POLSKO-JAPOŃSKA WYŻSZA SZKOŁA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Jacek Olszak

Nr albumu 6941

Marcin Świerczyński

Nr albumu 3759

JShards - transparentne wykorzystywanie rozproszonych baz danych zapewniające horyzontalną skalowalność aplikacji

Praca magisterska napisana pod kierunkiem:

dr inż. Mariusz Trzaska

Warszawa, wrzesień 2010

Streszczenie

Praca dotyczy istotnego praktycznego problemu zwiększenia przepustowości aplikacji internetowych, poprzez wykorzystanie rozproszonych baz danych w sposób transparentny. Klasyczne podejście zakłada wykorzystanie jednego fizycznego serwera bazodanowego, który przechowuje kompleksowe dane aplikacji. Mając na uwadze fakt, że operacje dyskowe są jednymi z najbardziej pracochłonnych w trakcie wykonywania dowolnych programów komputerowych, łatwo dojść do wniosku, iż serwer bazodanowy stanowi „wąskie gardło" aplikacji webowych. Efekt ten jest potęgowany ze względu na specyfikę tych aplikacji. Oprogramowanie to w sposób naturalny przeznaczone jest do działania w środowisku rozproszonym. Popularne serwisy internetowe muszą często obsługiwać dziesiątki tysięcy zapytań na sekundę, generowanych przez tysiące użytkowników. Oczywistym rozwiązaniem problemu jest rozbudowa i ulepszenie istniejącego serwera, czyli skalowanie wertykalne. Taki zabieg jest jednak nieefektywny ze względów ekonomicznych, ponieważ osiąganie sukcesywnie lepszych wyników wiąże się z koniecznością zakupu najlepszych rozwiązań sprzętowych, które charakteryzują się bardzo wysoką ceną. Rozwiązaniem problemu, które przedstawiono w niniejszej pracy jest skalowanie horyzontalne. Polega ono na rozdzieleniu odpowiedzialności serwera bazodanowego na wiele fizycznych maszyn. Dzięki takiemu podejściu można przygotować aplikacje na ogromne obciążenia, zachowując jednocześnie racjonalizm ekonomiczny. Dzieje się tak, gdyż zakup wielu słabszych serwerów bazodanowych jest zdecydowanie mniejszym wydatkiem, niż zakup jednego „super-komputera". Dodatkowym atutem jest fakt, że możliwości skalowania horyzontalnego są praktycznie nieograniczone, podczas gdy skalowanie wertykalne jest ograniczone przez rozwój techniki i sprzętu komputerowego. Praktyczną realizację omówionych koncepcji stanowi prototypowa aplikacja, którą zbudowano w ramach tej pracy. Oprogramowanie to ma formę sterownika JDBC na platformę Java. Dzięki niemu programiści nie muszą znać architektury sprzętowej. Prototyp ten umożliwia transparentną obsługę rozproszonych baz danych za pomocą większości dostępnych konstrukcji języka SQL.

Spis treści

1. Wstęp

W ramach tej pracy autorzy próbują rozwiązać problem dręczący świat aplikacji WWW - niewielką skalowalność aplikacji internetowych wynikającą z wykorzystania scentralizowanego serwera bazodanowego. Serwer taki jest „wąskim gardłem" każdej aplikacji, ponieważ operacje dyskowe stanowią jedne z najbardziej pracochłonnych operacji każdego rodzaju oprogramowania. Konieczność obsługi wielu tysięcy użytkowników, którzy generują dziesiątki tysięcy zapytań jednocześnie powoduje nieustające problemy wydajnościowe. Klasyczna metoda zaradzenia tym problemom polega na zainwestowaniu w bardziej wydajną infrastrukturę. Zwiększanie przepustowości poprzez wymianę serwera na szybszy nazywamy skalowaniem wertykalnym. Jednakże komputery, które mają podołać tak dużym obciążeniom, muszą być niezwykle wydajne co wiąże się z dużymi wydatkami, które należy ponieść na rzecz zakupu nowego sprzętu. Co więcej, zakup nowego sprzętu prawie zawsze wiąże się ze stratami wynikającymi z utratą mocy obliczeniowej udostępnianej przez dotychczasowy komputer, ponieważ maszyny te nie mogą pracować jednocześnie.

Odpowiedzią na te problem jest architektura skalowalna horyzontalnie. U podstaw tej koncepcji leży obserwacja mówiąca, że koszt pojedynczego komputera o określonej mocy obliczeniowej znacznie przewyższa koszt kilku komputerów, których łączna mocy obliczeniowa jest porównywalna. W tym kontekście naturalnym wydaje się pomysł rozdzielenia odpowiedzialności serwera bazodanowego na kilka fizycznie oddzielnych maszyn. Niestety współczesne systemy zarządzania bazami danych udostępniają tylko podstawową funkcjonalność w tym zakresie. W szczególności nie umożliwiają wykonania zapytań rozproszonych w sposób transparentny dla programisty. Ta obserwacja była bezpośrednim bodźcem do stworzenia warstwy pośrednika, dzięki któremu możliwe jest korzystanie z rozproszonych baz danych w sposób przezroczysty dla programisty.

Kolejny powód, który skłonił autorów pracy do zajęcia się tym problemem był czysto pragmatyczny. W swojej pracy zawodowej zajmują się oni tworzeniem aplikacji internetowych. Niejednokrotnie spotykali się oni z problemem wydajności spowodowanym brakiem skalowalności bazy danych. Tworzenie kodu wykorzystującego jawne odwołania do

rozproszonych baz danych jest skomplikowane i czasochłonne, przez co cała aplikacja narażona jest na większą ilość błędów.

Planuje się, aby prototyp powstały w wyniku podjętych prac był wykorzystywany w rozwiązaniach produkcyjnych, a także wzbogacany o kolejne funkcjonalności.

1.1. Cel pracy

Celem pracy jest opracowanie koncepcji sterownika JDBC w języku programowania Java, który ułatwi tworzenie i utrzymywanie horyzontalnie skalowalnych aplikacji internetowych. Z racji faktu, iż narzędzie to będzie stanowiło implementację standardu JDBC, jego użycie będzie zupełnie przezroczyste dla klienta. Docelowo będzie on mógł wykorzystywać wszystkie znane mu techniki pracy z bazami danych w języku Java, w szczególności klasy Connection, Statement, ResultSet. Dzięki temu programistę ominie konieczność fizycznego konfigurowania baz danych. Efektem tych starań będzie możliwość tworzenia mniejszej ilości kodu przy zachowaniu docelowej funkcjonalności. To z kolei sprawi, że zmniejszy się ryzyko popełnienia błędu, więc jakość oprogramowania wzrośnie. Ponadto oprogramowanie takie będzie można szybciej dostarczyć do klienta.

Uniwersalność przyjętej koncepcji powoduje, że narzędzie będzie mogło być wykorzystywane także przez administratorów baz danych. Sterownik będzie można śmiało wykorzystać w klientach SQL pokroju SQuirreL SQL Client, dzięki czemu administratorzy będą w stanie wygodnie zarządzać rozproszoną architekturą skalowalną horyzontalnie.

W związku z powyższymi, ostatecznym celem pracy jest zmniejszenie wydatków na utrzymanie infrastruktury zarówno sprzętu, jak i oprogramowania.

1.2. Rozwiązania przyjęte w pracy

Istotą prezentowanego rozwiązania jest podzielenie bazy danych, na której operuje aplikacja kliencka. Podziału tego dokonuje się poziomo, tzn. tak, aby podzbiory rekordów przechowywanych pierwotnie w monolitycznej bazie danych, znalazły się w fizycznie oddzielnych bazach, tzw. „shardach”. Schematy poszczególnych shardów są identyczne.

Pierwszym krokiem, który należy podjąć podczas wdrażania architektury skalowalnej horyzontalnie jest wybór ilości i fizycznej lokalizacji shardów. Sterownik wspomaga w tym miejscu użytkownika, udostępniając plik konfiguracyjny YAML, w którym można wskazać fizyczną lokalizację poszczególnych baz danych. Lokalizacja pliku konfiguracyjnego jest podawana jako parametr podczas pobierania połączenia – jest to jedyne miejsce, w którym użytkownik ma świadomość, że nie ma do czynienia z klasycznym sterownikiem scentralizowanej bazy danych.

Podstawą działania narzędzia jest koncepcja przechwytywania zapytania przez sterownik, jego ewentualnej modyfikacji na potrzeby architektury rozproszonej oraz wybranie odpowiednich baz danych z puli dostępnych shardów. Ze względów optymalizacyjnych zapytania uruchamiane są jedynie na tych shardach, które zawierają dane charakteryzowane przez warunki zapytania. Ponadto wszystkie operacje, których wykonanie można przełożyć na rzecz poszczególnych baz cząstkowych, są w nich realizowane. Inne, jak na przykład obliczanie wartości średniej, są wykonywane lokalnie, w pamięci RAM.

O wyborze sharda decyduje strategia wyboru, która jest dostarczana przez użytkownika w postaci klasy implementującej określony interfejs. Dla wygody użytkownika końcowego dostarczono przykładowe implementacje, konfigurowalne przez plik YAML. Będą one wystarczające w wielu przypadkach, a ponadto będą stanowiły wzorzec dla własnych implementacji.

Sterownik JShards został zaimplementowany w obiektowym języku programowania Java. Pozwoliło to na zastosowanie wysokopoziomowego programowania przy jednoczesnym utrzymaniu wysokiej wydajności związanej ze statyczną kontrolą typologiczną.

Jako że narzędzie zrealizowane jest w postaci sterownika JDBC, specyfikacja standardu Java Database Connectivity opisana w [1] była wykorzystywana w trakcie całego procesu wytwarzania oprogramowania.

Testy sterownika wykonano na bazie PostgreSQL.. Zdecydowano się na ten krok ze względu na dużą popularność i możliwości tego systemu, przy jednoczesnym zachowaniu statusu oprogramowania darmowego.

Większość kodu powstała specjalnie na potrzeby tej pracy. Wyróżnić możemy jedynie trzy zewnętrzne biblioteki:

- JSQLParser[1] – parser języka SQL, który przekłada jego konstrukcje na klasy Javy. Nawigacja po tych klasach odbywa się poprzez wykorzystanie wzorca projektowego Visitor.
- Google Collections[2] – zbiór nowych kolekcji i użytecznych mechanizmów manipulacji kolekcjami stanowiący naturalne rozszerzenie Java Collections Framework. Zastosowanie tej biblioteki znacznie poprawiło jakość i czytelność kodu.
- SnakeYAML[3] – parser dokumentów w formacie YAML, wykorzystywany do interpretacji plików konfiguracyjnych.

Wszystkie biblioteki dostępne są na zasadach open-source.

Ponadto do budowy sterownika wykorzystano narzędzie do testów jednostkowych – JUnit. Dzięki użyciu zautomatyzowanych testów, dużo łatwiej było kontrolować poprawność działania aplikacji, co przyczyniło się bezpośrednio do wzrostu jej jakości.

1.3. Rezultaty pracy

Rezultatem pracy jest koncepcja sterownika JDBC realizującego postawione wcześniej wymagania. W trakcie realizacji jego prototypu natknięto się na kilka interesujących problemów. Część z nich udało się rozwiązać, a część zostanie rozwiązana w przyszłości, w trakcie dalszych badań nad rozważaną dziedziną.

Podstawową kwestią, jaką należy poruszyć jest fakt, iż nie wszystkie operacje charakterystyczne dla relacyjnych baz danych można wykonać w sposób bezpośredni w architekturze shardów. Jako przykład rozważmy funkcje agregujące. Funkcje SUM, MAX i MIN są stosunkowo proste w implementacji. Wystarczy uruchomić wybraną funkcję na

[1] http://jsqlparser.sourceforge.net/
[2] http://code.google.com/p/google-collections/
[3] http://code.google.com/p/snakeyaml/

każdej bazie cząstkowej, a następnie jeszcze raz przeprowadzić tę samą operację lokalnie, na otrzymanych wynikach. Jednakże funkcja AVG nie pozwala na podobny zabieg, ponieważ wartość średnia wszystkich elementów nie jest równa wartości średniej z cząstkowych wartości średnich. W tej sytuacji zdecydowano się na uruchomienie na każdej z baz dwóch funkcji: SUM i COUNT i obliczenie finalnej wartości średniej na podstawie ich wyników. Te i inne problemy szczegółowo omówiono w rozdziałach 4 i 6.

Zastosowanie prototypu sterownika JShards przyniosło wzrost przepustowości na poziomie 32,5 % w stosunku do standardowego rozwiązania nie wykorzystującego skalowalności horyzontalnej. Wynik ten dotyczy użycia jedynie dwóch shardów. Można śmiało założyć, że zastosowanie większej ich liczby spowoduje dalszy wzrost przepustowości. Szczegółowe testy przedstawione zostały w Rozdziale 5.3.

1.4. Organizacja pracy

Pracę podzielono na 8 rozdziałów. Poniżej przedstawiono krótką charakterystykę najważniejszych z nich.

W rozdziale 2 omówiono stan sztuki, czyli istniejące próby rozwiązania problemu. Tutaj też opisano zalety prezentowanego rozwiązania.

Rozdział 3 zawiera krótką charakterystykę zewnętrznych narzędzi, które wykorzystano przy budowie omawianej aplikacji. Także tu omówiono koncepcję rozproszonych baz danych.

Rozdział 4 to szczegółowe omówienie projektu prototypu. Opisano tam sposób tworzenia sterownika JDBC, mechanizm przepisywania zapytań (ang. rewriting), a także koncepcję strategii wyboru sharda. Tutaj także wspomniano o sposobie analizy i przetwarzania wyników zapytań napływających z rozproszonych baz danych.

Rozdział 5 zawiera opis szczegółów implementacyjnych sterownika JShards. Omówiono w nim wykorzystany parser języka SQL oparty o wzorzec projektowy Visitor. Z detalami opisano implementację klasy ShardsStatement, dzięki której możliwa jest obsługa zapytań w architekturze shardów. Tutaj też przedstawiono wyniki przeprowadzonych testów wydajnościowych.

W 6 rozdziale zawarto podręcznik użytkownika. Jest to miejsce, dzięki któremu można szybko zorientować się, jak korzystać ze stworzonej aplikacji.

Rozdział 7 z kolei omawia problemy realizacyjne i plany dalszego rozwoju narzędzia. Poruszono tam problem złączeń w rozproszonej architekturze systemów bazodanowych. Omówiono także problemy wydajnościowe, w szczególności kwestię zrównoleglenia wykonywanych operacji oraz przenoszenia cząstkowych wyników z pamięci operacyjnej do pamięci trwałej.

2. Istniejące rozwiązania

Poniższe podrozdziały omawiają istniejące rozwiązania, dzięki którym możliwe jest osiągnięcie wzrostu wydajności aplikacji internetowych. Omówiono tu podstawowe techniki optymalizacji, takie jak indeksy, metody oparte o replikację i shardy, a także rozwiązania alternatywne zrywające z modelem relacyjnym.

2.1. Podstawowe techniki optymalizacji

Istnieje wiele technik stosowanych do poprawy wydajności i przepustowości relacyjnych baz danych. W wielu przypadkach są one wystarczające, dlatego przed wykorzystaniem bardziej zaawansowanych rozwiązań trzeba z nich skorzystać.

