania, rozszerzenie dokonuje obliczeń znacznie szybciej, niż gdyby zostały one zapisane w języku SQL. Gdy zależy nam jedynie na agregacji wyników, zaleca się optymalizować zapytanie, jeżeli jest to możliwe. Rozszerzenie Fuzzy może w przyszłości zostać dostosowane w ten sposób, aby optymalizować wykorzystanie prostych funkcji trapezowych do instrukcji sprawdzających jedynie obliczony wcześniej przedział.

Listing 4.6 prezentuje kod zapytań wykorzystanych w drugim teście. Zapytania polegają na utworzeniu krótkiego zestawienia przyporządkowującego liczbę osób do danego przedziału ciśnienia tętniczego. Jak widać, zapytanie bez elementu rozmytego jest dłuższe. Jednocześnie, znając wewnętrzną implementację operatora przyporządkowania nazwy funkcji lingwistycznej na podstawie wartości, można przypuszczać, iż uproszczone zapytanie wykona się wielokrotnie szybciej. W przypadku elementu rozmytego, dla każdego przetwarzanego wiersza pobierane są funkcje lingwistyczne typu rozmytego, a następnie sprawdzany jest stopień przynależności do każdej z funkcji. Proponowana jest tu optymalizacja pozwalająca na przechowanie typu rozmytego jako zbiór uproszczonych zakresów, mogących zwrócić w wyniku identyfikator znalezionej funkcji. Wynik zapytania wykonanego w aplikacji Fuzzybase znajduje się na Rys. 4.6.

```
echo Fuzzy group by pressure query
time_fn "SELECT
    trestbps~>'blood_pressure' AS pressure,
    COUNT("'*'") AS COUNT
    FROM heart_disease
    GROUP BY pressure;"

echo Standard group by pressure query
time_fn "SELECT
    CASE
       WHEN trestbps<80 THEN 'hypotension'
       WHEN trestbps<120 THEN 'normal'
       WHEN trestbps<140 THEN 'prehypertension'
       WHEN trestbps<170 THEN 'hypertension stage 1'
       WHEN trestbps<180 THEN 'hypertension stage 2'
       ELSE 'high crisis (emergency)'
    END
```

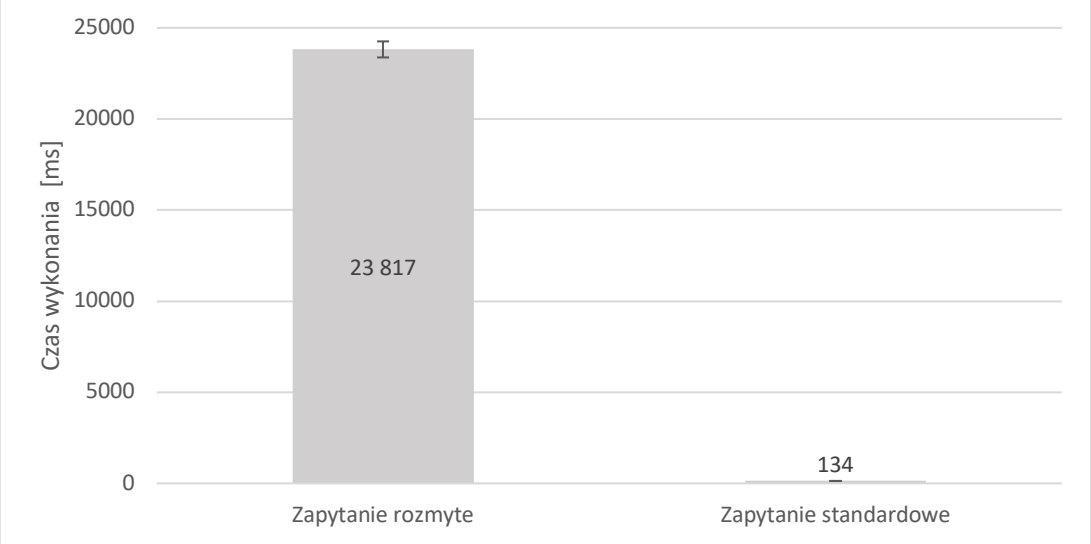
```
        AS pressure,
        COUNT("'*'") AS COUNT
    FROM heart_disease
    GROUP BY pressure;"
```

Listing 4.6 Zapytanie agregujące elementy tabeli na podstawie zakresu tętna

pressure ↑	count
	56320
high crisis (emergency)	27648
hypertension stage 1	232448
hypertension stage 2	152576
hypotension	13312
normal	262144
prehypertension	197632

Rys. 4.6 Wynik zapytania przyporządkowującego liczbę osób do zakresów ciśnienia tętniczego

Uzyskane czasy dostępu zaprezentowano na wykresie na Rys. 4.7. Czas wykonania zapytania grupującego jest ponad 170 razy dłuższy. Jest to spowodowane wspomnianą koniecznością odpytywania tablicy z typami i funkcjami lingwistycznymi i może zostać zoptymalizowane w przyszłości. Ważnym aspektem jest również prostota rozwiązania rozmytego, które jest bardziej zwięzłe i rozszerzalne (nie trzeba modyfikować nazw i wartości w wielu miejscach).



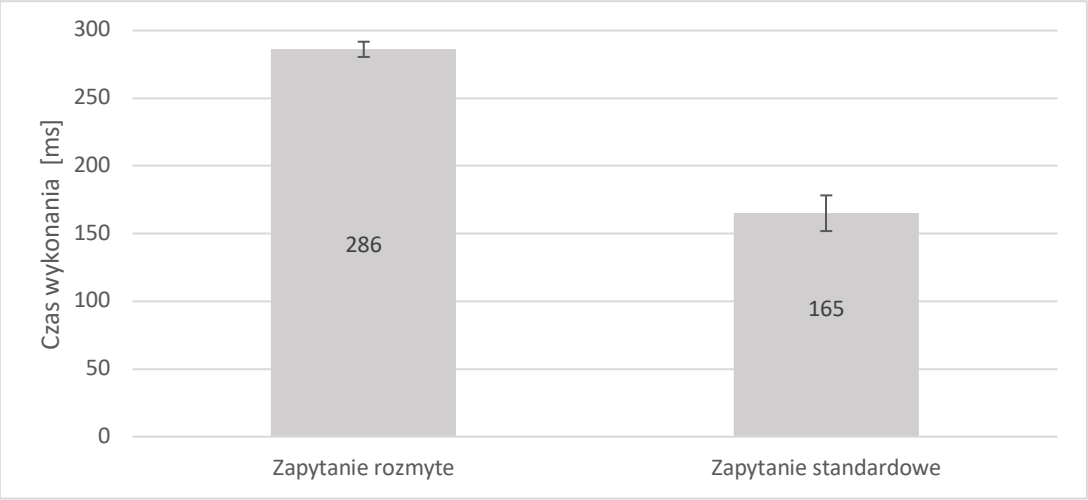
Rys. 4.7 Wykres czasu wykonania zapytań grupujących

