Spis treści

[1. Wstęp 3](#_Toc528283880)

[1.1. Plan pracy 3](#_Toc528283881)

[2. Analiza tematu 5](#_Toc528283882)

[2.1. Istniejące rozwiązania 5](#_Toc528283883)

[2.2. Wymagania 6](#_Toc528283884)

[2.3. Język SQL 7](#_Toc528283885)

[2.3.1. Elementy języka 8](#_Toc528283886)

[2.3.2. Rodzaje instrukcji 9](#_Toc528283887)

[2.3.3. Funkcje 11](#_Toc528283888)

[2.3.4. Operatory 12](#_Toc528283889)

[2.4. Logika rozmyta 13](#_Toc528283890)

[3. Przedmiot pracy 16](#_Toc528283891)

[3.1. Wykorzystane technologie 16](#_Toc528283892)

[3.1.1. PostgreSQL 17](#_Toc528283893)

[3.1.2. JavaScript/ECMAScript 17](#_Toc528283894)

[3.1.3. TypeScript 19](#_Toc528283895)

[3.1.4. Webpack 21](#_Toc528283896)

[3.1.5. Electron 21](#_Toc528283897)

[3.1.6. Architektura Flux, React i Redux 23](#_Toc528283898)

[4. Badania i implementacja 26](#_Toc528283899)

[4.1. Implementacja rozszerzenia dla elementów rozmytych w systemie PostgreSQL 26](#_Toc528283900)

[4.2. Architektura aplikacji klienckiej 29](#_Toc528283901)

[4.3. Metodyka badań 30](#_Toc528283902)

[4.4. Zbiory danych 30](#_Toc528283903)

[4.5. Wyniki 30](#_Toc528283904)

[5. Podsumowanie 32](#_Toc528283905)

[Bibliografia i](#_Toc528283906)

[Spis skrótów i symboli iv](#_Toc528283907)

[Zawartość dołączonej płyty v](#_Toc528283908)

[Spis rysunków vi](#_Toc528283909)

[Spis tabel vii](#_Toc528283910)

# 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 dziesięciu rozdziałów. W rozdziale pierwszym opisany jest cel, motywacja i plan pracy dyplomowej. W rozdziale drugim są określone wymagania funkcjonalne systemu i usprawnienia względem FuzzyQ. Rozdział trzeci opisuje język SQL, na którym opiera się system dydaktyczny. W rozdziale czwartym omawiane jest zagadnienie logiki rozmytej i podstawowych operatorów w analogii do logiki boolowskiej. Rozdział piąty przybliża technologie wykorzystane do zbudowania systemu, uzasadniając ich wybór. W rozdziale szóstym 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. Rozdział ósmy dotyczy rozwiązań technicznych, takich jak budowa komponentów aplikacji, komunikacja między wątkami, czy dodawanie operatorów do dowolnej bazy relacyjnej. W rozdziale dziewiątym jest opisany interfejs użytkownika. W rozdziale dziesiątym zaprezentowane są badania porównujące szybkość prezentacji danych w środowisku przeglądarkowym jakim jest Electron, a także różnice w wykorzystaniu zasobów między FuzzyQ, a nowym systemem. Rozdział jedenasty zawiera podsumowanie prac nad systemem i możliwości jego rozwoju.

# 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.

## 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 ilości. 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].

## Wymagania

Podstawowym wymaganiem jest współczesny interfejs umożliwiający tworzenie zapytań SQL ze wsparciem dla elementów rozmytych. Przyj [2] [1]mując, 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. 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.

## 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.

### 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 operatory: 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.