Podstawową techniką optymalizacji wydajności relacyjnej bazy danych są indeksy [2]. Wprowadzenie indeksu to prosty sposób na poprawienie wydajności zapytań SELECT. Administrator bazy danych na bieżąco monitoruje obciążenie serwera i czasy wykonywania poszczególnych poleceń. Na tej podstawie może podjąć decyzję o założeniu bądź usunięciu indeksu. Założenie indeksu może przyspieszyć zapytanie nawet setki razy, jednak w niektórych przypadkach okazuje się nieopłacalne. Należy pamiętać, że indeks to obiekt bazy danych, który musi być aktualizowany wraz z każdą zmianą wierszy w tabeli. Wszystko zależy od stosunku ilości zapytań SELECT do poleceń aktualizacji UPDATE lub INSERT. Gdy ilość zapytań SELECT jest kilkukrotnie większa to wprowadzenie indeksu na ogół jest opłacalne. Obecnie większość relacyjnych baz danych dostarcza możliwość zakładania indeksów na kolumnach tabeli. Ponadto, niektóre bazy danych dają też możliwość zakładania indeksów na funkcjach. Dostępne są głównie indeksy typu B-Tree, jednak można się spotkać także z indeksami typu Bitmap lub Hash. Więcej informacji o indeksach można znaleźć w [2].

Gdy wprowadzenie indeksów jest ciągle niewystarczające należy zastanowić się nad optymalizacją poszczególnych zapytań, czyli przepisaniem ich na bardziej optymalną formę. Czasem wykorzystanie podzapytania jest bardziej optymalne od wykonywania złączeń. W innych przypadkach opłaca się zawęzić zbiór zwracanych wyników poprzez modyfikację klauzuli WHERE. Może się także okazać, że pomimo istnienia indeksu baza z niego nie korzysta. W takiej sytuacji należy dodać warunek do klazuli WHERE lub przekazać w

zapytaniu wskazówkę (ang. „hint”). Optymalizacja zapytań jest zabiegiem bardzo złożonym i specyficznym dla danej bazy danych. Potrzeba wielu lat doświadczeń administratora, aby był on w stanie wykorzystać optymalnie silnik baz danych. Szczegółowe informacje o optymalizacji można znaleźć w [3] oraz [2].

Po wprowadzeniu indeksów i optymalizacji zapytań należy przeanalizować wykonywane transakcje. W systemach mocno obciążonych transakcje mogą znacząco wpływać na obniżenie przepustowości. Powodem takiego stanu rzeczy są blokady. Długo wykonujące się transakcje mogą skutecznie zablokować inne transakcje, zmniejszając ilość symultanicznie wykonywanych operacji. Optymalizacja polega głównie na wyeliminowaniu blokowania oraz zmniejszeniu liczby operacji w pojedynczej transakcji. Przykładowo w większości przypadków niepotrzebne jest blokowanie całych tabel. Często warto rozważyć technikę optymistycznego blokowania – zamiast wykorzystywać blokady bazodanowe lepiej dodać kolumnę wersji do tabeli. Stosuje się też optymalizację poziomu izolacji transakcji – w wielu przypadkach poziom READ COMMITTED jest w zupełności wystarczający.

Inną techniką zwiększenia przepustowości bazy danych jest buforowanie wyników zapytań w pamięci operacyjnej. Część relacyjnych baz danych ma takie funkcjonalności wbudowane – w momencie uruchamiania zapytania baza danych sprawdza czy zapytanie nie zostało wcześniej zbuforowane. Można też wykorzystać systemy dedykowane np. memcached lub skorzystać z bibliotek dla danego języka programowania np. Ehcache[4]. W tym przypadku odpowiedzialność buforowania wyników spoczywa na aplikacji. Buforowanie to powszechna technika stosowana w aplikacjach www – nie tylko do przechowywania wyników zapytań SQL. Umiejętne jej wykorzystanie może nawet kilkukrotnie przyspieszyć działania aplikacji.

Wydajność i przepustowość relacyjnych baz danych można poprawiać także poprzez optymalizację infrastruktury sprzętowej. Wymiana lub dodanie procesorów oraz pamięci operacyjnej może znacznie zmniejszyć czasy wykonywania zapytań. Większa ilość pamięci RAM może zostać wykorzystana także do buforowania całych wyników zapytań. Jednak największym problemem obniżającym wydajność systemu bazodanowego jest pamięć

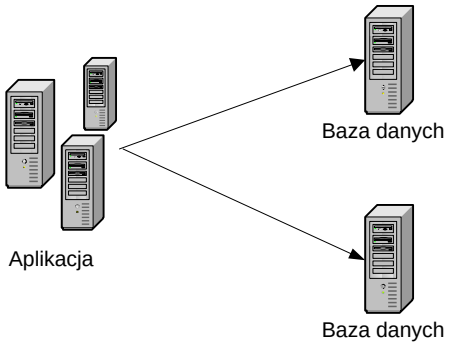
[4] http://ehcache.org/

masowa. Czasy odczytu i zapisu danych na tego typu urządzeniach są kilkunastokrotnie większe. Dlatego, aby zwiększyć przepustowość, wykorzystuje się rozmaite techniki polegające na wykorzystaniu wielu dysków jednocześnie – tzw. macierzy dysków. RAID (ang. *Redundant Array of Independent Disks*) to jedna z tego typu technologii. Dzięki niej można połączyć ze sobą wiele dysków twardych, które przez system będą rozpoznawane jako jeden logiczny dysk. RAID dostarcza podstawową technikę zwiększenia przepustowości - striping. Polega ona na podzieleniu pliku na kilka części, przy czym każda część przechowywana jest na innym dysku. Operacje odczytu i zapisu wykonywane są równolegle na odpowiednich dyskach. Dzięki temu czasy ulegają znacznermu skróceniu. Szczegółowe informacje o technologii RAID można znaleźć w [4].

Striping może być wykorzystywany także na wyższym poziomie – w silniku relacyjnej bazy danych. Część dostępnych na rynku baz danych dostarcza możliwość partycjonowania tabel. Zbiory rekordów są dzielone na części (partycje) i przechowywane na różnych dyskach. Sposobów podzielenia rekordów na partycje jest kilka. Najpopularniejszą techniką jest podział wg zakresu. Przykładowo dla tabeli *użytkownicy* istnieją kolumny *login* oraz *hasło*. Rekordy z loginem zaczynającym się od liter A-H będą przechowywane na dysku pierwszym. Pozostałe na dysku drugim. Zalety wykorzystania partycjonowania tabel są identyczne do stripingu na poziomie sprzętowym. Partycjonowanie tabel pozwala jednak na lepszą optymalizację kosztem większych nakładów pracy na konfigurację.

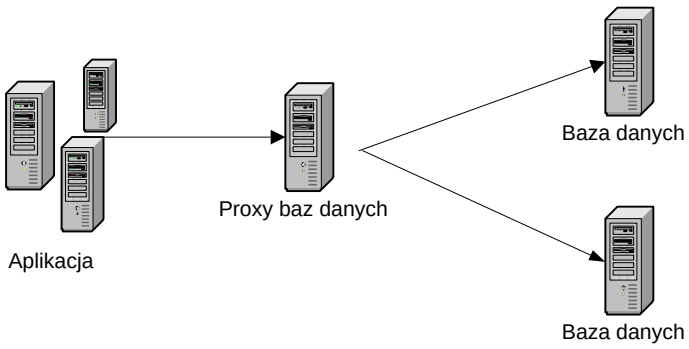
2.2. Replikacja

W wysokoobciążonych aplikacjach wykorzystanie pojedynczego serwera bazodanowego może być niewystarczające. Logiczne wydaje się więc korzystanie z wielu fizycznych maszyn jednocześnie. Serwery www zawierające aplikacje mogą łączyć się z dowolną z tych maszyn za pomocą technik równoważenia obciążenia (*load balancing*).



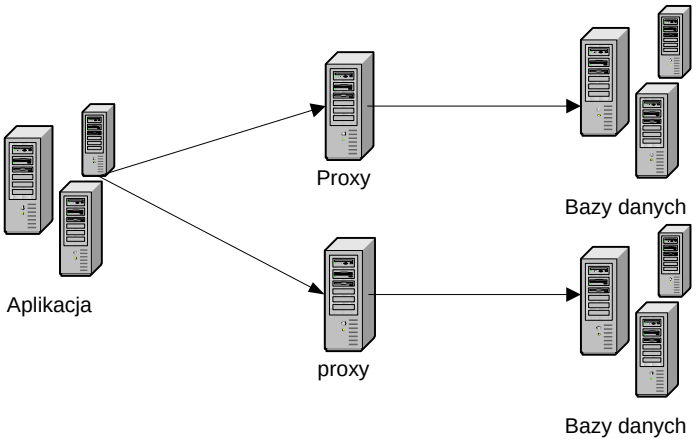
Rysunek 1 Równoważenie obciążenia po stronie aplikacji

Równoważenie obciążenia można realizować po stronie aplikacji, jak przedstawia to Rysunek 1 lub za pomocą serwera pośredniego (tzw. *proxy*), jak na Rysunek 2.



Rysunek 2 Równoważenie obciążenia za pomocą serwera proxy

Ze względu na to, że jeden serwer proxy pośredniczy w wymianie wszystkich komunikatów między aplikacją a bazami danych jest to rozwiązanie z ograniczoną skalowalnością. W sytuacji gdy liczba baz danych jest duża należy zastosować wiele serwerów proxy.



Rysunek 3 Równoważenie obciążenia przez wiele serwerów proxy

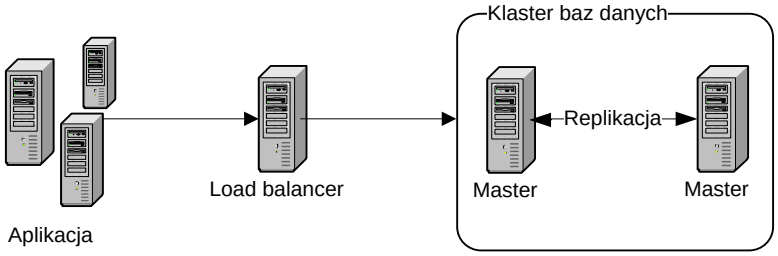
Wykorzystanie wielu serwerów bazodanowych przechowujących zawartość jednej logicznej bazy danych pociąga za sobą konieczność replikowania zmian między maszynami. Po modyfikacji danych przez aplikację muszą być one rozpropagowane do pozostałych baz. Sposób replikacji danych znacząco wpływa na skalowalność rozwiązania.

Istnieją dwa rodzaje replikacji – synchroniczna i asynchroniczna. W przypadku replikacji synchronicznej modyfikacje przesyłane są do pozostałych serwerów na bieżąco. Transakcja nie może zostać zakończona dopóki zmiany nie zostaną zapisane na wszystkich serwerach. Z kolei replikacja asynchroniczna polega na przesyłaniu zmian w sposób opóźniony – np. po zapełnieniu bufora. Aktualna transakcja nie czeka na zakończenie procesu replikacji.

Wybór architektury zależy od wybranego rodzaju replikacji. Przed przestawieniem architektur omówione zostaną podstawowe pojęcia. Serwery wchodzące w skład klastra mogą nosić nazwę *master* lub *slave*. Master to serwer, do którego trafiają wszystkie żądania – zarówno odczytu jak i zapisu. Slave to serwer służący tylko do odczytu.

2.2.1 Architektura multi-master

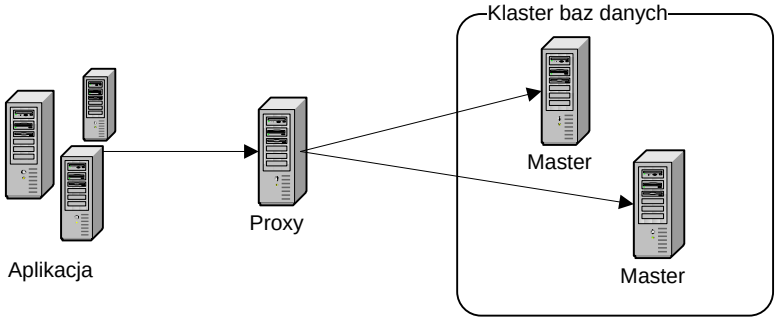
Architektura multi-master polega na wykorzystaniu tylko serwerów master. Dana transakcja w całości wykonywana jest na wybranym serwerze w klastrze. Wszelkie wprowadzone zmiany zapisywane są w specjalnym logu a następnie przesyłane w sposób synchroniczny lub asynchroniczny do pozostałych serwerów baz danych.



Rysunek 4 Architektura multi-master z load-balancerem

Wprowadzenie takiej architektury może być całkowicie przezroczyste dla aplikacji. Wystarczy wykorzystać load balancer, który będzie równoważył obciążenie kierując wykonywanie transakcji do odpowiedniego serwera. Niestety „replikacja w tle" może powodować konflikty. Może się zdarzyć sytuacja, w której dwie różne transakcje równolegle zmodyfikują te same dane, co spowoduje utratę ich spójności.

Innym rozwiązaniem jest zastosowanie serwera proxy, który oprócz równoważenia obciążenia dla zapytań SELECT zajmuje się także równoległym trasowaniem poleceń modyfikacji do wszystkich serwerów baz danych.



Rysunek 5 Architektur multi-master z serwerem proxy

Serwer proxy wykorzystany w tej architekturze oprócz równoważenia obciążenia musi wykonywać przepisywanie (ang. *rewrite*) zapytań w przypadku transakcji. Może się bowiem zdarzyć sytuacja, w której uruchomienie tego samego zapytania na różnych serwerach zwróci inny rezultat. Przykładowo zapytanie w PostgreSQL przedstawione na Listing 1 zwróci aktualny czas, który będzie się różnił w zależności od obciążenia sieci i różnicy w czasie systemowym poszczególnych serwerów.

```
SELECT now()
```

Listing 1

Obowiązkiem serwera proxy jest więc uruchomienie takich funkcji lokalnie i przepisanie zapytania, aby zamiast funkcji zawierało jej wynik. Dla powyższego przykładu przepisane zapytanie będzie miało postać przedstawioną na Listing 2.

```
SELECT '2010-03-31 17:37:29.773536+02'::TIMESTAMPTZ
```

Listing 2

Na rynku dostępnych jest kilka rozwiązań pozwalających wykorzystać architekturę multi-master. Większość z nich dedykowanych jest konkretnym implementacjom baz danych. Na przykład dla PostgreSQL dostępne jest narzędzie PGCluster[5] wykorzystujące replikację w tle. Można także spotkać rozwiązania bardziej uniwersalne dedykowane dowolnym bazom danych np. Sequoia[6] lub Tungsten SQL Router[7] wykonujące trasowanie zapytań.

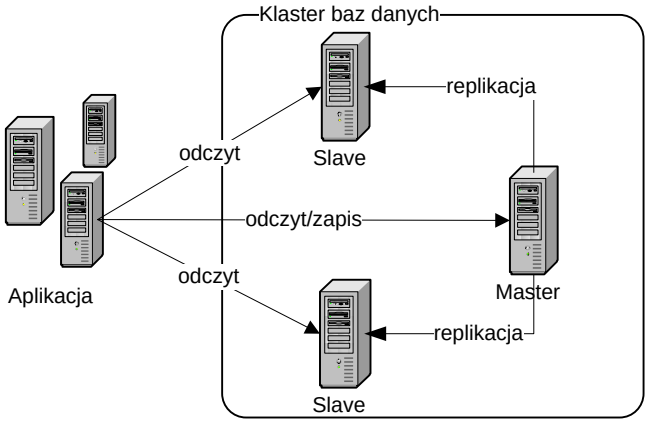
Architektura multi-master oprócz zwiększenia przepustowości zapewnia także pracę w trybie failover. Zadania serwera, który uległ awarii mogą być wówczas przetwarzane przez inne serwery.