Ostatni test wydajności operatorów to również zapytanie agregujące, jednak wykorzystujące pojedynczą funkcję rozmytą. Identycznie jak w poprzednich przypadkach, zdefiniowano jej nierozmyty, minimalistyczny odpowiednik. Funkcje testowe opisane są w listingu 4.7.

```
echo Fuzzy group by normal pressure query
time_fn "SELECT
    thalach~=('blood_pressure'|>'normal') AS pressure,
    COUNT("'*'") AS COUNT
    FROM heart_disease
    GROUP BY pressure;"
echo Standard group by normal pressure query
time_fn "SELECT
    CASE
        WHEN (thalach IS NULL) THEN NULL
        WHEN (thalach<75) THEN 0.0
        WHEN (thalach<80) THEN (thalach-75.0)/5.0
        WHEN (thalach<120) THEN 1.0
        WHEN (thalach<130) THEN (thalach-130.0)/-10.0
        ELSE 0.0
    END
     AS pressure,
     COUNT(*) AS count
    FROM heart_disease
    GROUP BY pressure;"
```

Listing 4.7 Zapytanie agregujące osoby na podstawie normalnego ciśnienia

Wynik zapytania znajduje się na Rys. 4.8. Optymalne zapytanie standardowe ponownie okazuje się szybsze od zapytania z elementem rozmytym. Z przeprowadzonych testów wynika jednoznacznie, że rozszerzenie wykonujące zapytania rozmyte wymaga optymalizacji. Obecnie obliczenia na typach rozmytych wykonywane są w naiwny sposób, który jest dostatecznie dobry, gdy w zapytaniu wykonywana jest jedna funkcja, jednak typy rozmyte stanowią warstwę abstrakcji nad złożonym pojęciem, którą cechuje uproszczenie zadania dla programisty lub uogólnienie pojęcia, co odbywa się kosztem spadku wydajności.



Rys. 4.8 Wykres czasu wykonania zapytania agregującego na podstwie jednej funkcji rozmytej

Poniżej, w tabeli 4.2 podsumowane zostało przeprowadzone badanie wydajności operatorów i czas wykonania względem zoptymalizowanej, minimalnej operacji w języku SQL bez rozszerzenia Fuzzy.

Zapytanie	Względny czas wykonania zapytania rozmytego do zapytania ostrego
1. Proste	2.12x
2. Agregujące nazwy typu	177.6x
3. Agregujące stopień przynależności do funkcji	1.73x

Tabela 4.2 Podsumowanie wyników badania wydajności operatorów rozmytych

4.4. Metodyka badań pomiaru wydajności aplikacji Fuzzybase

Pomiar wydajności został przeprowadzony na aplikacji z włączonym debugowaniem, aby móc obserwować czas wykonania poszczególnych etapów wykonania. Wydzielono trzy etapy: wykonanie polecenia przez serwer SQL oraz jego odbiór (etap SQL), przesyłanie odpowiedzi z procesu głównego do procesu przeglądarki (etap IPC) oraz wyświetlenie danych (etap VDOM oraz DOM). Z uwagi na sposób działania silnika języka JavaScript, czas wykonania jednego etapu może wpływać na resztę etapów, tj. zwiększenie czasu wykonania jednego z etapów może wpłynąć na drugi. Oczywistym jest, że objętość danych zwracanych w wyniku zapytania wpływa na wydłużenie czasu wykonania wszystkich etapów. Badanie ma na celu pokazać zależność czasu wykonania zapytań od liczby wyświetlanych wierszy przy uwzględnieniu dwóch sposobów renderingu: DOM oraz Virtual DOM. W tym celu zostały przygotowane dwa komponenty, z których jeden (DOM) rysuje wszystkie elementy tabeli na raz, a drugi (Virtual DOM) rysuje jedynie te, które są widoczne na ekranie. Mierzony czas, złożony się z wyżej wymienionych etapów, można uznać za składowe czasu, przez który użytkownik oczekuje na wynik. Na koniec zestawiono ze sobą wszystkie etapy obsługi polecenia oraz łączny czas wykonania w zależności od sposobu rysowania wyników. Zestawienie powinno pokazać, który z etapów należy zoptymalizować, aby umożliwić działanie aplikacji z każdą ilością danych.

4.5. Zbiór danych testowych do pomiaru wydajności aplikacji Fuzzybase

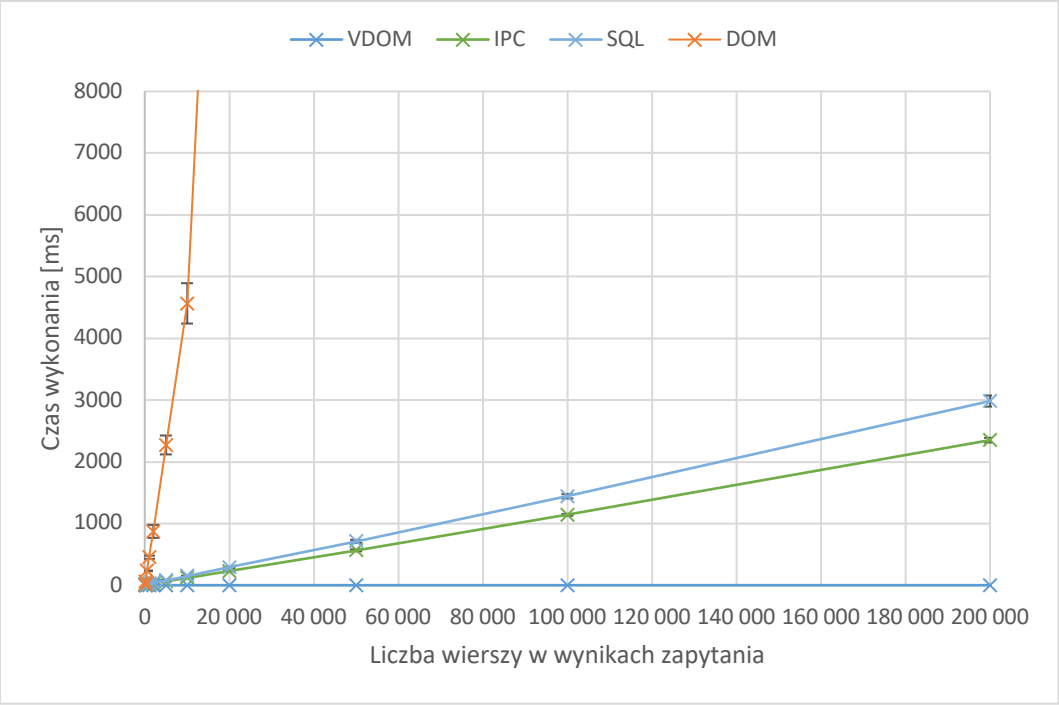
Do pomiarów wydajności wykorzystano ten sam zbiór danych medycznych, który zastosowano do badań wydajności operatorów rozmytych. Dla zapewnienia wystarczającej liczby danych do testów, zbiór został zduplikowany tak samo, jak zostało to opisane w rozdziale 4.2.