W poniższym przykładzie obrazującym elementy języka SQL, 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

### 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.

### 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 niżej.

**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

### 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 niżej.

CREATE OPERATOR ~> (  
 PROCEDURE = *get\_fuzzy\_name*,  
 LEFTARG = FLOAT8,  
 RIGHTARG = VARCHAR(64)  
);

Listing 2.3 Operator w języku SQL

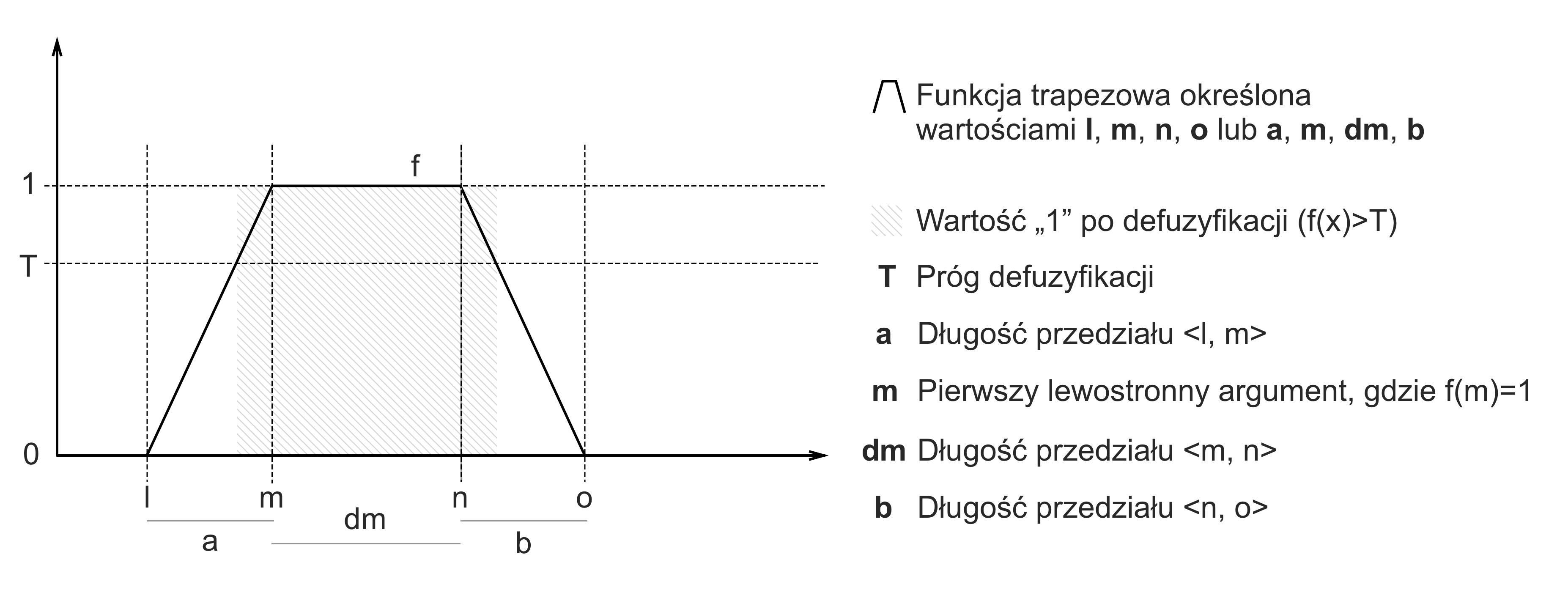
## Logika rozmyta

Twórcą teorii zbiorów rozmytych i logiki rozmytej jest Lotfi Zadeh, który zaproponował wykorzystanie funkcji przynależności przyjmującej wartości z przedziału <0, 1> [9]. Logika rozmyta to wielowartościowa odmiana logiki, która w odróżnieniu od klasycznej logiki boolowskiej pozwala na zapisanie wartości między prawdą, a fałszem. Wartość rozmyta może być następnie 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 opisane poniżej.

|  |  |
| --- | --- |
| 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

Logika rozmyta opisuje stopień przynależności wartości liczbowej do zbioru określonego odpowiednią funkcją. 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). Wewnętrznie wykorzystana jest 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 Rys. 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 funkcją 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.

|  |  |  |
| --- | --- | --- |
| 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. |
| 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

# 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 rozmyty i baza - Fuzzybase.

## 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].

### 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 4.1.

### 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 ECMAScript (skrótowo ES). 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. 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.

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.