W przypadku architektury multi-master bardzo ryzykowne wydaje się być wykorzystanie replikacji asynchronicznej. Prawdopodobieństwo konfliktu danych jest tym większe im większy jest lag replikacyjny, czyli czas, który upłynął od momentu modyfikacji danych w „oryginalnej" bazie do czasu propogacji tych zmian w innej bazie. W przypadku serwerów produkcyjnych wysokoobciążonych aplikacji czas ten może sięgać nawet do kilkudziesięciu minut. Bardziej bezpieczne jest więc wykorzystanie replikacji synchronicznej, co wpływa jednak na obniżenie wydajności rozwiązania.

[5] http://pgcluster.projects.postgresql.org/
[6] http://www.continuent.com/community/lab-projects/sequoia
[7] http://www.continuent.com/community/tungsten-sql-router

2.2.2 Architektura master-slave

Architektura master-slave w przeciwieństwie do multi-master jest rozwiązaniem bardziej bezpiecznym pozwalającym zachować wszystkie warunki ACID. W architekturze tej istnieje tylko jeden serwer master, do którego kierowane są wszystkie transakcje aktualizujące. Z kolei zapytania do odczytu trafiają na jeden z wielu serwerów slave. Replikacja najczęściej odbywa się w tle – po każdej modyfikacji danych tworzony jest log a następnie przesyłany do wszystkich serwerów slave.



Rysunek 6 Architektura master-slave

Wykorzystanie tej techniki nie jest przezroczyste dla aplikacji. Należy stworzyć odpowiedni kod, który będzie w stanie wykorzystać zalety master-slave. Część zapytań musi być wykonywana na specjalnym połączeniu przeznaczonym tylko do odczytu, zaś transakcje aktualizujące muszą być wykonywane na połączeniach do serwera master.

Istnieje wiele narzędzi pozwalających wdrożyć architekturę master-slave. Przykładowo dla PostgreSQL najpopularniejszą technologią jest Slony-I bazująca na wyzwalaczach, które zapisują logi w specjalnej tabeli. W bazie danych MySQL replikacja master-slave jest wbudowana.

W celach optymalizacyjnych w architekturze master-slave najczęściej wykorzystywaną metodą replikacji jest replikacja asynchroniczna. Dopuszczalne jest, aby dane na serwerach slave były przestarzałe (ang. *stale data*). Nie powoduje to utraty spójności gdy dane te nie są wykorzystywane w transakcjach. Aby wyeliminować pojedynczy punkt awarii (ang. *single*

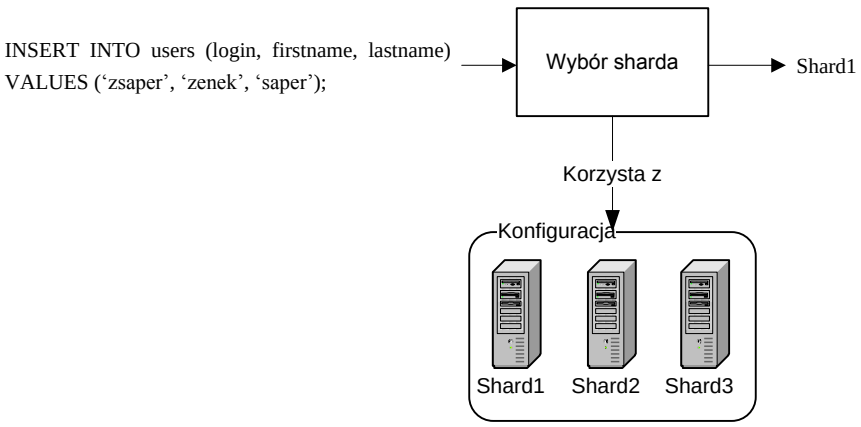
point of failure) można dodatkowo stosować replikację synchroniczną z jednym serwerem slave. Dzięki temu zawsze będzie istnieć wierna kopia serwera master, na którą będzie się można przełączyć w razie awarii.

2.3. Sharding

Replikacja może skutecznie zwiększyć przepustowość systemu jednak nie skaluje się do naprawdę wysokoobciążonych aplikacji. Dodawanie kolejnych fizycznych serwerów zwiększa głównie przepustowość wykonywania żądań tylko do odczytu, czyli zapytań SELECT. Nie poprawia jednak w znaczący sposób wydajności poleceń modyfikacji.

Sharding to technika pozwalająca podzielić dane pomiędzy różne serwery. Zapytania wykonywane są tylko na tych serwerach, na których znajdują się dane lub w przypadku dodawania nowych rekordów na serwerach, które te dane mają zawierać. Sharding jest bardzo podobny do partycjonowania tabel, jednak działa na poziomie całego klastra serwerów, a nie tylko dysków twardych.

O tym, na których serwerach uruchomić zapytanie decyduje algorytm (strategia) wyboru przyjmujący na wejściu zapytanie i zwracający nazwy serwerów.



Rysunek 7 Sharding

W powyższym przykładzie widać, że zapytanie INSERT zostanie przesłane jedynie do serwera Shard1.

Istnieje kilka implementacji mechanizmu shardingu. Pierwszą z nich jest Hibernate Shards[8]. To rozszerzenie Hibernate - znanego ORM[9] dla języka Java, które umożliwia użytkownikowi wykonywanie zapytań na wielu serwerach. Niestety Hibernate Shards korzysta z algorytmu wyboru sharda tylko dla operacji dodawania (INSERT) nowych rekordów oraz aktualizacji rekordów po kluczu głównym. Zapytania SELECT wykonywane są na wszystkich serwerach, co w przypadku dużej liczby shardów może skutecznie spowolnić system. Nie jest to więc rozwiązanie idealne dla większych wdrożeń i nie do końca realizuje zagadnienie shardingu.

Innym rozwiązaniem jest PL/Proxy[10] firmy Skype Technologies S.A wykorzystywanym w bazie danych użytkowników komunikatora Skype. Jest to technologia dedykowana do relacyjnej bazy danych PostgreSQL. Aplikacja łączy się z serwerem proxy i uruchamia na nim specjalne funkcje tworzone w języku plproxy (rozszerzenie pgPL/SQL). Funkcje te dokonują wyboru sharda a następnie uruchamiają na nim kod. Wykonywanie zapytań w klastrze nie jest więc przezroczyste dla programisty. To on jest odpowiedzialny za jawne uruchomienie kodu wybierającego shard oraz złączenie rezultatów przesłanych z poszczególnych serwerów. Aplikacje tworzone z wykorzystaniem PL/Proxy są więc bardzo trudne w utrzymaniu i rozwijaniu.

Sharding zaczyna zyskiwać popularność, ponieważ rośnie zapotrzebowanie na coraz większą przepustowość systemów. Dzisiejsze aplikacje www obsługują już miliony użytkowników a istniejące techniki nie pozwalają obsłużyć tak dużego ruchu. Dlatego w najbliższym czasie z pewnością pojawią się kolejne narzędzia realizujące sharding.

2.4. Alternatywne rozwiązania

Relacyjne bazy danych i istniejące techniki ich optymalizacji często nie wystarczają aby obsłużyć wysokobociążone serwery. Dlatego część firm decyduje się na wykorzystanie alternatywnych silników baz danych, które zrywają z modelem relacyjnym.

[8] http://www.hibernate.org/subprojects/shards.html
[9] http://en.wikipedia.org/wiki/Object-relational_mapping
[10] https://developer.skype.com/SkypeGarage/DbProjects/PlProxy

Bazy relacyjne ze względu na swoją specyfikę są dość trudne w optymalizacji – szczególnie jeśli chcemy wykorzystać bardzo wiele maszyn jednocześnie. Powodów jest wiele, jednak najważniejszymi problemami są transakcje, złączenia (ang. *join*) oraz spójność danych. Transakcje wykonywane w środowisku rozproszonym muszą korzystać z protokołów zatwierdzania rozproszonego np. *Two-Phase Commit*. Protokoły te są bardzo wolne oraz nieodporne na błędy. W transakcji rozproszonej uczestniczyć może wiele serwerów. Awaria jednego powoduje wycofanie całej rozproszonej transakcji. Wykorzystując bazy relacyjne trzeba więc zainwestować w niezawodny sprzęt oraz sieć.

Złączenia to operacje, które degradują wydajność wykonywania zapytań w środowisku rozproszonym. Przez ich obecność w wielu przypadkach nie jest możliwe wybranie odpowiedniego sharda, przez co zapytanie musi być uruchomione na każdym serwerze. Kolejny problem to spójność danych. W środowisku rozproszonym ze względu na problemy replikacyjne bardzo trudno jest przestrzegać tej zasady.

Rozwiązania nierelacyjne nie zapewniają wszystkich tych cech. W większości przypadków silniki te nie umożliwiają wykonywania transakcji, złączeń oraz dostarczają tzw. ostateczną spójność (ang. *eventual consistency*), czyli spójność nie w każdym punkcie czasu. Wbrew pozorom cechy te są wystarczające w wielu przypadkach.

Bazy nierelacyjne wprowadzają dodatkowo pewne optymalizacje, które w niektórych przypadkach mogą być niedopuszczalne. Przykładowo nie zapisują informacji synchronicznie na dysk zaraz po zakończeniu operacji (*fsync*), nie obsługują transakcji (nawet lokalnych) a ze względu na brak relacji te same dane mogą się powtarzać (redudancja).

NoSQL to ruch promujący nierelacyjne magazyny danych. Słowo magazyny danych zostało tu użyte celowo, gdyż część rozwiązań to bardzo proste aplikacje dostarczające jedynie podstawowe funkcjonalności znane ze współczesnych baz danych. W ciągu ostatnich kilku lat pojawiło się bardzo wiele tego typu rozwiązań i wszystko wskazuje na to, że bazy relacyjne w końcu doczekały się konkurencji. Obecnie na rynku nierelacyjnych baz danych wykorzystywanych w wysokobociążonych systemach wyróżnić można 2 typy silników – magazyny klucz-wartość oraz bazy danych dokumentów.

Magazyny klucz-wartość to bardzo proste mechanizmy pozwalające wykonywać tylko proste operacje CRUD. Programista może jedynie odczytać, zapisać, zmodyfikować lub usunąć wartość dostarczając klucz główny. Jednym z tego typu rozwiązań jest Project Voldemort rozwijany przez programistów LinkedIn, który obsługuje replikację asynchroniczną, failover oraz waliduje dane przed umieszczeniem ich w bazie wg zgodności z ustalonym schematem.

Bazy danych dokumentów to znacznie bardziej zaawansowane silniki, które oprócz podstawowych operacji CRUD dostarczają także możliwość tworzenia indeksów oraz wykonywania złożonych zapytań, także z wykorzystaniem algorytmów Map/Reduce. Jedną z najpopularniejszych baz danych dokumentów jest MongoDB, która pozwala na tworzenie skomplikowanych zapytań, indeksowanie właściwości dokumentów (odpowiednik kolumn z tabel relacyjnych) a w niedalekiej przyszłości ma wspierać również sharding.

Magazyny nierelacyjne pod względem możliwości nie dorównują bazom relacyjnym, jednak ze względu na rezygnację z wielu cech baz relacyjnych (np. z transakcji, złączeń, synchronicznego zapisu na dysk) są znacznie wydajniejsze i bardziej skalowalne. Aby jednak w pełni wykorzystać drzemiącą w nich moc należy zmienić sposób myślenia. Przykładowo, wykorzystując bazy dokumentów należy posługiwać się dużymi drzewiastymi strukturami zamiast płaskimi wierszami znanymi z baz relacyjnych.

2.5. Podsumowanie

Istnieje wiele sposobów optymalizacji wydajności systemów bazodanowych. Każdy z nich ma zarówno wady jak i zalety. Często istnieje potrzeba połączenia wielu technik aby uzyskać zamierzony efekt. Sharding będący tematem tej pracy jest metodą przeznaczoną dla aplikacji o bardzo dużym obciążeniu, które uruchamiane są na kilkunastu a nawet kilkudziesięciu serwerach baz danych. Opracowywane rozwiązanie będzie jednak pozbawione wyżej wymienionych wad implementacji shardingu. W przeciwieństwie do Hibernate Shards zapytania SELECT będą parsowane i analizowane, co umożliwi wybór podzbioru serwerów, na których należy uruchomić dane zapytanie. Znacząco zmniejszy to obciążenie serwerów. Ponadto programista będzie mógł wykorzystać dowolną technologię, która korzysta ze standardu JDBC np. Hibernate, IBatis, Spring JDBC. JShards będzie także rozwiązaniem bardziej przezroczystym dla programisty aniżeli PL/Proxy. Programista nie będzie zmuszany

do używania dedykowanego języka programowania, przetwarzania wyników pochodzących z wielu shardów i będzie mógł wykorzystać dowolną bazę danych obsługującą język SQL.

3. Opis narzędzi wykorzystywanych w pracy

Kolejne podrozdziały opisują narzędzia wykorzystane podczas implementacji prototypu sterownika JShards. Mowa jest tutaj o rozproszonych relacyjnych bazach danych, standardzie JDBC, języku SQL i jego dialektach oraz bibliotekach dostarczonych przez firmy trzecie, które wykorzystuje sterownik.

3.1. Rozproszone relacyjne bazy danych

Relacyjne bazy danych to obecnie najchętniej wykorzystywane silniki baz danych. Zyskały sobie ogromną popularność ze względu na szeroki wachlarz możliwości. Umożliwiają wykonywania złożonych zapytań, można je optymalizować, zapewniają równoległy dostęp, a operacje mogą być wykonywane w transakcjach.

Zasada działania relacyjnych baz danych jest prosta. Baza danych składa się z relacji (tabel), w których przechowywane są dane w postaci wierszy. W skład każdej tabeli wchodzi określona liczba kolumn. Dla każdej kolumny istnieje komórka wiersza. Taki podział sprzyja eliminacji redundancji danych. W jednej tabeli przechowywane są wiersze dotyczące jednego typu. Dostęp do danych opiera się o algebrę relacji. Wykorzystywane są operatory rzutowania, selekcji, złączenia, sumy, różnicy oraz produktu kartezjańskiego.

Jedną z ważniejszych cech relacyjnych baz danych jest e. Skrót ten oznacza, że transakcje w bazie danych są atomowe (A - atomic), baza danych jest zawsze spójna (C – consistency), transakcje wykonywane współbieżnie są od siebie odizolowane (I – isolation), a w przypadku nagłej awarii serwera wykonane transakcje nie są tracone (D – durable).

Rozproszona baza danych to zbiór serwerów baz danych, które stanowią jedną całość. Dla użytkownika aplikacji nie ma znaczenia czy ma do czynienia z pojedynczym serwerem czy z rozproszoną bazą danych. Z logicznego punktu widzenia są to te same bazy.

Rozproszone bazy danych stosuje się głównie z następujących powodów: równoważenia obciążenia (ang. *load balancing*) oraz zapewnieniu wysokiej dostępności (ang. *high availability*).

Jeden fizyczny serwer bazy danych może nie być w stanie obsłużyć wszystkich nadchodzących żądań. W takiej sytuacji jednym z rozwiązań jest dokupienie kolejnych serwerów i zrównoważenie obciążenia poprzez rozdział żądań między serwery. Innym powodem stosowania rozproszonych baz danych jest zapewnienie wysokiej dostępności – gdy jeden serwerów przestanie działać to inny przejmie jego obowiązki (*failover*).