4.6. Wyniki pomiaru wydajności aplikacji Fuzzybase

Wykres liniowy utworzony na podstawie danych zebranych z aplikacji znajduje się na Rys. 4.9. Z uwagi na szeroki zakres danych pomiarowych, zaprezentowane zostały uśrednione wyniki z 20 prób (Tabela 4.3). W przypadku pomiarów wydajności DOM, aplikacja przestała zwracać wyniki dla zapytań zwracających powyżej 20 tys. wierszy. Dla metody Virtual DOM czas rysowania był stały i wynosił mniej niż 1 milisekunda.

Liczba wierszy	**100**	**200**	**500**	**1 tys.**	**2 tys.**	**5 tys.**	**10 tys.**	**20 tys.**	**50 tys.**	**100 tys.**	**200 tys.**
VDOM	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1
DOM	4	56	233	453	873	2272	4566	18265	-	-	-
IPC	11	17	14	23	31	64	118	230	565	1140	2350
SQL	13	19	17	26	38	81	149	289	712	1440	2983
Łącznie DOM	29	94	266	504	944	2418	4834	18785	-	-	-
Łącznie VDOM	26	38	34	51	72	147	269	521	1278	2582	5336

Tabela 4.3 Podsumowanie danych pomiarowych dot. wydajności aplikacji Fuzzybase



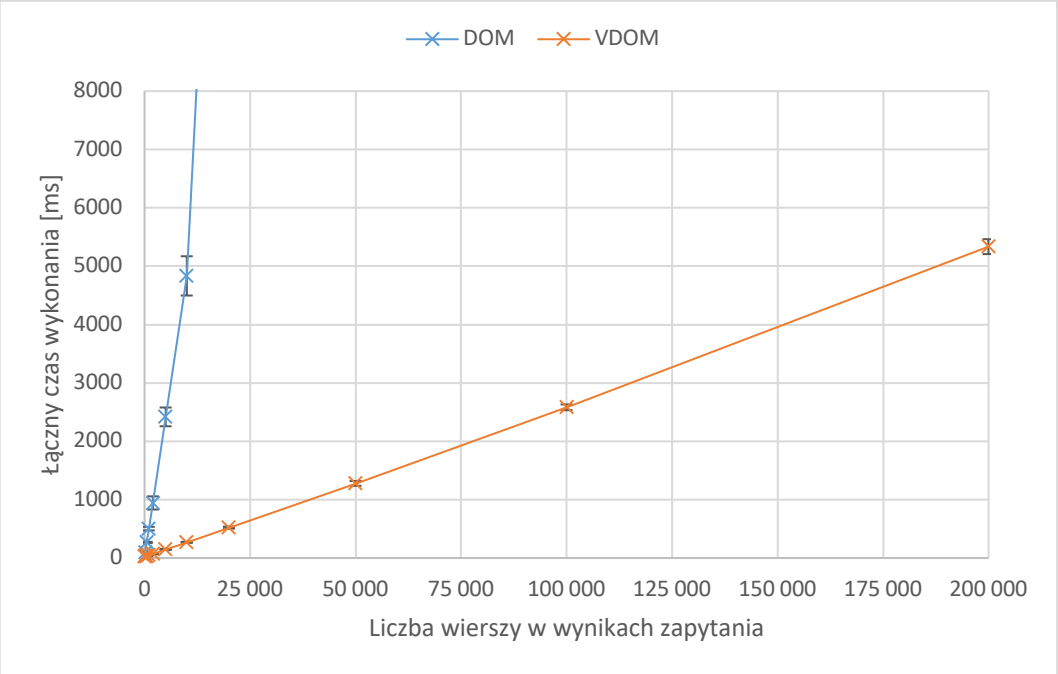
Rys. 4.9 Wykres czasu przetwarzania danych w zależności od liczby wierszy w odpowiedzi z serwera

Wyniki pokazują liniową zależność między czasem wykonania poszczególnych etapów względem liczby zwracanych wierszy. W przypadku czasu wykonania etapu rysowania tabeli z rezultatem zapytania z wykorzystaniem Virtual DOM osiągnięte wyniki są zadowalające, a głównym czynnikiem wpływającym na utrzymanie stałego czasu wykonania etapu jest renderowanie jedynie tych elementów, które są widoczne na ekranie. Aby osiągnąć taki wynik, posłużono się biblioteką `ag-grid`, która przyjmuje dane wejściowe w postaci tablicy oraz nagłówków. Komponent skonfigurowano tak, aby obliczył szerokość kolumn dopasowując je do szerokości wyświetlanych danych. Biblioteka wykorzystuje przekazany rozmiar komórek i tworzy jedynie tyle elementów drzewa DOM, ile to konieczne. Zawartość oraz lokalizacja komórek są podmieniane podczas przewijania listy, dzięki czemu można zaoszczędzić pamięć oraz czas procesora, który mógłby niepotrzebnie zostać poświęcony tworzeniu elementów drzewa, które nigdy nie zostaną wyświetlone. Użyty komponent do wyświetlania wyników DOM jest niewydajny, a czas rysowania rośnie wykładniczo wraz ze wzrostem liczby zwracanych wierszy.

Aplikacja nie ma wpływu na wynik etapu SQL, składa się na niego faktyczny czas wykonania zapytania na serwerze oraz czas przyjmowania odpowiedzi z serwera (wliczając w to konwersję danych do notacji JSON). Jest to czas, przez którego

fragment aplikacja przestaje być responsywna, co wpływa negatywnie na wrażenia z użytkowania aplikacji. Negatywny efekt można zniwelować, przenosząc moduł obsługi bazy danych do osobnego, dedykowanego wątku roboczego, co zostało opisane w rozdziale 4.8.4.