### TypeScript

TypeScript jest rozszerzeniem języka JavaScript, dokładającym do niego możliwość typowania. 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.

**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.

### Webpack

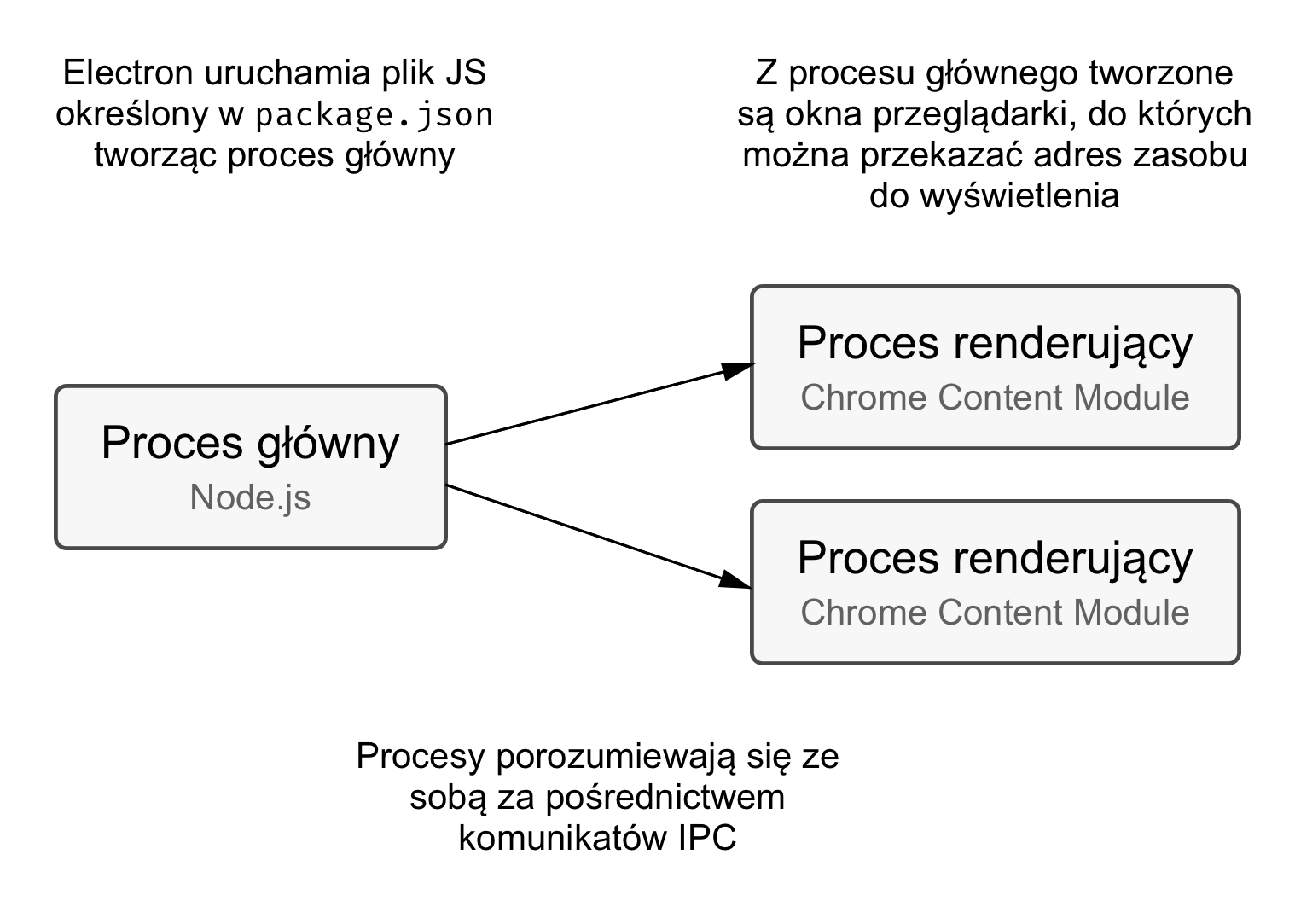
Webpack jest narzędziem służącym do automatyzacji złożonego procesu budowania aplikacji JavaScriptowych. Dzięki narzędziom ładującym (ang. *loaders*) umożliwia przetworzenie wszystkich plików projektu i wyłapanie zależności między nimi i wykorzystywanych zasobów. Pozwala to na udostępnienie w pakiecie 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.