Wykorzystywanie rozproszonych baz danych może być przede wszystkim bardziej korzystne ze względów ekonomicznych, jednak wpływa też na zwiększenie stopnia skomplikowania systemu. Potrzebna jest większa liczba osób w dziale operacyjnym (np. administratorów baz danych) a bez odpowiedniej automatyzacji zarządzania utrzymanie dużej ilości baz danych graniczy z cudem.

3.1. Standard JDBC[11]

JDBC (ang. *Java DataBase Connectivity*) to interfejs programowania API dla języka Java pozwalający na wykonywanie zapytań SQL na dowolnych implementacjach baz danych (nie tylko relacyjnych). Dzięki uwspólnionemu API programista nie musi poznawać interfejsu każdej bazy danych z osobna a migracja z jednego silnika baz danych do innego jest mniej bolesna. Dodatkowo JDBC może być bazą dla wielu narzędzi operujących na wyższym poziomie abstrakcji – np. dla narzędzi mapowań obiektowo-relacyjnych (Hibernate) czy narzędzi ułatwiających posługiwanie się JDBC (IBatis, Spring JDBC). Wykorzystując więc Hibernate czy IBatis można korzystać z dowolnego silnika bazy SQL.

Interfejs programowania JDBC API implementowany jest przez tzw. sterowniki baz danych. Każdy producent baz danych dostarcza takie implementacje w postaci pliku (plików) jar. Aby skorzystać z konkretnej bazy danych wystarczy więc dodać sterownik tej bazy na ścieżkę klas (*classpath*) a następnie uruchomić program. Za pomocą JDBC programista łączy się z bazą danych, przesyła do niej zapytania i przetwarza zwrócone wyniki. Najważniejsze interfejsy JDBC to *Connection, Statement, PreparedStatement* oraz *ResultSet*. Obiekt *Connection* reprezentuje połączenie do bazy danych, obiekty *Statement* oraz *PreparedStatement* są

[11] Więcej na stronie http://java.sun.com/products/jdbc/overview.html

wykorzystywane do uruchamiania kodu SQL. *Statement* służy do uruchamiania zapytań ad-hoc, *PreparedStatement* do wielokrotnego uruchamiania tych samych zapytań z różnymi wartościami parametrów. Obiekt *ResultSet* przechowuje wyniki zapytań. Zwracany jest do aplikacji po uruchomieniu zapytania i dostarcza metod pozwalających na iterowanie po jego wynikach.

JDBC to API wbudowane w Java Platform Standard Edition (Java SE). W wersji 1.6 dostępna jest wersja 4 standardu JDBC. Sterownik tworzony w ramach tej pracy magisterskiej będzie implementował właśnie tą wersję specyfikacji.

Standard JDBC powstał w 1996 roku i od tego czasu wprowadzono w nim wiele zmian. Dodano obsługę tzw. punktów zachowań (ang. *savepoint*), automatycznie generowanych kluczy oraz nowych typów danych (takich jak BLOB, CLOB, XML, tablic, ROWID). To wszystko sprawiło, że API znacznie się rozrosło i wymaga od producentów sterowników poświęcenia dużej ilości czasu na implementację. Sterowniki nawet najbardziej popularnych baz danych nie implementują wszystkich metod wersji czwartej. Przykładowo sterownik PostgreSQL nie implementuje większości nowych metod tej wersji.

3.2. Język SQL i problem wielu dialektów

SQL (ang. *Structured Query Language*) to obecnie najpopularniejszy i najczęściej wykorzystywany język zapytań stosowany w relacyjnych baza danych. Język ten jest językiem deklaratywnym, czyli takim, w którym programista definiuje co należy zrobić zamiast tego jak to należy zrobić. Stoi to w opozycji do programowania imperatywnego gdzie programista wskazuje konkretne algorytmy.

Zasadniczo wszystkie zapytania SQL dzielą się na 3 grupy: DML (ang. *Data Mainpulation Language*), DDL (ang. *Data Definition Language*) oraz DCL (ang. *Data Control Language*). DML to najczęściej wykorzystywane zapytania służące do pobierania (zapytania *SELECT*), umieszczania (*INSERT*), modyfikowania (*UPDATE*) oraz usuwania (*DELETE*) danych z bazy. Zadaniem sterownika tworzonego w ramach tej pracy magisterskiej będzie w głównej mierze przetwarzanie tego typu zapytań. Zapytania DDL służą do manipulacji strukturami bazodanowymi (np. tabelami, indeksami). Są to m.in. zapytania *CREATE*, *DROP* oraz

ALTER. Ostatnia grupa – DCL to zbiór zapytań służących do zarządzania uprawnieniami. Użytkownicy bazy danych mogą bowiem mieć dostęp tylko do wybranych obiektów baz danych. Najważniejsze zapytania DCL to: *GRANT*, *REVOKE* oraz *DENY*.

Obecnie na rynku dostępnych jest wiele implementacji baz danych wykorzystujących język SQL. Jest to jedna z przyczyn procesu standaryzacji tego języka. Takie organizacje jak ISO oraz ANSI od wielu lat wspierają standard SQL. Początkowe wersje specyfikacji (SQL86 oraz SQL89) były bardzo ogólne przez co zbiór wspólny dla wszystkich implementacji był bardzo wąski. Każda implementacja dostarczała własnych rozwiązań, w konsekwencji czego programy nie mogły być przenoszone pomiędzy silnikami baz danych. Wiedza zdobyta przy wykorzystaniu jednej implementacji nie przydawała się w innym projekcie dotyczącym innego silnika. Dopiero kolejne standardy SQL92 oraz SQL99 znacznie uściśliły specyfikację języka. Obecnie implementacje SQL są już do siebie bardziej zbliżone, jednak ciągle wiele baz danych dostarcza niestandardowych rozszerzeń, które skutecznie łamią kompatybilność. Dlatego też sterownik tworzony w ramach tej pracy magisterskiej będzie implementował jedynie pewną część wspólną najpopularniejszych dialektów języka SQL, a dokładnie podzbiór standardu SQL92.

3.3. Wykorzystane biblioteki

Poniższe sekcje omawiają zewnętrzne biblioteki, które wykorzystano podczas implementacji narzędzia JShards. Są to JSqlParser, SnakeYAML oraz Google Collections.

3.3.1 JSqlParser

Język SQL ma bardzo złożoną składnię, dlatego stworzenie własnego parsera wymagałoby sporych nakładów czasowych. Autorzy tej pracy postanowili więc wykorzystać gotowe narzędzie jakim jest JSqlParser.

Zadaniem JSqlParser jest przetworzenie wyrażenia SQL a następnie stworzenie hierarchii obiektów Java będących odzwierciedleniem tego wyrażenia. Hierarchia ta nazywana jest także drzewem składni abstrakcyjnej (ang. *abstract syntax tree*). Nawigowanie po tak utworzonej hierarchii odbywa się za pomocą obiektu wizytatora (ang. *visitor*) tworzonego przez programistę. Dzięki temu kod jest przejrzysty i zawiera ograniczoną ilość bloków

warunkowych. Przykładowy kod źródłowy programu, który wypisuje nazwę tabeli użytej w zapytaniu znajduje się na Listing 3.

```
public class PrintTableVisitor implements StatementVisitor,
SelectVisitor, FromItemVisitor {

    /** Tu zapisana będzie nazwa tabeli */
    private String tableName;

    public void visit(Select select) {
        select.getSelectBody().accept(this);
    }

    public void visit(PlainSelect plainSelect) {
        plainSelect.getFromItem().accept(this);
    }

    public void visit(Table table) {
        tableName = table.getName();
    }

    public static void main(String[] args) throws
JSQLParserException {
        String sql = "SELECT * FROM users";

        // stworzenie obiektu parsera i przetworzenie zapytania
        CCJSqlParserManager pm = new CCJSqlParserManager();
        Statement statement = pm.parse(new StringReader(sql));

        // tworzenie obiektu wizytatora
        PrintTableVisitor visitor = new PrintTableVisitor();
        statement.accept(visitor);

        System.out.println("Użyta tabela w zapytaniu to " +
visitor.tableName);
    }

    // niezaimplementowane metody zostały pominięte
```

```
}
```

Listing 3 Przykład użycia biblioteki JSqlParser

Po uruchomieniu programu na konsoli wyświetli się napis, jak na Listing 4.

```
Użyta tabela w zapytaniu to users
```

Listing 4

Więcej o wzorcu wizytatora można znaleźć w [5].

3.3.2 SnakeYAML

Sposób działania sterownika tworzonego w ramach tej pracy zależeć będzie od jego konfiguracji. Konfiguracja zawierać będzie informacje o lokalizacji serwerów bazodanowych oraz opis algorytmów wykorzystywanych w odczycie i zapisie danych. Plik konfiguracyjny będzie miał postać hierarchicznej struktury drzewiastej.

Rozpatrzono 2 rodzaje języków formalnych przeznaczonych do reprezentowania danych w strukturalizowany sposób: XML oraz YAML. Najważniejszym kryterium wyboru była czytelność.

XML (ang. *Extensible Markup Language*) to jeden z najpopularniejszych formatów wykorzystywanych w plikach konfiguracyjnych. Zaletą XML jest to, że jest zrozumiały dla ludzi ale także dla maszyny – większość dostępnych języków programowania potrafi przetwarzać takie pliki. Pliki XML można więc edytować w zwykłych edytorach tekstu, ale także w specyficznych edytorach graficznych dedykowanych wybranym formatom. Przykładowy plik konfiguracyjny w XML przedstawia Listing 5.

```
<connections>
  <connection>
    <name>shard1</name>
    <url>jdbc:postgresql://dev:5432/shards</url>
    <user>shards</user>
    <password>shards</password>
  </connection>
```

```
</connections>
```

Listing 5 Przykładowy plik konfiguracyjny w formacie XML

YAML (ang. *YAML Aint' Markup Language*) to alternatywa dla XML. Ze względu na znacznie mniejszą popularność nie stworzono dla niego tak wielkiej liczby parserów. Jednak główną zaletą YAML jest jego czytelność. Twórcy zastąpili znane z XML znaczniki otwierające i zamykające znakami nowej linii i wcięciami, co prezentuje Listing 6.

```
connections:
  -
    name: shard1
    url: "jdbc:postgresql://dev:5432/shards"
    user: shards
    password: shards
```

Listing 6 Przykładowy plik konfiguracyjny w formacie YAML

Taki dokument jest o wiele czytelniejszy dla człowieka, a zarazem może być przetwarzany przez programy. Jedyne o czym musi pamiętać osoba dokująca edycji to odpowiednia liczba spacji i znaków nowej linii.

Specyfikację YAML można znaleźć w [6]. W skrócie YAML reprezentuje dowolne struktury danych używając trzech typów gałęzi: sekwencji, słowników oraz skalarów. Sekwencje to uporządkowane zbiory elementów, słowniki to mapy klucz-wartość a skalary to dowolne dane prezentowane jako zbiór znaków. W powyższym przykładzie sekwencją jest *connections*, której elementy zaczynają się myślnikiem, słownikiem pierwszy element sekwencji zawierający pary klucz-wartość oddzielone dwukropkiem, a skalarami poszczególne atrybuty słownika – *name*, *url*, *user* oraz *password*.

Sterownik JDBC tworzony w ramach tej pracy będzie wykorzystywał gotowy parser YAML o nazwie SnakeYAML. Jest to obecnie najbardziej stabilna implementacja parsera dostępna dla języka Java. Zaletą tego narzędzia jest także jego ciągły i intensywny rozwój.

3.3.3 Google Collections

Praktycznie każdy program napisany w języku Java korzysta z klas kolekcji. The Collections Framework[12] będący składnikiem Java Runtime Environment zawiera większość implementacji klas kolekcji oraz narzędzi do manipulacji nimi. Jest on rozwijany od początku istnienia Javy, czyli od 1995 roku. Pomimo swojego sędziwego wieku i ciągłego rozwoju wciąż brakuje mu wielu przydatnych funkcji. Biblioteka Google Collections znakomicie uzupełnienia ten szkielet.

Biblioteka Google Collections jest inicjatywą firmy Google, jednak udostępniana jest na zasadach Open Source, więc każdy może uczestniczyć w jej tworzeniu. Biblioteka ta głównie dostarcza nowych implementacji klas kolekcji oraz nowych mechanizmów manipulacji kolekcjami. Przykładowo udostępnia ona klasę MultiMap, czyli implementację mapy mogącą zawierać wiele wartości dla jednego klucza (odpowiednik zwykłej mapy przypisującej listę wartości do każdego klucza). Google Collections zapożycza też wiele z języków funkcyjnych. Udostępnia programiście nowe mechanizmy manipulacji kolekcjami – filtrowanie, transformację i wyszukiwanie.

Dzięki zastosowaniu Google Collections tworzony kod sterownika będzie bardziej zwięzły i czytelny.

[12] http://java.sun.com/j2se/1.5.0/docs/guide/collections/

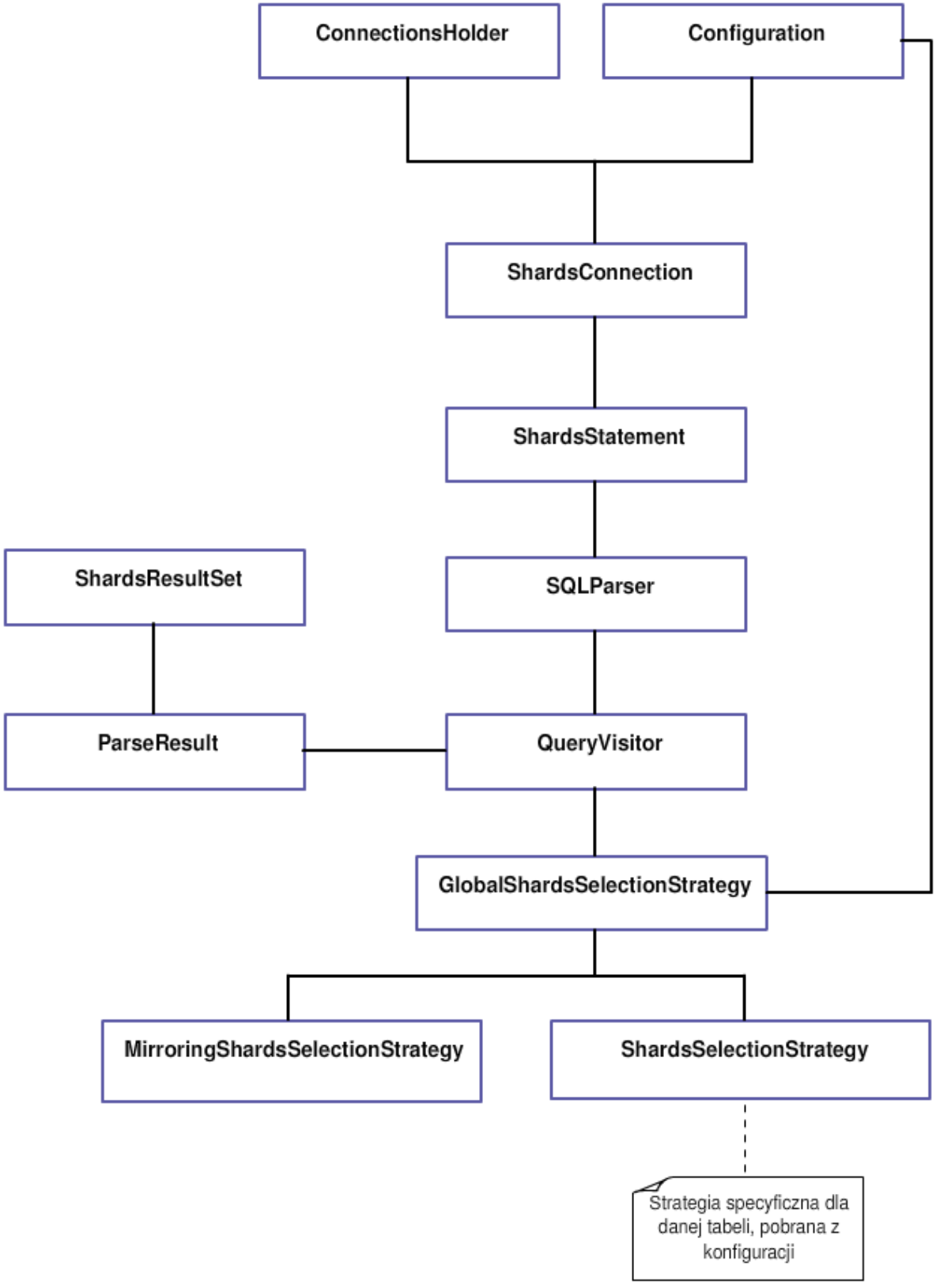
4. JShards – projekt rozwiązania

Koncepcja pracy zakłada stworzenie sterownika implementującego specyfikację JDBC. Zadaniem sterownika będzie umożliwienie poziomego podziału bazodanowej architektury monolitycznej na wiele rozproszonych fragmentów – shardów. Oczywistym zatem jest, że wyjątkowo istotnym elementem implementacji będzie strategia wyboru sharda.