Ostatnią składową czasu jest etap IPC. Z perspektywy użytkownika aplikacja przestaje być w tym czasie responsywna i jest to czas całkowicie zmarnowany. Przy użyciu narzędzi w technologiach wspierających natywnie wielowątkowość (np. Java), aplikacja nie miałaby najmniejszego problemu w płynnym działaniu, nawet jeśli zaszłaby konieczność wykonania obliczeń, natomiast problem blokującej komunikacji międzyprocesowej w ogóle by nie istniał, gdyż inne rozwiązania oferują zazwyczaj pamięć współdzieloną. W przypadku języka JavaScript można wykorzystać wspomniane wątki robocze, które działają niezależnie od interfejsu użytkownika. Nie posiadają one jednak pełnoprawnej pamięci współdzielonej, przez co komunikacja między wątkiem roboczym, a wątkiem przeglądarki wiązałaby się z koniecznością serializacji danych lub ich konwersji do formatu pozwalającego na szybkie kopiowanie lub też przenoszenie między wątkami.



Rys. 4.10 Wykres łącznego czasu przetwarzania danych w zależności od liczby wierszy w odpowiedzi z serwera

4.7. Wpływ wykorzystania zapytań rozmytych na ilość zwracanych danych

Zapytania rozmyte, choć mniej wydajne od zapytań ostrych, pozwalają na uzyskanie szerszego zbioru danych dzięki zmianie warunków brzegowych na takie, które dopuszczają część wartości spoza sztywno ustalonego przedziału. Zapytania typu Fuzzy pozwalają na określenie poziomu przynależności wyników, dzięki czemu nie traci się na istotności wyników. W przypadku uczenia maszynowego połączonego z bazą danych, etap preselekcji z wykorzystaniem zapytania rozmytego pozwala na zyskanie dodatkowych przypadków uczących. Wykorzystując bazę danych z danymi medycznymi dotyczącymi chorób serca, zbadano zapytania ostre oraz rozmyte pod kątem trafności dostarczanych wyników.
Zbudowano zapytania ostre (Listing 4.8) oraz rozmyte (Listing 4.9) wyszukujące pacjentów z:

1. wysokim tętnem
2. podniesionym ciśnieniem po wysiłku
3. wysokim poziomem cholesterolu

Na listingach pokazano ostatnie, końcowe zapytanie, składające się ze wszystkich elementów. Eksperyment przeprowadzano wykorzystując:

- w zapytaniu pierwszym – tętno (1)
- w zapytaniu drugim – tętno (1) i ciśnienie (2)
- w zapytaniu trzecim – tętno (1), ciśnienie (2) i poziom cholesterolu (3).

```
SELECT * FROM heart_disease
    WHERE
        thalach>130 AND -- 1 tętno
        trestbps>140 AND -- 2 ciśnienie krwi
        chol>200; -- 3 poziom cholesterolu
```

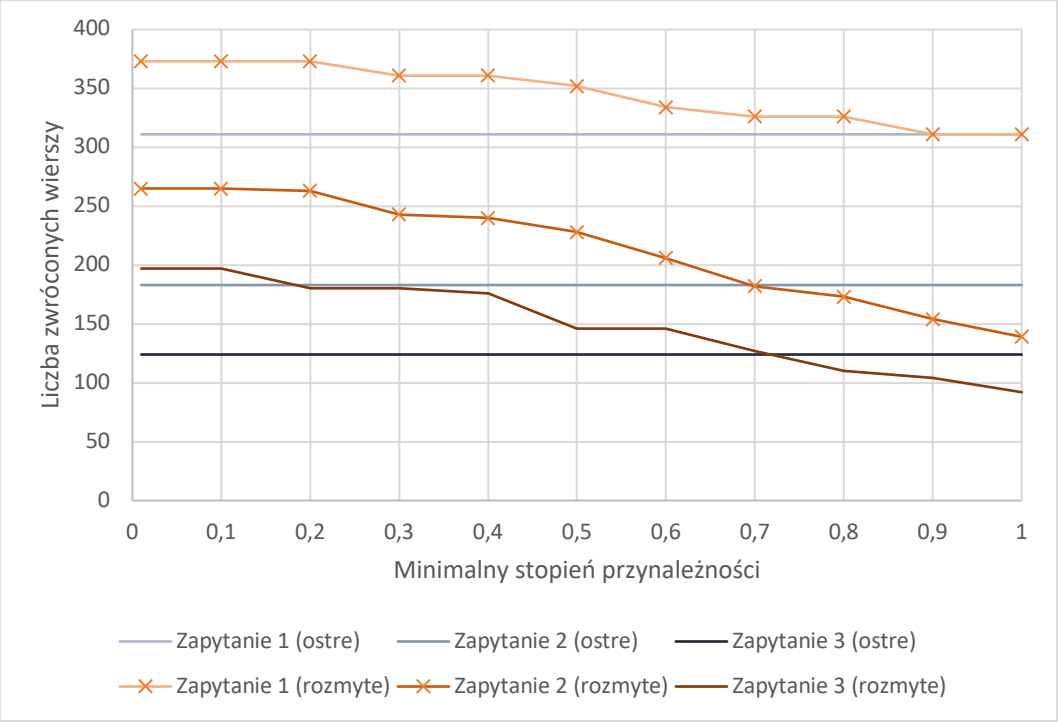
Listing 4.8 Zapytanie ostre wyszukujące pacjentów

```
SELECT * FROM heart_disease WHERE (
  ( -- 1 tętno
    (trestbps~=('blood_pressure'|>'hypertension stage 1')) |||
    (trestbps~=('blood_pressure'|>'hypertension stage 2')) |||
```

```
    (trestbps~=('blood_pressure'|>'high crisis (emergency)'))
  ) &&& ( -- 2 ciśnienie krwi
    (thalach~=('heart_rate'|>'very high')) |||
    (thalach~=('heart_rate'|>'high'))
  ) &&& ( 3 – poziom cholesterolu
    (chol~=('total_cholesterol'|>'borderline high')) |||
    (chol~=('total_cholesterol'|>'high'))
  )
) >=.01;
```

Listing 4.9 Zapytanie rozmyte wyszukujące pacjentów

Liczba zwróconych wierszy z tabeli zawierającej łącznie 920 wierszy, w zależności od wykorzystanego typu zapytania oraz stopnia przynależności jest pokazana na wykresie na Rys. 4.11. Ze względu na niezazębiające się przedziały wartości zmiennych lingwistycznych w przypadku ciśnienia krwi, zapytanie rozmyte zwróciło mniej wyników dla stopnia przynależności równego 1. Widać jednak, że zapytania z elementami rozmytymi zwracają więcej wyników przy ustaleniu progu stopnia przynależności na poziomie około 50%.