### 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 [19].

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 [20]. 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. 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.

### 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 (*callbacks*) oraz akcji.

Aplikacje realizowane w architekturze Flux najczęściej posiadają wspólną strukturę przechowującą dane, nazywaną magazynem (z 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 wszystkich danych w aplikacji Flux. 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

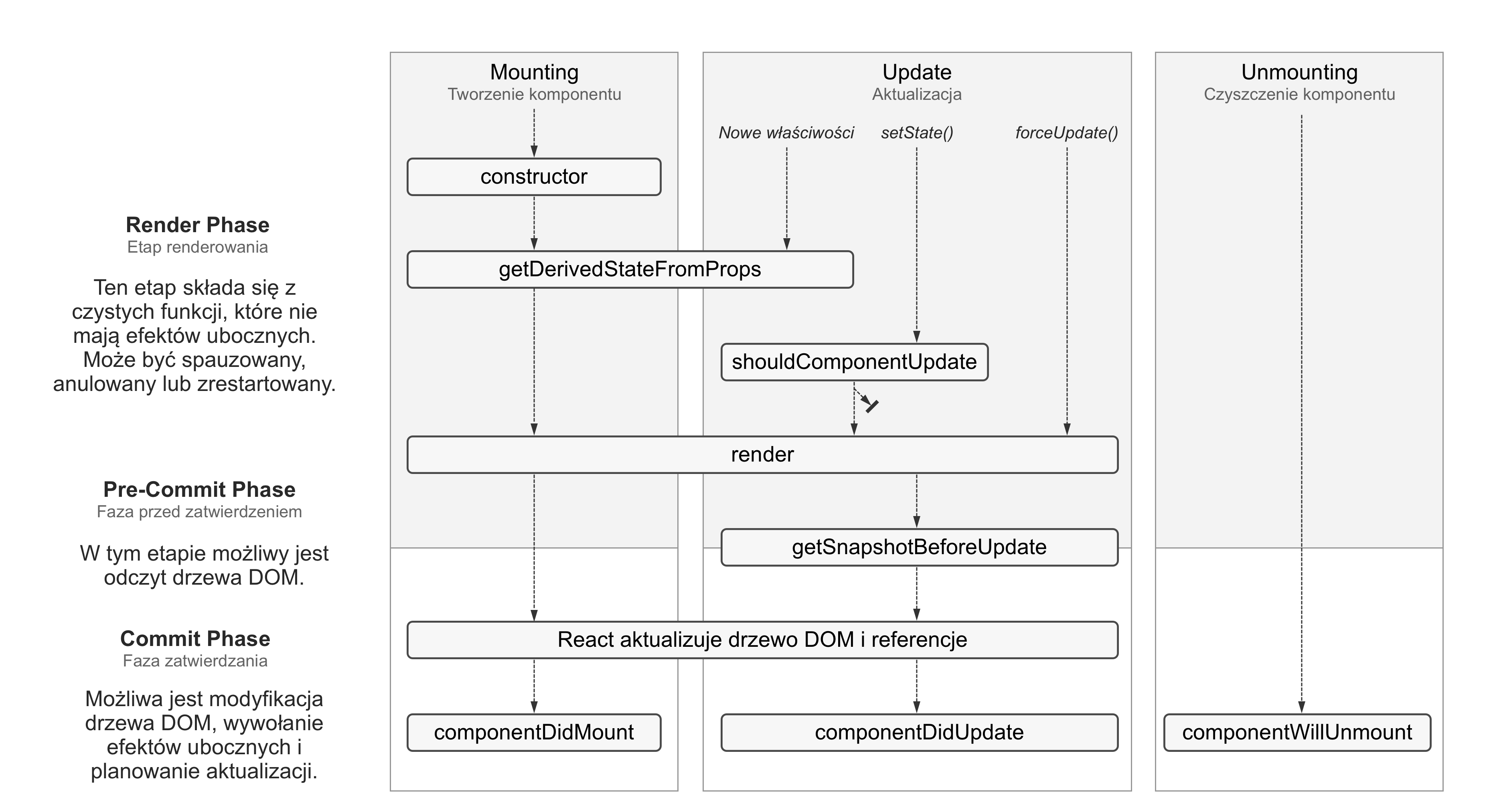
Biblioteka Redux to implementacja mechanizmów magazynu oraz dyspozytora. Programista ma za zadanie zdefiniowanie części logicznej magazynu, czyli tzw. *reducerów*. Akcje definiują jedynie to, co się wydarzyło, natomiast *reducer* definiuje, jak powinien zmienić się stan magazynu aplikacji po otrzymaniu 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*.