Specyfika niektórych konstrukcji języka SQL nie jest bezpośrednio przystosowana do warunków środowiska rozproszonych i horyzontalnie skalowalnych baz danych. Dotyczy to na przykład funkcji agregujących lub klauzul LIMIT i OFFSET. Niezbędne stanie się zatem zbudowanie mechanizmu przepisywania zapytań (ang. rewriting).

Chęć zachowania wiernego podobieństwa do klasycznego sposobu korzystania ze standardu JDBC jest genezą konieczności ponownego przemyślenia wielu założeń tego standardu. Główny problem stanowi specyfika klasy ResultSet, a także wiele połączeń, które trzeba będzie obsłużyć.

Wszystkie te rozwiązania projektowe zostały omówione w niniejszym rozdziale. Uproszczony diagram klas, zawierający jedynie najistotniejsze z punktu widzenia projektu klasy, przedstawiono na Rysunek 8.



Rysunek 8 Uproszczony diagram klas

4.1. Stworzenie sterownika JDBC

Sterownik, który zostanie stworzony – klasa ShardsDriver – będzie implementacją interfejsu java.sql.Driver zgodną ze specyfikacją JDBC 4. Jego użycie w przeważającej części nie będzie różniło się od użycia dowolnego innego sterownika. Jedyna różnica ujawni się w chwili podawania adresu bazy danych do metody DriverManager.getConnection(). Adres URL, którego będzie oczekiwał sterownik będzie charakteryzowany przez prefiks „jdbc:shards:”. Następnym jego fragmentem będzie ścieżka do pliku konfiguracyjnego.

Dalsze korzystanie ze sterownika będzie przebiegało w sposób standardowy. Omówimy zatem szczegóły koncepcji tego rozwiązania.

Implementacja klasy ShardsDriver będzie korzystała z klasy ConfigurationLoader, która będzie odpowiedzialna za załadowanie nazwy klasy sterownika bazy danych, rezydującej na rozproszonych serwerach oraz listy połączeń do tych serwerów. Poszczególne połączenia zapisywane będą do instancji klasy ConnectionsHolder, dzięki czemu możliwe będzie późniejsze ich wykorzystanie.

Finalnie, metoda DriverManager.getConnection() zwróci obiekt klasy ShardsConnection implementujący interfejs Connection. Obiekt ten reprezentować będzie pojedyncze połączenie w realiach architektury shardów. Należy podkreślić, że faktycznie będzie to kontener, który przechowuje połączenia do wszystkich fizycznych baz danych. Dzięki takiemu podejściu, wszystkie żądania kierowane do obiektu-kontenera będą w rzeczywistości uruchamiane na połączeniach do wszystkich shardów. Dla przykładu wywołanie instrukcji z Listing 7, spowoduje przełączenie wszystkich baz składowych w tryb manualnego potwierdzania transakcji.

```
connection.setAutoCommit(false);
```

Listing 7

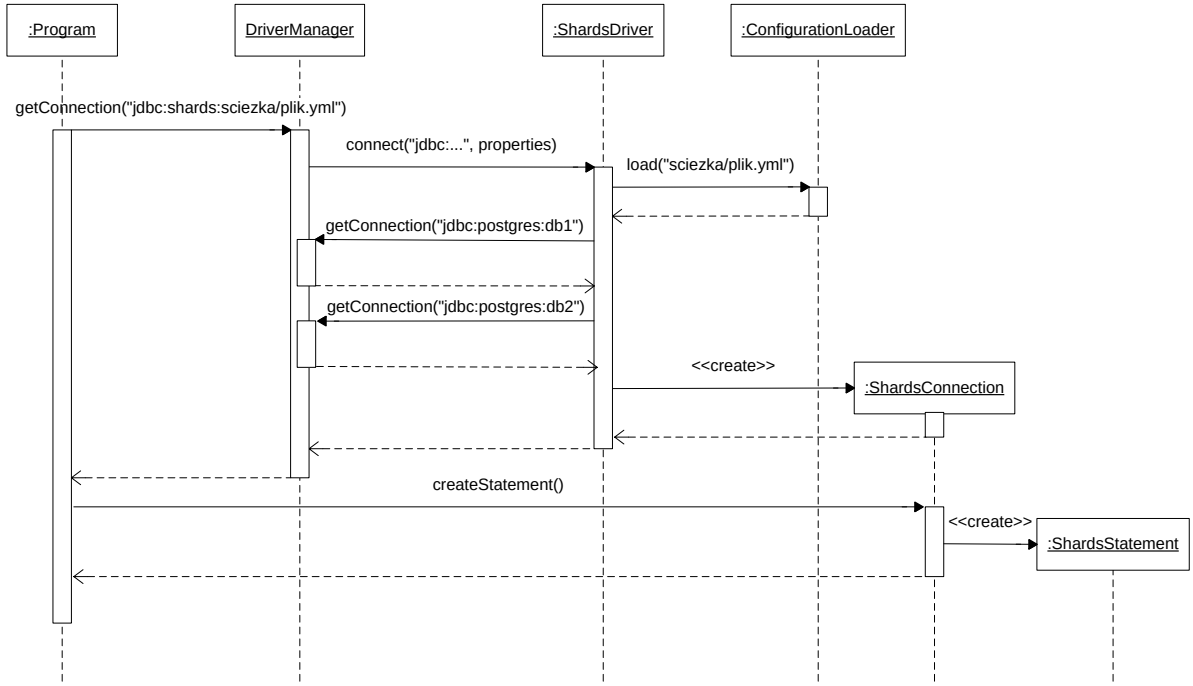
Od obiektu Connection można będzie zażądać zarówno obiektu klasy Statement, jak i PreparedStatement. W obu przypadkach będzie można stosować wszystkie znane techniki pracy z tymi interfejsami. W szczególności w PreparedStatement będzie można korzystać z

zapytań zawierających miejsca przeznaczone na dane, które będą później uzupełniane, co przedstawia Listing 8.

```
PreparedStatement                prepareStatement                =
connection.prepareStatement("SELECT count(id) FROM shards WHERE id
= ?");
prepareStatement.setLong(1, 1l);
ResultSet resultSet = prepareStatement.executeQuery();
```

Listing 8 Przykład użycia PreparedStatement

Opisany proces przedstawia Rysunek 9.



Rysunek 9 Diagram sekwenecji - tworzenie obiektu ShardsStatement

Zanim bliżej przyjrzymy się projektowi klas Statement i PreparedStatement, poznamy jedno z najważniejszych założeń przyjętych w sterowniku – koncepcję strategii wyboru sharda.

4.2. Koncepcja strategii wyboru sharda

Każde zapytanie SQL, zarówno typu DDL jak i DML, będzie podlegało analizie przez sterownik. Ze względów wydajnościowych zapytanie to będzie kierowane wyłącznie do

sharda, który jest istotny w danym kontekście. Kontekst ten z kolei będzie wyznaczany przez konstrukcję zapytania i strategię wyboru sharda.

Strategia wyboru sharda to klasa implementująca interfejs – ShardsSelectionStrategy. Interfejs ten zawiera metodę odpowiedzialną za dostarczenie zbioru identyfikatorów shardów, co w praktyce oznacza wyznaczenie podzbioru ze zbioru dostępnych baz danych.

Parametrem wejściowym metody jest obiekt klasy implementującej interfejs ParameterInfo. Klasa ta będzie przechowywać informacje istotne z punktu widzenia strategii. Są to m.in. nazwa kolumny, operator oraz wartość. Warto zaznaczyć, że w wyniku analizy każdego zapytania tworzonych jest tyle obiektów ParameterInfo, ile kolumn występuje w klauzuli WHERE tego zapytania – dla każdej kolumny po jednym obiekcie ParameterInfo.

Rozważmy zatem zapytanie przedstawione na Listing 9.

```
SELECT * FROM tabela WHERE id = 1
```

Listing 9

Analiza tego zapytania doprowadzi do powstania obiektu klasy ParameterInfo o atrybutach zaprezentowanych na Listing 10.

```
column = „id”
operator = Operator.EQUAL
value = 1
```

Listing 10 Atrybuty obiektu klasy ParameterInfo

Dzięki temu obiektowi strategia wyboru sharda może zdecydować, których shardów będzie dotyczyć dane zapytanie. Wartość atrybutu column obiektu ParameterInfo jest porównywana z kluczem strategii wyboru sharda. Klucz ten identyfikuje kolumnę w tabeli, która stanowi swoisty dyskryminator – wartość pola w tej kolumnie determinuje shard, na którym zostanie wykonane zapytanie.

To, do którego shardu faktycznie trafi zapytanie zależy od szczegółów implementacyjnych i konfiguracji strategii. Konfiguracja ta zostanie oparta o plik konfiguracyjny w formacie YAML. W pliku tym będzie można zdefiniować pary tabela bazodanowa – strategia wyboru

sharda. Tam też ustalać się będzie klucz strategii wyboru sharda – odpowiednią kolumnę ze wspomnianej tabeli. Dodatkowo plik ten będzie zawierał szczegółową konfigurację dla danej strategii, a także informacje o sposobie łączenia się z konkretnymi shardami.

Poniżej omówiono przykładowe strategie wyboru sharda.

4.2.1 Strategia oparta o przedziały

Rozważmy strategię opartą o przedziały – RangeShardsSelectionStrategy. Niezbędnym parametrem tej strategii jest nazwa kolumny w tabeli bazy danych, która będzie brana pod uwagę. Ponadto wymaga się zdefiniowania zakresów wartości i odpowiadających im identyfikatorów shardów. Konfiguracji tej można będzie dokonać bezpośrednio w pliku konfiguracyjnym, co zostanie omówione w rozdziale 5.

Wywołanie metody selectShards() na rzecz strategii RangeShardsSelectionStrategy, spowoduje porównanie nazwy kolumny zapisanej w strategii z nazwą pochodzącą z zapytania, a więc przechowywaną w obiekcie ParameterInfo. Wynik pozytywny spowoduje porównanie wartości pochodzącej z zapytania z dostępnymi przedziałami i zwrócenie odpowiednich shardów. Z kolei negatywny rezultat porównania będzie miał miejsce wtedy, gdy zapytanie lub jego fragment nie będzie dotyczyło określonej strategii. Stanie się tak na przykład wtedy, gdy rozważany parametr nie będzie parametrem klucza strategii wyboru sharda, ale będzie z nim połączony za pomocą operacji logicznych.

Przyjmując, że kluczem strategii wyboru sharda jest kolumna „id”, rozważmy zapytanie z Listing 11.

```
SELECT * FROM tabela WHERE id = 1 AND name = 'Marcin'
```

Listing 11

Zapytanie to spowoduje utworzenie dwóch obiektów ParameterInfo – dla kolumny „id” oraz „name”. Następnie każdy z tych obiektów zostanie przekazany do strategii wyboru sharda. Pierwszy spowoduje dodanie odpowiedniego sharda do zbioru wyjściowego, ponieważ przekazywana kolumna odpowiada kluczowi strategii. Drugi natomiast spowoduje zwrócenie zbioru wszystkich shardów, ponieważ kolumna „name” nie stanowi klucza strategii. Następnie na zbiorach tych wykonywana jest operacja teoriomnogościowa odpowiadająca

zastosowanej operacji logicznej. W przykładzie tworzony jest iloczyn zbiorów wynikowych, co finalnie skutkuje zastosowaniem zapytania jedynie na shardach należących do pierwszego zbioru.

Problemu nie będzie stanowiło także zapytanie z operacją sumy teoriomnogościowej, w szczególności gdy oba operandy kwalifikują się jako klucz strategii wyboru sharda, jak w zapytaniu na Listing 12.

```
SELECT * FROM tabela WHERE id = 1 OR id = 2
```

Listing 12

Wtedy zwrócone zostaną dwa podzbiory zbioru wszystkich shardów, a operacja sumy tych zbiorów wyłoni odpowiednie bazy danych.

Ponadto, ze względów wydajnościowych planuje się optymalizację funkcji zwracającej „wszystkie shardy". Zakłada się bowiem, że określona strategia nie musi wykorzystywać wszystkich shardów, tzn. wynikiem jej działania będzie podzbiór właściwy zbioru wszystkich shardów. W związku z tym funkcja ta nie będzie zwracać wszystkich shardów w sensie zdefiniowanych połączeń, a jedynie shardy uwzględnione w konfiguracji tej strategii.

4.2.2 Strategia oparta na funkcji haszującej

Kolejna warta uwagi strategia to strategia oparta na funkcji haszującej. Koncepcję tę realizuje klasa abstrakcyjna HashShardsSelectionStrategy. Jej przykładowa specjalizacja, ModHashShardsSelectionStrategy, realizuje operację wyznaczania odpowiedniego sharda na podstawie wyniku funkcji modulo.

Dla przykładu załóżmy, że w danym środowisku istnieją dwa shardy. Załóżmy także, że kluczem strategii wyboru sharda jest kolumna „id". Wykonajmy w tym środowisku zapytania zawarte na Listing 13.

```
(1) SELECT * FROM tabela WHERE id = 1
(2) SELECT * FROM tabela WHERE id = 2
(3) SELECT * FROM tabela WHERE id = 3
```

Listing 13

Wartość każdej kolumny z warunku zapytania zostanie wykorzystana jako argument funkcji modulo. Tutaj będzie to funkcja modulo 2, ponieważ w środowisku występują dwa shardy. W związku z tym, że zachodzi zależność przedstawiona na Listing 14, zapytania 1 i 3 zostaną uruchomione na tym samym shardzie.

```
1 mod 2 = 3 mod 2 = 1
```

Listing 14

Z kolei zapytanie 2 zostanie uruchomione na drugim shardzie, ponieważ zachodzi zależność przedstawiona na Listing 15.

```
2 mod 2 = 0
```

Listing 15

Jak widać, w tej strategii to wartość funkcji modulo determinuje wybór sharda.