Rys. 4.11 Wykres zależności liczby wierszy od rodzaju zapytania i stopnia przynależności

W zapytaniu może zostać uwzględniony stopień przynależności wiersza do ustalonych kryteriów, który pozwala odpowiednio wartościować jego wyniki. Opisany eksperyment ilustruje przewagę zapytań rozmytych nad klasycznymi. Należy jednak wziąć pod uwagę, że jakość danych zależy od użytych kryteriów zapytania – niezależnie od tego, czy wykorzystuje ono elementy rozmyte czy jedynie warunki ostre. W przypadku niepokrywających się przedziałów można uzyskać mniej wyników niż w przypadku zapytania ostrego, co nie oznacza, że wyniki te są gorszej jakości, jeżeli taki był zamysł osoby projektującej zapytanie.

4.8. Problemy napotkane w pracy

Podczas realizacji projektu programistycznego napotkano problemy, które spowolniły pracę. Postanowiono udokumentować te najważniejsze wraz z rozwiązaniami. Jedno z najistotniejszych zagadnień to etap, który nastąpił po refaktoryzacji rozszerzenia. Polegał on na opracowaniu niezawodnego sposobu budowania oraz poprawieniu błędów kompilacji rozszerzenia. Mechanizm budowania oparto o dodatek PGXS dostarczany jako element narzędzi deweloperskich bazy PostgreSQL. Nie są one jednak dostępne dla systemu Windows, co zostało opisane w podrozdziale 4.8.1. Mimo braku błędów kompilacji, zrefaktoryzowane rozszerzenie nie działało w bazie danych z powodu brakujących metadanych i makr wprowadzonych w pierwszej wersji konwencji wywołań (podrozdział 4.8.2).
Dalsze trudności dotyczą tworzenia aplikacji w środowisku Electron. Konieczne było zbudowanie lub znalezienie komponentu pozwalającego na pracę z kodem SQL. Komponent powinien posiadać wsparcie dla podkreślania składni oraz podpowiedzi kontekstowych. Samodzielne stworzenie takiego komponentu wymagałoby poświęcenia znacznej ilości czasu na rozwój osobnego projektu, dlatego zdecydowano się na wykorzystanie edytora CodeMirror, który okazał się ograniczony i niewystarczający (podrozdział 4.8.3). Na koniec napotkano problemy z architekturą procesów w aplikacji, czego rozwiązanie zostało zaproponowane w podrozdziale 4.8.4.

4.8.1. Dostosowanie projektu rozszerzenia do obsługi wielu systemów operacyjnych

W stosunku do kosmetycznych różnic interfejsu obsługiwanych przez narzędzie Electron, o wiele większym wyzwaniem było napisanie rozszerzenia, które można zbudować i zainstalować na wszystkich platformach. Aby uprościć zadanie wielokrotnej kompilacji natywnej biblioteki systemowej, zrezygnowano z wsparcia dla systemów 32-bitowych i ograniczono się do architektury 64-bitowej. Kod samego rozszerzenia jest uniwersalny, lecz nie został zbudowany pod kątem architektury 32-bitowej. Z racji tego, że cały projekt systemu dydaktycznego był tworzony na platformie macOS, w pierwszej kolejności biblioteka była kompilowana na tę platformę. Wykorzystano narzędzie do tworzenia rozszerzeń PGXS, które uprościło cały proces, umożliwiając dodanie testów, instalację, uruchamianie i jednoznaczną weryfikację działania całego rozszerzenia. Po pomyślnych testach i rozwiązaniu problemów opisanych w rozdziale 4.8, postanowiono uruchomić rozszerzenie na platformie Linux. Mimo kompilacji rozszerzenia do formatu biblioteki współdzielonej `.so`, niemal oczywistym było, że biblioteka skompilowana na systemie macOS, nie będzie działać na systemie Linux. Potwierdziła to próba bezpośredniego zainstalowania tak skompilowanej wtyczki. Skompilowano następnie wtyczkę na systemie Linux. Poza instalacją paczki narzędzi deweloperskich PostgreSQL wskazanej automatycznie przez skrypt `Makefile`, kompilacja nie wymagała dodatkowych kroków. Taka przenośność rozwiązania jest możliwa dzięki korzeniom obu systemów operacyjnych, u których podstawie leży Unix. Po pomyślnym uruchomieniu wtyczki na platformie Linux, przyszła kolej na system Microsoftu. Rozszerzenie PGXS nie jest oficjalnie wspierane na platformie Windows, jednak jest dostępne w paczce z systemem bazodanowym PostgreSQL. Zdecydowano się na instalację narzędzi MinGW, których zastosowanie nie przyniosło oczekiwanych rezultatów, skutkując jedynie długą listą błędów. Po znalezieniu artykułu opisującego proces tworzenia rozszerzeń PostgreSQL pod kontrolą systemu Windows [30], zdecydowano się na wykonanie kroków w zawartej weń instrukcji. Okazało się, że w zainstalowanym wcześniej środowisku programistycznym Visual Studio Enterprise 2016, nie można utworzyć nowego, pustego projektu. Spowodowało to duże opóźnienie w budowaniu rozszerzenia, w czasie którego podjęto próby naprawienia środowiska, aż w końcu zdecydowana się je odinstalować i skorzystać z aktualnej

wersji Visual Studio Community, która umożliwiła utworzenie projektu. Projekt został skonfigurowany według instrukcji z artykułu, jednak nie udało się go skompilować bez ręcznego dodania brakujących zależności do projektu.
Po pomyślnej kompilacji i testach na wszystkich trzech platformach, uznano, że najpewniejszym sposobem dystrybucji jest dostarczenie zbudowanych plików z natywnymi rozszerzeniami (pliki `*.so` i `*.dll`) i wymaganych przez PostgreSQL plików *`.control` i *`.sql`.