**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.

# Badania i implementacja

W niniejszym rozdziale są przedstawione rozwiązania problemów implementacyjnych, które pojawiły się podczas pracy nad systemem dydaktycznym Fuzzybase, a także badania wydajności stworzonego rozwiązania. Badania dotyczą wydajności operatorów zaimplementowanych w części bazodanowej (rozszerzenie PostgreSQL), a także części aplikacyjnej w zależności od ilości przetwarzanych danych oraz zastosowanego rozwiązania.

## Implementacja 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 typów i funkcji rozmytych. Dodatkowo, 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 [21]. Poza kompilacją, pakowaniem oraz instalacją rozszerzeń, PGXS umożliwia wykonywanie testów regresyjnych. Dzięki tej funkcjonalności, całe rozszerzenie zostało najpierw ręcznie sprawdzone pod kątem błędów, a następnie wyniki testów zostały zapisane w projekcie. Dzięki temu po dokonaniu zmian można mieć pewność, iż stare funkcje rozszerzenia działają tak samo jak poprzednio.

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. Poniższy przykład 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 wartości

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\_function\_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 4.1 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 4.2 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 jest 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 [21]. 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.

## 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.

Podczas przeprowadzania wstępnych testów wydajnościowych, okazało się, że serializacja dużej ilości 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.

## 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. Bazowa część skryptu powłoki została przedstawiona w listingu poniżej.

**#!/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 4.3 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 konkretnej 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 4.4 Przykładowe wywołanie funkcji pomiarowej

Postanowiono przetestować 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.

## Zbiór danych testowych

Za dane testowe, wykorzystane do zapełnienia bazy, przyjęto dostępne publicznie wyniki badań dotyczące chorób serca [22]. 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). Testowana tabela składa się z 15 kolumn opisujących pacjentów badanych pod kątem chorób serca. 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 – zalecany poziom cholesterolu określony jako przekroczony/nieprzekroczony
* 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 – ilość 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 – ilość 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 powyższych parametrów wyselekcjonowano trzy, do przetwarzania których można wykorzystać typy i funkcje rozmyte, są to: age, trestbps oraz restecg.

## Wyniki pomiaru wydajności operatorów rozmytych

Kod pierwszych, prostych zapytań jest przedstawiony w listingu 4.5. Zapytanie standardowe 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;"

Listing 4.5 Kod prostego zapytania testowego

Na wykresie poniżej (Rys. 4.1) zostały przedstawione uśrednione czasy wykonania wraz z odchyleniem standardowym. W zależności od tego, czy zwracane były kompletne wyniki, czy też tylko ilość wyfiltrowanych wierszy, czasy wykonani znacząco różniły się od siebie. W przypadku zwracania kompletnych wyników (wszystkie wiersze w całości), różnica w czasie wykonania nie wychodzi poza odchylenie standardowe. Natomiast, gdy wynikiem jest jedynie suma znalezionych wierszy, czas wykonania jest ponad dwukrotnie krótszy w przypadku klasycznego, nierozmytego zapytania, co pokrywa się z przypuszczeniami. Na podstawie przeprowadzonego testu można wysnuć wniosek, że 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.