4.2.3 Strategia globalna

Najważniejsza, bo wykorzystywana przy analizie każdego zapytania, będzie strategia globalna, reprezentowana przez klasę GlobalShardsSelectionStrategy. Jak każda strategia, także strategia globalna będzie implementowała interfejs ShardsSelectionStrategy. Każde zapytanie poddawane będzie początkowo analizie strategii globalnej. Polegać ona będzie na określeniu, czy tabela, której dotyczy zapytanie jest tabelą shardowaną, czy też globalną:

- Tabela shardowana – tabela bazodanowa, która jest podzielona horyzontalnie i której poszczególne części w sensie podzbioru zbioru wszystkich wierszy, umieszczone są na różnych rozproszonych bazach danych.
- Tabela globalna – tabela bazodanowa, której dokładna kopia (ang. mirror) umieszczona jest na każdej rozproszonej bazie danych. Tabela ta nie ma przypisanej żadnej strategii w pliku konfiguracyjnym.

Jeśli zapytanie dotyczyć będzie tabeli shardowanej, strategia specyficzna dla tej tabeli zostanie pobierana z konfiguracji. Następnie zapytanie o shardy, których należy użyć, oddelegowane będzie do tej specyficznej strategii.

Jeśli jednak zapytanie dotyczyć będzie tabeli globalnej, zostanie ono oddelegowane do specjalnej strategii odpowiedzialnej za mirroring – MirroringShardsSelectionStrategy. Strategia ta odpowiadać będzie za przekazanie zapytania do wszystkich shardów, a konkretnie:

- O ile zapytanie SQL ma charakter aktualizujący (INSERT, UPDATE, DELETE) lub jest to zapytanie typu DDL, jest ono przekazywane do wszystkich shardów.
- W przeciwnym wypadku, tzn. jeśli jest to zapytanie typu SELECT, zapytanie kierowane jest do jednego, losowo wybranego sharda.

Nic nie stoi na przeszkodzie, aby strategii mirroringu używać w klasyczny sposób – można ją skonfigurować do obsługi tabel w pliku konfiguracyjnym.

4.3. Przepisywanie zapytań SQL

Projekt zakłada konieczność przepisywania zapytań SQL (ang. rewriting), które kierowane są do sterownika JShards. Spowodowane jest to nieprzystosowaniem specyfiki języka SQL do warunków środowiska rozproszonego. Pierwszym przykładem są funkcje agregujące.

4.3.1 Funkcje agregujące

Rozważmy następujące zapytanie z Listing 16.

```
SELECT sum(value) FROM shards;
```

Listing 16

W klasycznej architekturze scentralizowanej zapytanie zwraca sumę wszystkich wartości zawartych w kolumnie „value” tabeli „shards”. W architekturze shardów oczekuje się identycznego rezultatu, jednak baza danych nie może przejąć całej odpowiedzialności, ponieważ wiersze, które powinny być brane pod uwagę w podsumowaniu mogą znajdować się na różnych fizycznych bazach. Z tego powodu aplikacja powinna przejąć odpowiedzialność bazy danych. Najprostszym rozwiązaniem byłoby pobranie do pamięci operacyjnej wszystkich rekordów z istotnych w danym kontekście baz a następnie dokonanie sumowania tych wartości. Ze względów wydajnościowych zdecydowano się jednak na nieco inne rozwiązanie. Otóż podsumowanie zostanie wykonane w każdej bazie z osobna, po czym

wyniki cząstkowe zostaną sprowadzone do pamięci i tam dopiero zsumowane. Dzięki temu unika się potencjalnych problemów związanych z koniecznością sprowadzenia do pamięci wielu tysięcy rekordów.

Analogicznemu procesowi podlegać będą funkcje:

- COUNT – cząstkowe zliczenia z fizycznych baz podlegają operacji sumowania w pamięci
- MIN, MAX – cząstkowe wyniki z fizycznych baz stanowią argumenty funkcji minimum/maksimum wykonywanej w pamięci.

Odrębnego traktowania wymaga funkcja obliczania średniej arytmetycznej – AVG. Jak wiadomo średnia arytmetyczna zbioru liczb nie jest równa wartości średniej ze średnich arytmetycznych podzbiorów zbioru wejściowego. Z tego powodu w przypadku funkcji AVG nie można zastosować rozumowania przedstawionego wcześniej. Można natomiast zaobserwować, iż do obliczenia całościowej średniej arytmetycznej wystarczy suma oraz ilość liczb, które podlegały sumowaniu na każdej z fizycznych baz. W związku z tym zdecydowano się zamieniać wywołania funkcji AVG na wywołania funkcji składowych – SUM i COUNT. Dzięki temu lokalnie w pamięci będzie można obliczyć sumę wszystkich elementów pochodzących z różnych baz, a następnie podzielić tę wartość przez sumaryczną liczbę elementów. Proces podmiany wywołania funkcji AVG na wywołanie odpowiadających jej funkcji SUM i COUNT przeprowadzany jest na każdym zapytaniu analizowanym przez sterownik.

4.3.2 Sortowanie wyników

Innym przypadkiem, w którym nieodzowny okazuje się rewriting są zapytania z sortowaniem. W środowisku shardów nie jest możliwe pozostawienie odpowiedzialności za sortowanie po stronie bazy danych. Rekordy sprowadzone z poszczególnych fizycznych baz mogą mieć inny porządek, niż gdyby zostały posortowane w jednej bazie. Z tego powodu konieczne okazało się wykonywanie sortowania w pamięci operacyjnej.

Ta decyzja pociąga za sobą pewne konsekwencje. Otóż jakkolwiek w środowisku scentralizowanym kolumny, które znajdują się na liście klauzuli ORDER BY nie muszą

jednocześnie znajdować się na liście klauzuli SELECT, tak w środowisku rozproszonym musi zachodzić ten warunek. Dzieje się tak ponieważ aplikacja musi mieć możliwość porównywania wartości z kolumn sortowania. Dlatego też założono, że zapytania z klauzulą ORDER BY muszą zostać przepisane – wszystkie kolumny z listy klauzuli ORDER BY zostają dodane do listy klauzuli SELECT. Przykładowe zapytanie z Listing 17 zostanie przepisane do postaci zamieszczonej na Listing 18.

```
SELECT id FROM shards ORDER BY value
```

Listing 17

```
SELECT id, value FROM shards ORDER BY value
```

Listing 18

Dodatkowo przyjęto założenie, że kolumny, po których będzie przebiegało sortowanie muszą implementować interfejs Comparable. Jest to niezbędny krok, który ma na celu umożliwienie wykorzystania natywnych dla języka Java mechanizmów sortowania.

Problem porządkowania jest szczególnie istotny, jeśli idzie w parze z klauzulami LIMIT i OFFSET.

4.3.3 Klauzule LIMIT i OFFSET

Jak wspomniano, w zapytaniach z sortowaniem i klauzulami LIMIT i/lub OFFSET nieodzowny okazuje się rewriting. Przeanalizujmy to na przykładzie. Przyjmijmy, że kluczem strategii wyboru sharda jest kolumna „id”, a zawartość naszych baz przedstawia Listing 19.

```
Shard 1 – rekordy o identyfikatorach 1 i 2
Shard 2 – rekordy o identyfikatorach 3, 4, 5 oraz 6
```

Listing 19

W tym środowisku rozważmy zapytanie z Listing 20.

```
SELECT id FROM shards ORDER BY id DESC LIMIT 3 OFFSET 2
```

Listing 20

W klasycznym środowisku scentralizowanym, oczekiwany wynik stanowiłyby rekordy o identyfikatorach równych 4, 3 i 2. Jednak gdyby zdecydowano się przekazać klauzule LIMIT i OFFSET do poszczególnych shardów, wynik byłby błędny i wynosiłby:

- Rekordy o identyfikatorach 4 i 3, o ile przekazano by obie klauzule
- Rekordy o identyfikatorach 4, 2 i 1, o ile przekazano by wyłącznie klauzulę LIMIT, a przesunięcia (OFFSET) dokonano by lokalnie

Rozwiązaniem, które nasuwa się bezpośrednio jest wykonanie obu operacji – przesunięcia i ograniczenia – lokalnie. Niestety wiązałoby się to z poważnymi ograniczeniami wydajności, ponieważ konieczne byłoby sprowadzenie z baz danych wielu rekordów, które ostatecznie i tak zostałyby pominięte. Wniosek ten był bezpośrednią przyczyną wprowadzenia następującej optymalizacji.

OFFSET, o ile występuje w zapytaniu, zawsze jest z niego usuwany i wykonywany lokalnie. Jednakże, jeśli klauzula ta występuje w towarzystwie klauzuli LIMIT, ograniczenie nie jest usuwane z zapytania, natomiast wartość ograniczenia jest powiększana o wartość przesunięcia. Dla przykładu, powyższe zapytanie będzie przepisane do postaci z Listing 21.

```
SELECT id FROM shards ORDER BY id DESC LIMIT 5
```

Listing 21

Zapytanie takie trafia do wszystkich shardów. Dzięki pozostawieniu klauzuli LIMIT, straty wydajności nie są tak duże. Przesunięcie i ograniczenie nastąpi lokalnie, w pamięci operacyjnej, wykorzystując oczywiście parametry z pierwotnego zapytania.

Ostatnim elementem, który będzie interesujący z punktu widzenia przepisywana zapytań jest tandem klauzul GROUP BY i HAVING.

4.3.4 Klauzule GROUP BY i HAVING

Analogicznie, jak w przypadku sortowania, tak i grupowanie wymaga konieczności przepisania kolumn z klauzuli GROUP BY do klauzuli SELECT. Grupowanie nie może przebiegać po stronie pojedynczych baz, ponieważ istnieje ryzyko, że elementy tej samej grupy mogą znaleźć się na różnych shardach, co doprowadziłoby do przekłamań w wynikach

zapytania. Dlatego też operację grupowania postanowiono realizować w pamięci operacyjnej. Aby tego dokonać, należy pobrać wartości kolumn sortowania z poszczególnych shardów, co z kolei implikuje konieczność przepisania kolumn zawartych na liście klauzuli GROUP BY na listę klauzuli SELECT.

Ze względów wydajnościowych zdecydowano, że klauzula GROUP BY zostanie przekazana do każdego istotnego sharda. Dzięki temu poszczególne bazy danych dokonują wstępnego grupowania, do aplikacji przesyłane są o wiele mniejsze porcje rekordów, a lokalnie dokonuje się jedynie złączenia poszczególnych grup.

Z klauzulą GROUP BY nieodłącznie wiąże się klauzula HAVING. Podobnie, jak poprzednio, kolumny lub funkcje z listy tej klauzuli będą musiały trafić na listę klauzuli SELECT. Po zgrupowaniu rezultatów, wyniki tego grupowania zostaną przefiltrowane w celu odrzucenia tych, które nie spełnią warunków HAVING.

Ponadto klauzula HAVING nie będzie mogła pozostać w zapytaniu kierowanym do shardów. Należy ją usunąć, ponieważ grupa, która nie spełnia warunków na pojedynczym shardzie, może spełniać te same warunki po połączeniu wyników ze wszystkich shardów. Przeanalizujmy ten przypadek na przykładzie środowiska z Listing 22.

```
Shard 1 – rekordy: [id: 1, value: 'a'], [id: 2, value: 'a'], [id:
3, value: 'b']
Shard 2 – rekordy: [id: 4, value: 'a'], [id: 5, value: 'b'], [id:
6, value: 'c']
```

Listing 22

Oczekiwany wynik zapytania z Listing 23 zawiera Listing 24.

```
SELECT value, count(id) FROM shards GROUP BY value HAVING
count(id) > 2
```

Listing 23

```
 [value: 'a', count(id): 3] – zaakceptowany
[value: 'b', count(id): 2] – odrzucony
```

```
[value: 'c', count(id): 1] – odrzucony
```

Listing 24

Tymczasem, przekazanie klauzuli HAVING do shardów skutkowałoby zwróceniem zbioru pustego, ponieważ zapytanie to zwróci na poszczególnych shardach wyniki przedstawione na Listing 25.

```
Shard 1: [value: 'a', count(id): 2] – odrzucony
             [value: 'b', count(id): 1] – odrzucony
Shard 2: [value: 'a', count(id): 1] – odrzucony
             [value: 'b', count(id): 1] – odrzucony
             [value: 'c', count(id): 1] – odrzucony
```

Listing 25

Z tego powodu wspomniane zapytanie zostanie ostatecznie przepisane do postaci pokazanej na Listing 26.

```
SELECT count(id), value, count(id) FROM shards GROUP BY value
```

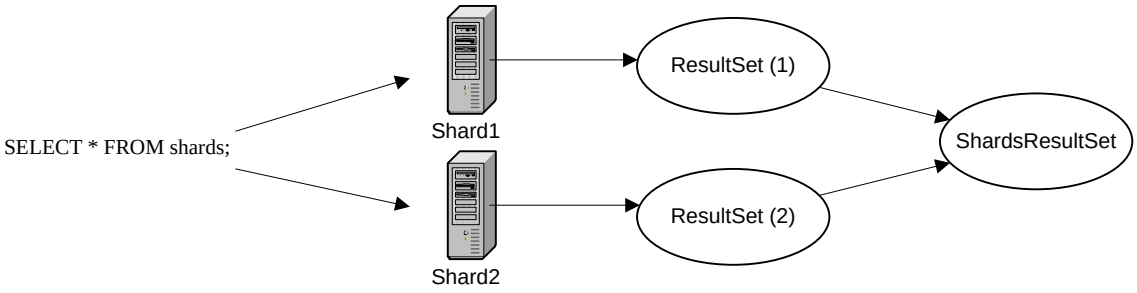
Listing 26

Jako że przeanalizowano już przypadki dostosowania zapytań SQL w środowisku rozproszonym, warto teraz przeanalizować sposoby postępowania z wynikami tych zapytań.

4.4. Mechanizm analizy i przetwarzania danych napływających z rozproszonych baz danych

Podstawowym problemem wprowadzenia przezroczystości obsługi środowiska shardów okazuje się łączenie i przetwarzanie zapytań pochodzących z różnych fizycznych baz danych.

Zapytanie kierowane do bazy zwraca obiekt implementujący interfejs ResultSet – inny dla każdego sharda, do którego trafiło zapytanie. Aby zachować transparentność, programista korzystający ze sterownika JShards powinien otrzymać tylko jeden obiekt ResultSet będący produktem złączenia obiektów częściowych. Koncepcję tę przedstawia Rysunek 10. Z tego powodu planuje się stworzyć własną implementację interfejsu ResultSet, do której będzie można łatwo dodawać zawartość klasycznych implementacji tego interfejsu.



Rysunek 10 Łączenie obiektów ResultSet pochodzących z odrębnych shardów

Celem jest możliwość zainicjowania obiektu agregującego kolekcją obiektów ResultSet z poszczególnych shardów. Właściwość taka zapewni klasa ShardsResultSet (Listing 27).

```
ResultSet resultSet = new ShardsResultSet(List<ResultSet>
resultSets);
```

Listing 27 Tworzenie obiektu klasy ShardsResultSet

Kolejnym etapem przetwarzania danych napływających z wielu shardów będzie wykonanie wszystkich operacji, o których mowa w rozdziale 4.3, tzn.:

- Grupowanie
- Obliczanie wartości średniej arytmetycznej
- Filtrowanie ze względu na klauzulę HAVING
- Sortowanie
- Obsługa klauzul LIMIT i/lub OFFSET

Przed zwróceniem obiektu klasy ShardsResultSet istotne będzie jeszcze usunięcie wszystkich tych kolumn, które zostały dodane do zapytania ze względu na konieczność lokalnego wykonania operacji wyszczególnionych wyżej.