4.8.2. Kompilacja rozszerzenia dla elementów rozmytych w systemie PostgreSQL

Podczas pracy nad kodem rozszerzenia w języku C napotkano na problemy związane z dużymi zmianami interfejsu programistycznego wykorzystywanego w systemie PostgreSQL. Po refaktoryzacji pierwotnego rozwiązania, okazało się, że współcześnie PostgreSQL akceptuje kod napisany zgodnie z pierwszą konwencją wywołań (*V1 calling convention*), a oryginalnie kod został opracowany zgodnie z konwencją w wersji 0. Konwencja *V0* określała wywołanie funkcji w standardowy sposób, w jaki tworzone są biblioteki w języku C. Nowszy standard wymaga natomiast zdefiniowania dodatkowych metadanych i korzystania z makr, które pozwalają na przekazanie danych do funkcji w postaci rekordu zawierającego bardziej złożone informacje o zwracanych wartościach. Jednocześnie nie jest możliwe bezpośrednie przekazywanie wartości przez wskaźniki oraz proste zwracanie wartości, a każda wartość musi być uprzednio zdekodowana. Przykład (Listing 4.10) obrazuje różnicę między starą a nową konwencją.

```
// Konwencja wywołań 0
bool defuzzy_fext_r(trapezoidal_function_extended *fext, float8 data)
{
    bool result = calc(fext, data); // Obliczenia na podstawie warto-
ści
    return result; // Zwrócenie wyniku
}
// Konwencja wywołań 1
PG_FUNCTION_INFO_V1(pg_defuzzy_fext_r);
Datum pg_defuzzy_fext_r(PG_FUNCTION_ARGS) {
    // Wyłuskanie danych wejściowych
    trapezoidal_function_extended* fext = (trapezoidal_func-
tion_extended*) PG_GETARG_POINTER(0);
```

```
    float8 data = (float8) PG_GETARG_FLOAT8(1);

    bool result = calc(fext, data); //Obliczenia
    PG_RETURN_BOOL(result); // Zwrócenie wyniku
}
```

Listing 4.10 Porównanie konwencji wywołań PostgreSQL funkcji w języku C

Jak można łatwo zauważyć, narzut kodu jest duży, dodatkowo nie ma możliwości ponownego wykorzystania funkcji napisanej w konwencji pierwszej przez inną funkcję znajdującą się w bibliotece. Rozszerzenie zawierające ponad 60 funkcji wymagało zatem kolejnej refaktoryzacji, tym razem w celu dostosowania do wymagań narzuconych przez system PostgreSQL. Aby uniknąć zbędnej duplikacji kodu i przyspieszyć prace nad projektem, postanowiono stworzyć makra, które pozwoliły obłożyć istniejące funkcje dodatkową, wymaganą warstwą abstrakcji. Po zbudowaniu własnych makr, rozwiązaniem problemu było dodanie kodu do każdej funkcji w postaci jak niżej.

```
PG_FUNC_2(defuzzy_fext_r, bool, BOOL, trapezoidal_function_extended*,
POINTER, float8, FLOAT8);
```

Listing 4.11 Makro upraszczające konwencję wywołań PostgreSQL

Argumentami są kolejno: nazwa funkcji, która ma zostać wywołana, oryginalny typ zwracany, typ zwracany w konwencji PostgreSQL oraz typy oryginalne oraz postgresowe argumentów funkcji. Niestety, nie wszystkie argumenty mogą być przekazywane jako wskaźniki, czego przykładem okazał się typ `float8` (zmiennoprzecinkowa liczba podwójnej precyzji). Dla tego typu makro `PG_GETARG_POINTER` zwracało nieprawidłową wartość, co kończyło się wymuszeniem restartu serwera bazy danych. Najprawdopodobniej jest to spowodowane optymalizacją dla procesorów 64-bitowych, na których można przekazać taką wartość bezpośrednio. Nie jest to jednak uwzględnione w dokumentacji systemu bazodanowego. Z powodu niemożności zastąpienia wszystkich zmiennych wejściowych oraz wartości zwracanej wskaźnikami, makra muszą przyjmować zarówno typy C, jak i postgresowe. Dodatkowym utrudnieniem jest brak możliwości zwracania wskaźnika `NULL` poprzez makro `PG_RETURN_POINTER`. W związku z tym utworzono makra `PG_FUNC_*_POINTER`, które obsługują zwracanie wskaźników.

Ostatnim problemem dotyczącym kompilacji okazało się ograniczone wsparcie podczas budowania rozszerzeń do bazy danych. Podczas pracy z kodem nie było

dostępnej dokumentacji, z której można byłoby korzystać bezpośrednio w środowisku programistycznym, gdyż standardowa instalacja PostgreSQL nie dodaje odpowiednich ścieżek do zmiennych środowiskowych. W związku z tym ręczna kompilacja kodu biblioteki jest również znacznie utrudniona i zalecane jest posługiwanie się infrastrukturą PGXS [26]. Sam fakt, że istnieją narzędzia do tworzenia rozszerzeń dla PostgreSQL nie jest rozpowszechniony, wydłużył czas pracy nad rozszerzeniem. Ponadto, dokumentacja dotycząca konwencji wywołań nie zawiera informacji dotyczących sposobu działania makr, co pociąga za sobą konieczność przeglądania plików nagłówkowych w poszukiwaniu przyczyn odmiennego działania programu.

4.8.3. Rozwiązanie pozwalające na edycję kodu

Aplikacja Fuzzybase pozwala na tworzenie zapytań SQL z elementami rozmytymi, które są dodatkowo podpowiadane. Aby uzyskać tą funkcjonalność, początkowo zdecydowano się skorzystać z popularnego rozwiązania CodeMirror. Szybki rozwój technologii JavaScript, który umożliwił tworzenie aplikacji w oparciu platformę Electron, pociągnął za sobą wiele zmian związanych ze sposobem pracy, w tym zarządzaniem modułami. Biblioteka CodeMirror może zostać zawarta na stronie w klasyczny sposób, nie wspierając dobrodziejstw płynących z bezpośredniego załączania zewnętrznych modułów do plików z kodem JavaScript. Biblioteka okazała się trudna w implementacji, nie posiada wsparcia dla języka TypeScript, a także jest niekompatybilna z współczesnym językiem JavaScript, nie pozwalając na użycie poprzez polecenie `import` lub `require` oraz umieszczając zależności w obiekcie `global`, co od czasu wzrostu popularności Node.js jako uniwersalnej platformy do tworzenia aplikacji, jest piętnowane jako zła praktyka programistyczna. Autorzy biblioteki CodeMirror pracują nad wydaniem nowej, przepisanej od zera wersji, jednak nie można jej jeszcze użyć w środowisku produkcyjnym. Po sprawdzeniu innych otwartoźródłowych rozwiązań dostępnych na rynku, bardzo dobrym rozwiązaniem okazał się edytor kodu Monaco, opisany w rozdziale 3.2.3.

4.8.4. Płynność działania aplikacji pod obciążeniem

Podczas przeprowadzania wstępnych testów wydajnościowych, okazało się, że serializacja dużej liczby danych (ponad milion rekordów) blokuje wątek główny, powodując zawieszenie aplikacji do czasu obsługi akcji. Powodem zawieszenia, mimo z pozoru osobnych procesów, jest zależność procesu Chromium od procesu

Node.js, który zarządza wszystkimi natywnymi zdarzeniami i jest wymagany np. do przerysowania zawartości okna przeglądarki. Rozwiązaniem tego problemu jest wykorzystanie dedykowanych wątków roboczych, zarówno po stronie procesu głównego jak i procesu przeglądarki. Wątki robocze mogą być obciążone bez ograniczeń, nie powodując przy tym dużego spadku wydajności aplikacji. Opisane rozwiązanie może zostać wdrożone w przyszłości.