Rys. 4.1 Wykres czasu wykonania prostych zapytań

Kod zapytań wykorzystanych w drugim teście zaprezentowany jest w listingu 4.6. 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 poniżej na Rys. 4.2.



Rys. 4.2 Wynik zapytania przyporządkowującego liczbę osób do zakresów ciśnienia tętniczego

*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

Uzyskane czasy dostępu zaprezentowano na wykresie na Rys. 4.3. 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. 4.3 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. 4.4. 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. 4.4 Wykres czasu wykonania zapytania agregującego na podstwie jednej funkcji rozmytej

Poniżej, w tabeli 4.1 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 klasycznego |
| 1. Proste | 212% |
| 1. Agregujące nazwy typu | 17 764% |
| 1. Agregujące stopień przynależności do funkcji | 173% |

Tabela 4.1 Podsumowanie wyników badania wydajności operatorów rozmytych

## 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). Z uwagi na sposób działania silnika języka JavaScript, czas wykonania poszczególnych etapów nie powinien być w żaden sposób zależny, tj. zwiększenie czasu wykonania jednego z etapów nie wpływa na drugi. Oczywistym jest, że objętość danych zwracanych w wyniku zapytania wpływa na czas wykonania wszystkich etapów. Badanie ma na celu pokazać zależność czasu wykonania zapytań od ilości danych. Mierzony czas, złożony się z wyżej wymienionych etapów, można uznać za składowe czasu, przez który użytkownik oczekuje na wynik.

## 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 ilości danych do testów, zbiór został zduplikowany tak samo, jak zostało to opisane w rozdziale 4.4.

## Wyniki pomiaru wydajności aplikacji Fuzzybase

Wykres liniowy utworzony na podstawie danych zebranych z aplikacji znajduje się na Rys. 4.5. Z uwagi na szeroki zakres danych pomiarowych, zostały przedstawione one również w tabeli 4.2.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **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 |
| **IPC** | 11 | 17 | 14 | 23 | 31 | 64 | 118 | 230 | 565 | 1140 | 2350 |
| **SQL** | 13 | 19 | 17 | 26 | 38 | 81 | 149 | 289 | 712 | 1440 | 2983 |

Tabela 4.2 Podsumowanie danych pomiarowych dot. wydajności aplikacji Fuzzybase

Wyniki pokazują liniową zależność między czasem wykonania poszczególnych etapów względem ilości zwracanych wierszy. W przypadku czasu wykonania etapu rysowania tabeli z rezultatem zapytania (VDOM) 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 rozmiar 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.

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 4.2.

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

# Podsumowanie

optymalizacja, otwarcie źródła, rozszerzalność, klient sql

Rozdział ten zawiera:

* syntetyczny opis wykonanych prac,
* wnioski,
* opis możliwości rozwoju, kontynuacji prac, potencjalne nowe kierunki,
* informację, czy cel pracy zrealizowany.

# 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. Kinney, w *Electron in Action*, Manning Publications, 2018, pp. 3-16. |
| [20] | „Chromium for Developers,” [Online]. Available: https://www.chromium.org/developers/content-module. [Data uzyskania dostępu: 09 10 2018]. |
| [21] | 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]. |
| [22] | A. Janosi, W. Steinbrunn, M. Pfisterer i R. Detrano, „Heart Disease,” 1988. [Online]. Available: https://data.world/uci/heart-disease. |

# Spis skrótów i symboli

|  |  |
| --- | --- |
| DNA | kwas deoksyrybonukleinowy (ang. *deoxyribonucleic acid*) |
| MVC | model – widok – kontroler (ang. *model–view–controller*) |
| N | Liczebność zbioru danych |

# 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 14](#_Toc527843033)

[Rys. 3.1 Schemat działania procesów w platformie Electron 22](#_Toc527843034)

[Rys. 3.2 Przepływ danych w architekturze Flux 23](#_Toc527843035)

[Rys. 3.3 Cykl życia komponentu React 25](#_Toc527843036)

# Spis tabel

[Tabela 2.1 Operatory logiczne 13](#_Toc527843173)