Ostatnim problemem projektowym, któremu należało stawić czoła była obsługa zapytań predefiniowanych – PreparedStatement. Jest to o tyle istotne, że wiele popularnych narzędzi i bibliotek, np. Hibernate [7], przygotowuje zapytania w tej formie. Zaimplementowanie ich obsługi w sterowniku JShards stało się zatem koniecznością. W ten sposób możliwe będzie wykorzystanie sterownika w licznych przypadkach użycia. Dzięki zastosowaniu polimorfizmu i dziedziczenia, możliwe okazało się ponowne użycie kodu klasy

ShardsStatement w klasie ShardsPreparedStatement. Szczegóły techniczne tego rozwiązania omówiono w rozdziale dotyczącym implementacji.

Skoro koncepcja projektu została już przedstawiona, warto przejść do szczegółów implementacyjnych.

5. Implementacja prototypu

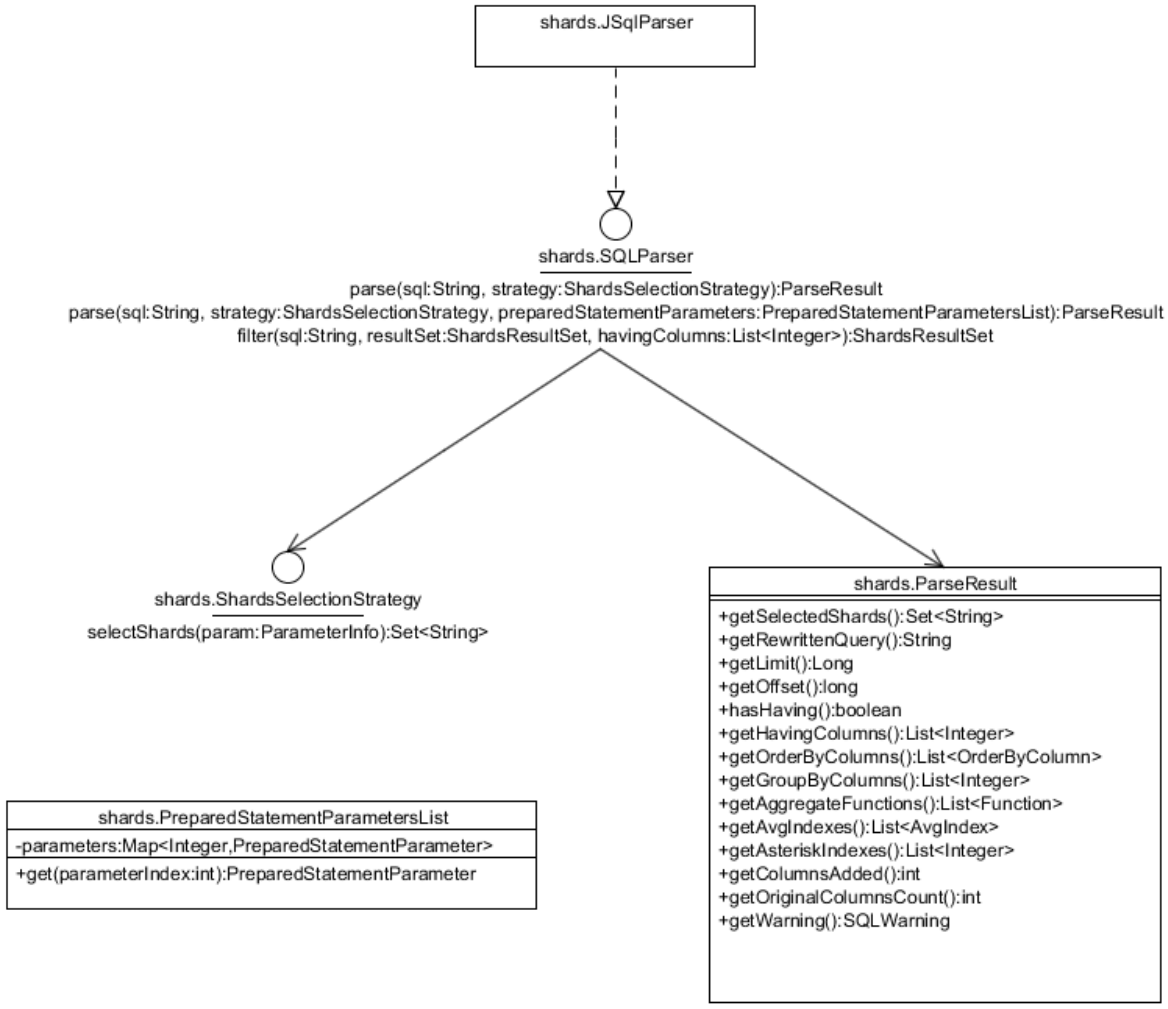
Rozdział ten ma celu opisanie w jaki sposób zaimplementowany został prototyp sterownika JDBC. W szczególności omówiony będzie tu parser zapytań SQL oraz klasa ShardsStatement, dzięki której możliwe jest wykonywanie zapytań w środowisku rozproszonym. Rozdział czwarty opisywał ramowy projekt sterownika, ten rozdział opisuje szczegóły implementacyjne.

5.1. SQLParser

Najważniejszym elementem sterownika jest parser zapytań SQL. Parser odpowiedzialny jest za przetworzenie zapytania i zwrócenie zbioru shardów, na których zapytanie powinno zostać uruchomione. W razie potrzeby parser może dodatkowo przepisywać zapytanie (ang. *rewrite*), tak aby mogło być uruchomione w środowisku shardów.

Interfejs parsera to *shards.SQLParser*. Obecnie dostępna jest tylko jedna implementacja tego interfejsu o nazwie *shards.JSqlParser*. Wykorzystuje ona bibliotekę JSqlParser opisaną w rozdziale 3.3.1. Diagram klas zawierający parser oraz klas przez niego wykorzystywanych przedstawia Rysunek 11. Interfejs *shards.SQLParser* definiuje trzy metody. Dwie metody o nazwie *parse* służą do przetwarzania zapytań i zwracania listy shardów, na których uruchomione ma zostać zapytanie. Pierwsza dwuargumentowa metoda używana jest przy wykorzystaniu zwykłych wyrażeń *java.sql.Statement*, druga do wyrażeń sparametryzowanych *java.sql.PreparedStatement*. Specjalny obiekt klasy *shards.PreparedStatementParametersList* hermetyzuje parametry przekazane do zapytania. Obie metody zwracają obiekt klasy *shards.ParseResult* przechowujący wynik działania metody – wybrane nazwy shardów, przepisane zapytanie oraz inne dodatkowe informacje powstałe podczas przetwarzania zapytania SQL.

Trzecia metoda interfejsu *shards.SQLParser* o nazwie *filter* jest uruchamiana dopiero po otrzymaniu i złączeniu wyników z poszczególnych shardów. Wykorzystywana jest tylko dla zapytań zawierających klauzulę HAVING i jej zadaniem jest filtrowanie złączonych wyników wg kryteriów podanych w tej klauzuli.



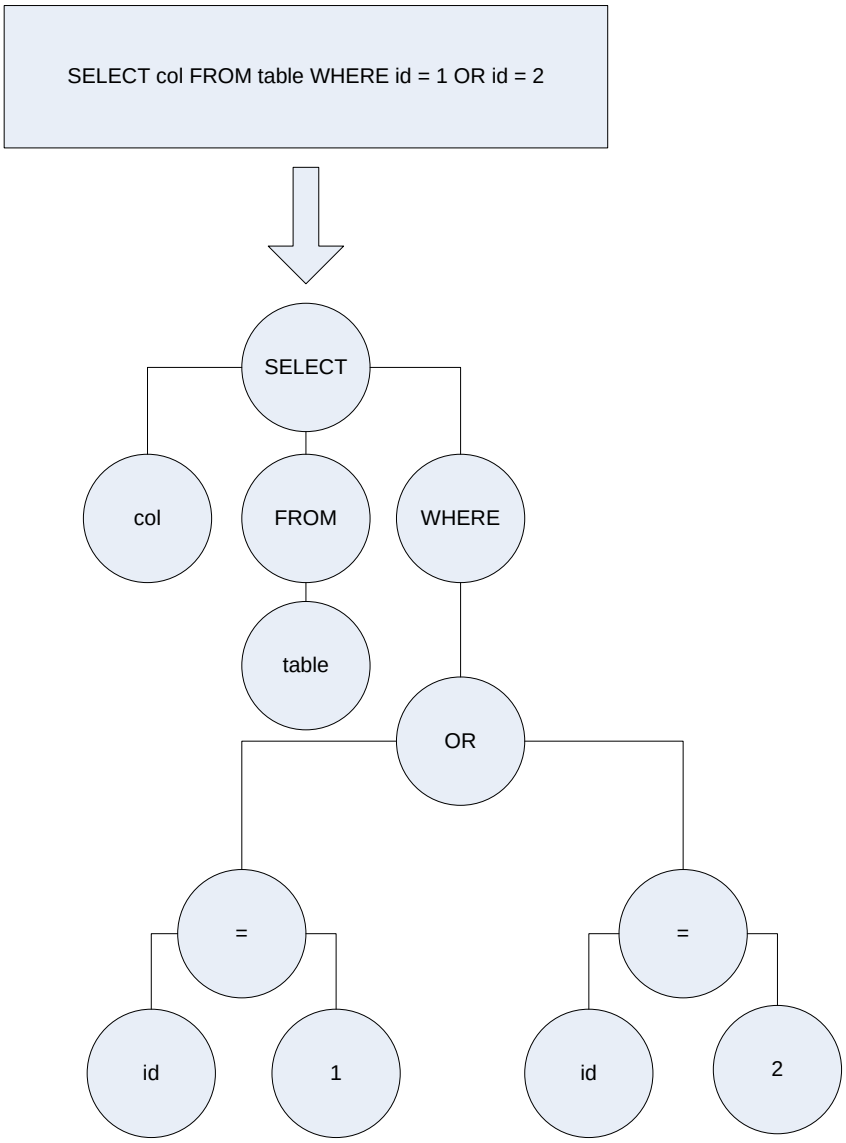
Rysunek 11 Diagram klas interfejsu parsera

Sercem *shards.JSqlParser* jest obiekt wizytatora. Więcej o tym wzorcu projektowym można przeczytać w [5]. Przykład kodu źródłowego znalazł się w rozdziale 3.3.1. *JSqlParser* wykorzystuje 2 obiekty wizytatorów – główny o nazwie *shards.QueryVisitor* wykorzystywany podczas przetwarzania zapytań SQL oraz *shards.HavingExpressionVisitor* używany do filtrowania wyników zapytań z klauzulą *having*. *QueryVisitor* wykorzystywany jest przez metody parse, natomiast *HavingExpressionVisitor* przez metodę filter.

5.1.1 QueryVisitor

Podczas przetwarzania zapytania SQL wykorzystywany jest obiekt klasy *shards.QueryVisitor*. Obiekt ten przechowuje między innymi **stos** wybranych **shardów** oraz

stos rezultatów wykorzystywany w obliczeniach. Na podstawie zapytania tworzony jest obiekt Java będący drzewem syntaktycznym. Rysunek 12 Przykładowe drzewo syntaktyczne w znacznym uproszczeniu przedstawia sposób podziału zapytania na składowe. Parser porusza się po takim obiekcie modyfikując oba stosy.