5. Podsumowanie

Na przestrzeni ostatnich 12 miesięcy, stopniowo powstawał system dydaktyczny, składający się z rozszerzenia do systemu zarządzania bazą danych PostgreSQL oraz samodzielna aplikacja, będąca w stanie wykonywać zapytania, asystować w procesie ich tworzenia oraz wyświetlać ich wyniki. Aplikacja oraz rozszerzenie spełniają założenia pracy, pozwalając również na dynamiczne dodawanie własnych funkcji rozmytych i wykorzystanie ich w zapytaniach, bez konieczności ponownej kompilacji rozszerzenia. Rozwiązanie to zostało zainspirowane narzędziem SQLf_j opisanym w rozdziale 2.1.

Jak wykazały badania porównawcze wydajności operatorów, wskazana jest optymalizacja rozwiązania, co jest możliwe poprzez wykonywanie pewnych fragmentów funkcji pomocniczych tylko raz na zapytanie, a także utworzenie natywnych typów i funkcji pozwalających na szybką konwersję wartości liczbowej na etykietę funkcji. Badanie wydajności aplikacji wykazało, że czas obsługi zapytań rośnie liniowo wraz ze wzrostem liczby wierszy, a obecnie wydajność jest na zadowalającym poziomie. Aplikacja może zostać zoptymalizowana pod kątem przetwarzania dużych ilości danych, dzięki zastosowaniu wątków roboczych. Architektura utworzonej aplikacji Fuzzybase jest modułowa, dzięki czemu rozwiązanie będzie można poszerzyć o funkcjonalności, np. opisaną w rozdziale 3.2.3 pełną funkcjonalność podpowiedzi kontekstowych. Rozwiązanie zostanie również udostępnione na potrzeby laboratorium przedmiotu Bazy Danych, lecz ma potencjał wykorzystania nie tylko w środowisku akademickim. Cały projekt był rozwijany w repozytorium Git przy użyciu platformy GitHub, dlatego może ona zostać upubliczniony jako oprogramowanie open source.

Mimo wad, takich jak problemy z wydajnością podczas operacji na dużych zbiorach danych, tytułowy cel pracy został zrealizowany. Projekt dydaktycznego systemu wspomagającego projektowanie zapytań SQL zawierających elementy rozmyte został zrealizowany i może zostać wykorzystany oraz rozwijany. Badania,

mające na celu sprawdzenie wydajności odświeżonego rozszerzenia oraz aplikacji stworzonej przy użyciu współczesnego stosu technologicznego (Electron, React, Node.js), wykazały, że choć rozwiązanie nie sprawdza się obecnie przy przeszukiwaniu dużych ilości danych, to ze względu na zastosowaną architekturę może zostać ono usprawnione w niedalekiej przyszłości, zanim jeszcze zostanie ono upublicznione.

Bibliografia

[1] B. Małysiak, Metody aproksymacyjnego wyszukiwania obiektów w bazach danych, Gliwice, 2003.

[2] B. Dziedzic, Występowanie elementów rozmytych w prostych i złożonych pytaniach SQL, Gliwice, 2005.

[3] P. Kalinowski, „SQLf_j - Fuzzy querying system in Java," 2006. [Online]. Available: http://calypso.cs.put.poznan.pl/projects/sqlf_j/. [Data uzyskania dostępu: 20 10 2018].

[4] P. Kalinowski, „SQLf - rozmyte zapytania do baz danych," 2006. [Online]. Available: http://calypso.cs.put.poznan.pl/projects/sqlf_j2006/tutorial/. [Data uzyskania dostępu: 20 10 2018].

[5] J. G. Gómez, „FSQL - A Fuzzy Query Language," 2004. [Online]. Available: http://www.lcc.uma.es/~ppgg/FSQL.html. [Data uzyskania dostępu: 20 10 2018].

[6] J. M. Medina, „GEFRED: A Generalized Model of Fuzzy Relational Databases," 1994.

[7] The PostgreSQL Global Development Group, „PostgreSQL 11 Documentation: SELECT," 2018. [Online]. Available: https://www.postgresql.org/docs/11/static/sql-select.html. [Data uzyskania dostępu: 21 10 2018].

[8] P. Kudłacik, „Fuzzy SQL queries in standard SQL database," *Journal of Medical Informatics & Technologies,* tom 25, pp. 28-37, 2016.

[9] Z. Peng, „A Study on Database Fuzzy Query Method in SQL," w *International Conference on Advances in Engineering*, Procedia Engineering, 2011, p. 340–344.

[10] J. Podstawny i B. Małysiak, „Realizacja grupowania i agregacji danych w module translacji zapytań rozmytych na zapytania klasyczne," w *Bazy Danych: Rozwój metod i technologii*, Gliwice, 2008.

[11] The PostgreSQL Global Development Group, „PostgreSQL 11 Documentation: F.15. fuzzystrmatch," [Online]. Available: https://www.postgresql.org/docs/11/static/fuzzystrmatch.html. [Data uzyskania dostępu: 21 10 2018].

[12] D. Hellmann, „Using Fuzzy Matching to Search by Sound with Python," 3 Marzec 2012. [Online]. Available: http://www.informit.com/articles/article.aspx?p=1848528. [Data uzyskania dostępu: 21 10 2018].

[13] Elastic, „Fuzzy Query | Elasticsearch Reference [6.4]," [Online]. Available: https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-fuzzy-query.html. [Data uzyskania dostępu: 21 10 2018].

[14] D. D. Chamberlin, „Early History of SQL," *IEEE Annals of the History of Computing,* tom 34, nr 4, pp. 78-82, 2012.

[15] M. Winand, „What's New in SQL:2016," 2017. [Online]. Available: https://modern-sql.com/blog/2017-06/whats-new-in-sql-2016. [Data uzyskania dostępu: 21 10 2018].

[16] The PostgreSQL Global Development Group, „PostgreSQL 11 Documentation: CREATE FUNCTION," 2018. [Online]. Available: https://www.postgresql.org/docs/11/static/sql-createfunction.html. [Data uzyskania dostępu: 21 10 2018].

[17] The PostgreSQL Global Development Group, „PostgreSQL 11 Documentation: User-defined Operators," 2018. [Online]. Available: https://www.postgresql.org/docs/11/static/xoper.html. [Data uzyskania dostępu: 21 10 2018].

[18] The PostgreSQL Global Development Group, „PostgreSQL 11 Documentation: C-Language Functions," 2018. [Online]. Available: https://www.postgresql.org/docs/11/static/xfunc-c.html. [Data uzyskania dostępu: 21 10 2018].

[19] S. Gude, M. Hafiz i A. Wirfs-Brock, „Javascript: The Used Parts," w *IEEE 38th Annual International Computers, Software and Applications Conference*, 2014.

[20] K. Simpson, You Don't Know JS: Async & Performance, O'Reilly Media, 2017.

[21] K. F. Tómasdóttir, M. Aniche i A. van Deursen, „The Adoption of JavaScript Linters in Practice: A Case Study on ESLint," w *IEEE/ACM 39th International Conference on Software Engineering*, 2018.

[22] S. Fenton, Pro TypeScript: Application-Scale JavaScript Development, Apress, 2017.

[23] JS Foundation, „Webpack Concepts," [Online]. Available: https://webpack.js.org/concepts/. [Data uzyskania dostępu: 1 11 2018].

[24] S. Kinney, w *Electron in Action*, Manning Publications, 2018, pp. 3-16.

[25] „Chromium for Developers," [Online]. Available: https://www.chromium.org/developers/content-module. [Data uzyskania dostępu: 09 10 2018].

[26] The PostgreSQL Global Development Group, „PostgreSQL 11 Documentation: Extension Building Infrastructure," 2018. [Online]. Available: https://www.postgresql.org/docs/11/static/extend-pgxs.html. [Data uzyskania dostępu: 21 10 2018].

[27] „electron-builder," Electron Community, [Online]. Available: https://electron.build/. [Data uzyskania dostępu: 30 10 2018].

[28] Slack, „Where work happens | Slack," [Online]. Available: https://slack.com/. [Data uzyskania dostępu: 30 10 2018].

[29] A. Janosi, W. Steinbrunn, M. Pfisterer i R. Detrano, „Heart Disease," 1988. [Online]. Available: https://data.world/uci/heart-disease.

[30] C. Ringer, „Compiling PostgreSQL extensions with Visual Studio on Windows," [Online]. Available: https://blog.2ndquadrant.com/compiling-postgresql-extensions-visual-studio-windows/. [Data uzyskania dostępu: 30 10 2018].

Spis skrótów i symboli

Zawartość dołączonej płyty

Na płycie DVD dołączonej do dokumentacji znajdują się następujące materiały:

- praca w formacie pdf,
- źródła programu,
- zbiory danych użyte w eksperymentach.

Spis rysunków

Rys. 2.1 Funkcja trapezowa 16
Rys. 3.1 Schemat działania procesów w platformie Electron 26
Rys. 3.2 Przepływ danych w architekturze Flux 27
Rys. 3.3 Cykl życia komponentu React 29
Rys. 3.4 Schemat zawartości paczki z projektem 30
Rys. 3.5 Struktura plików rozszerzenia Fuzzy dla PostgreSQL 32
Rys. 3.6 Uproszczony schemat architektury aplikacji Fuzzybase 34
Rys. 3.7 Ekran tworzenia zapytania i podpowiedzi kontekstowe 35
Rys. 3.8 Zrzut ekranu systemu dydaktycznego Fuzzybase w systemie Ubuntu 16.04 37
Rys. 3.9 Zrzut ekranu systemu dydaktycznego Fuzzybase w systemie Windows 10 38
Rys. 3.10 Zrzut ekranu systemu dydaktycznego Fuzzybase w systemie macOS . 38
Rys. 3.11 Okno komunikatora biurowego Slack [28] 40
Rys. 3.12 Okno aplikacji FuzzyQ prezentujące widok zapytania i wyników 41
Rys. 4.1 Wykres prezentujący wartości zmiennej lingwistycznej age 46
Rys. 4.2 Wykres prezentujący wartości zmiennej lingwistycznej `heart_rate` 47
Rys. 4.3 Wykres prezentujący wartości zmiennej lingwistycznej `blood_pressure` 47
Rys. 4.4 Wykres prezentujący wartości zmiennej lingwistycznej `cholesterol` 47
Rys. 4.5 Wykres czasu wykonania prostych zapytań 49
Rys. 4.6 Wynik zapytania przyporządkowującego liczbę osób do zakresów ciśnienia tętniczego 51
Rys. 4.7 Wykres czasu wykonania zapytań grupujących 51
Rys. 4.8 Wykres czasu wykonania zapytania agregującego na podstwie jednej funkcji rozmytej 53

Rys. 4.9 Wykres czasu przetwarzania danych w zależności od liczby wierszy w odpowiedzi z serwera 56
Rys. 4.10 Wykres łącznego czasu przetwarzania danych w zależności od liczby wierszy w odpowiedzi z serwera 57
Rys. 4.11 Wykres zależności liczby wierszy od rodzaju zapytania i stopnia przynależności 59

Spis tabel

Tabela 2.1 Operatory logiczne 15
Tabela 2.2 Wykaz operacji na trapezowych funkcjach przynależności 17
Tabela 4.1 Fragment zbioru danych heart_disease 45
Tabela 4.2 Podsumowanie wyników badania wydajności operatorów rozmytych53
Tabela 4.3 Podsumowanie danych pomiarowych dot. wydajności aplikacji Fuzzybase 55

Spis listingów

Listing 2.1 Przykładowe zapytanie SQL 10
Listing 2.2 Funkcja w języku SQL dodająca typy rozmyte do bazy 13
Listing 2.3 Operator w języku SQL 14
Listing 3.1 Przykład niepoprawnego wyrażenia w języku TypeScript 23
Listing 3.2 Przykładowy bezstanowy komponent funkcyjny, wyświetlający informacje o użytkowniku 28
Listing 3.3 Epic odpowiadający za wykonywanie zapytań do bazy danych 34
Listing 4.1 Fragment skryptu powłoki wykonujący zapytania i mierzący ich czas 43
Listing 4.2 Przykładowe wywołanie funkcji pomiarowej 44
Listing 4.3 Zapytanie języka SQL definiujące zmienną lingwistyczną age 46
Listing 4.4 Kod prostego zapytania testowego 48
Listing 4.5 Kod funkcji obliczająca stopień przynależności 49
Listing 4.6 Zapytanie agregujące elementy tabeli na podstawie zakresu tętna 51
Listing 4.7 Zapytanie agregujące osoby na podstawie normalnego ciśnienia 52
Listing 4.8 Zapytanie ostre wyszukujące pacjentów 58
Listing 4.9 Zapytanie rozmyte wyszukujące pacjentów 59
Listing 4.10 Porównanie konwencji wywołań PostgreSQL funkcji w języku C .. 63
Listing 4.11 Makro upraszczające konwencję wywołań PostgreSQL 63