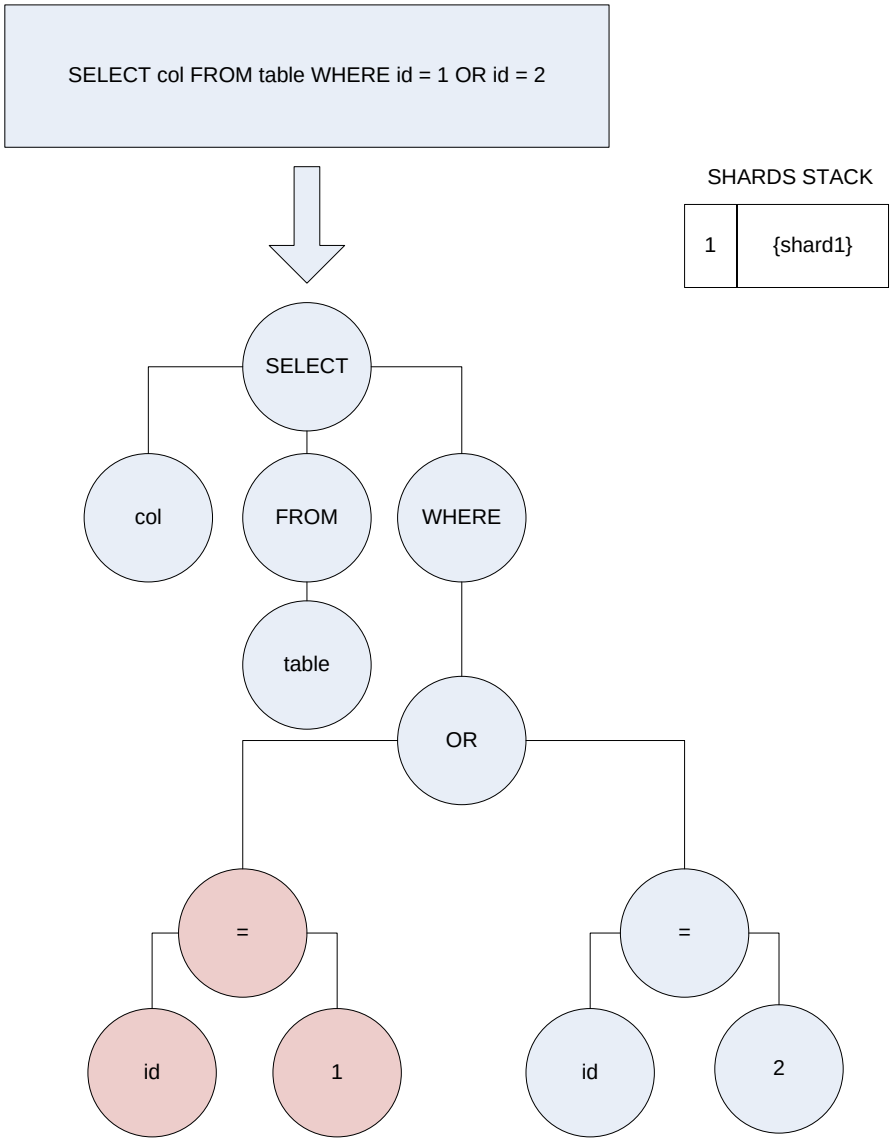


Rysunek 12 Przykładowe drzewo syntaktyczne

5.1.1.1 Stos shardów

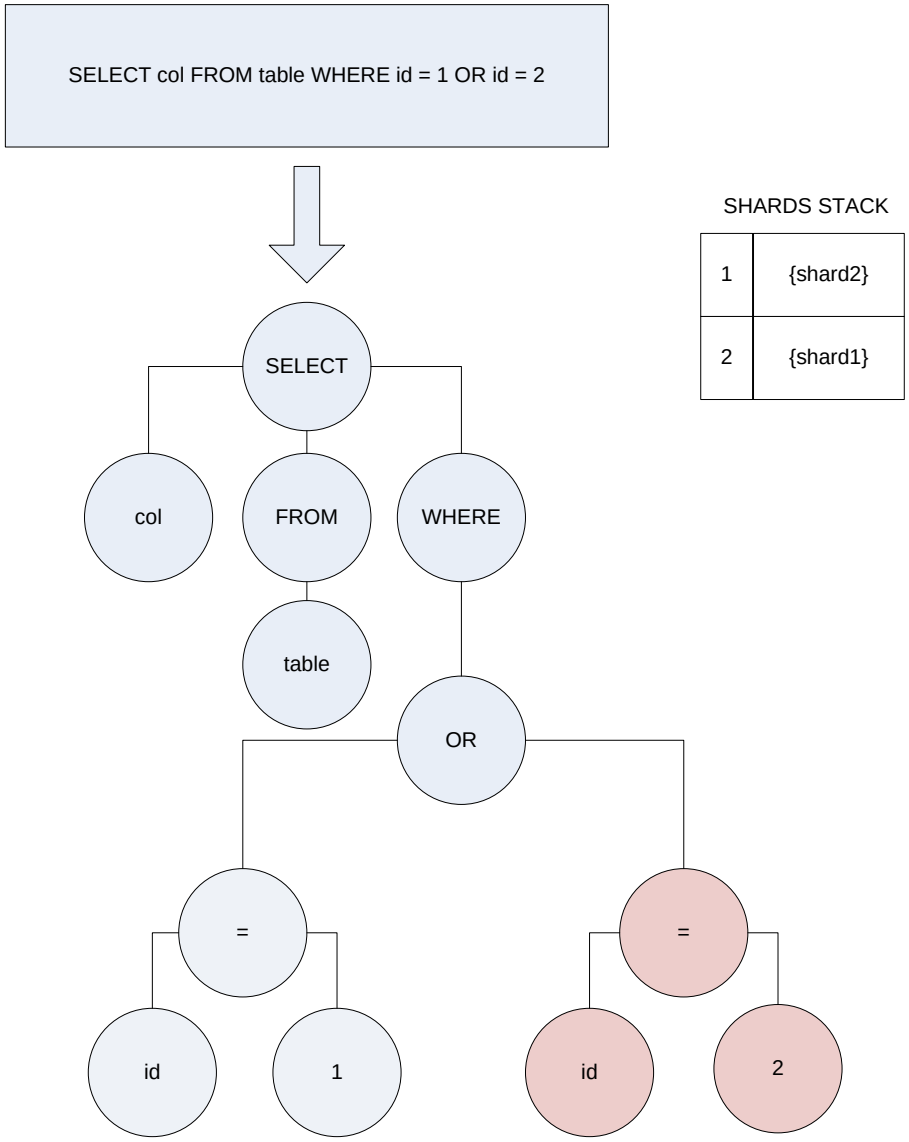
Na stosie shardów przechowywane są elementy będące zbiorami wybranych shardów. Podczas przetwarzania drzewa stos jest modyfikowany. Na podstawie wszelkich wyrażeń porównujących tj. „=”, „>”, „<”, „>=”, „<=”, „like”, „<>” znalezionych w klauzuli WHERE

wybierane są shardy. Parser deleguje takie pojedyncze wyrażenia do obiektu strategii, która decyduje, na których shardach zapytanie powinno się uruchomić. Wynikowy zbiór shardów w postaci pojedynczego elementu dodawany jest do stosu shardów. Rysunek 13 przedstawia tę operację. Wyrażenie id = 1 jest przekazywane do obiektu strategii, która zwraca jednoelementowy zbiór składający się z shard1.



Rysunek 13

Parser przechodzi do kolejnego wyrażenia. Rysunek 14 przedstawia tą operację. Wyrażenie id = 2 przekazywane jest do strategii, która zwraca jednoelementowy zbiór składający się z shard2. Zbiór ten podobnie jak w poprzedniej operacji dodawany jest na szczyt stosu.

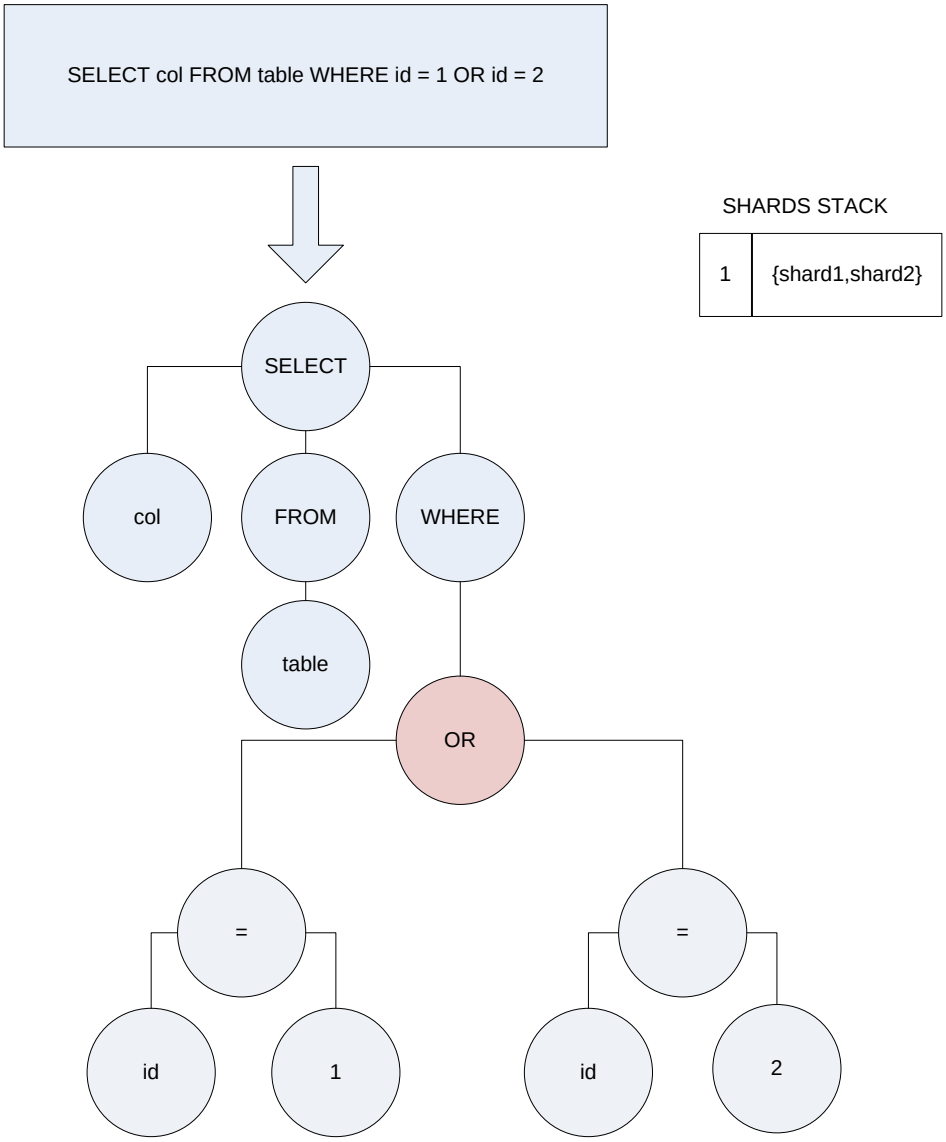


Rysunek 14

Parser posiada obsługę wyrażeń logicznych OR, AND, NOT itp. W powyższym przykładzie w klauzuli WHERE zostało użyte wyrażenie OR. Gdy parser natrafi na takie wyrażenie to ściąga ze stosu shardów dwa elementy. W powyższym przykładzie ściągnie {shard2} oraz {shard1}. Następnie wykonywana jest logiczna suma na zbiorach:

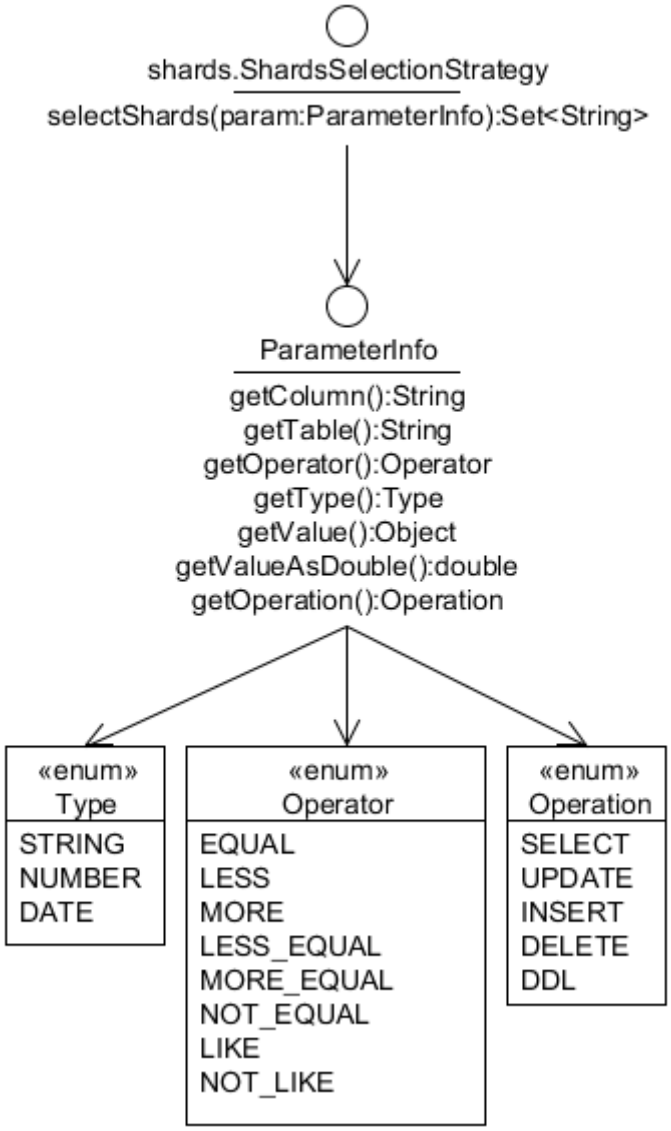
```
{shard1} OR {shard2} = {shard1, shard2}
```

Wynik tej operacji zapisywany jest na stosie - Rysunek 15. Po zakończeniu przetwarzania zapytania wartość na stosie będzie specyfikowała, na których shardach zapytanie należy uruchomić.



Rysunek 15

Wykorzystanie stosu shardów znacznie upraszcza proces pisania własnych strategii wyboru sharda. Programista nie musi tworzyć złożonego kodu analizy składniowej zapytania, a jedynie kod, który sprawdza użyty operator oraz parametry prostego wyrażenia porównania. Rysunek 16 Diagram klas - interfejs strategii wyboru sharda przedstawia interfejs strategii wyboru sharda.



Rysunek 16 Diagram klas - interfejs strategii wyboru sharda

Parser uruchamiając metodę *selectShards* przekazuje jej obiekt *ParameterInfo* zawierający informacje o wyrażeniu, takie jak: nazwa kolumny, tabeli, użyty operator porównania, rodzaj zapytania czy wartość. Wszystkie wyrażenia porównania mają więc postać:

```
tabela.kolumna operator wartość
```

Parser wykorzystuje zawsze tylko jedną implementację strategii wyboru sharda. Domyślnie jest to klasa *shards.GlobalShardsSelectionStrategy*. Implementacja ta pełni rolę delegata – sprawdza ona tabelę wykorzystaną w zapytaniu, a następnie deleguje wywołanie do

odpowiedniej strategii opisanej w konfiguracji. To sprawia, że programista może wykorzystywać wiele strategii, w zależności od tabeli.

W przypadku poleceń INSERT wykorzystywane są te same obiekty strategii co w zapytaniach typu SELECT. Po prostu operator wyrażenia w tym wypadku jest zawsze „=”.

```
INSERT INTO table (id, col) values (1, 10);
```

Dodawany wiersz trafi więc do shardów wybranych dla wyrażenia id = 10.

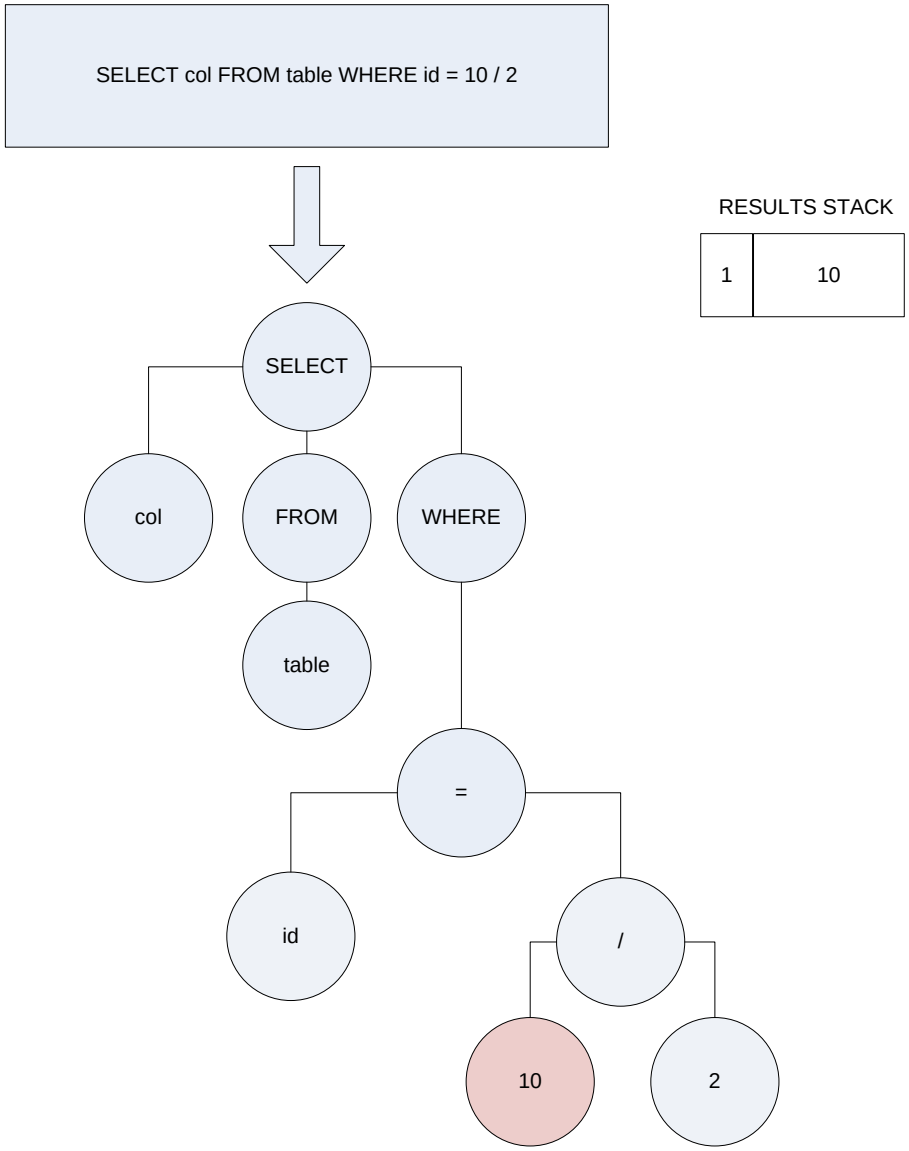
5.1.1.2 Stos rezultatów

Dowolne zapytania zawierać mogą wyrażenia arytmetyczne, na przykład:

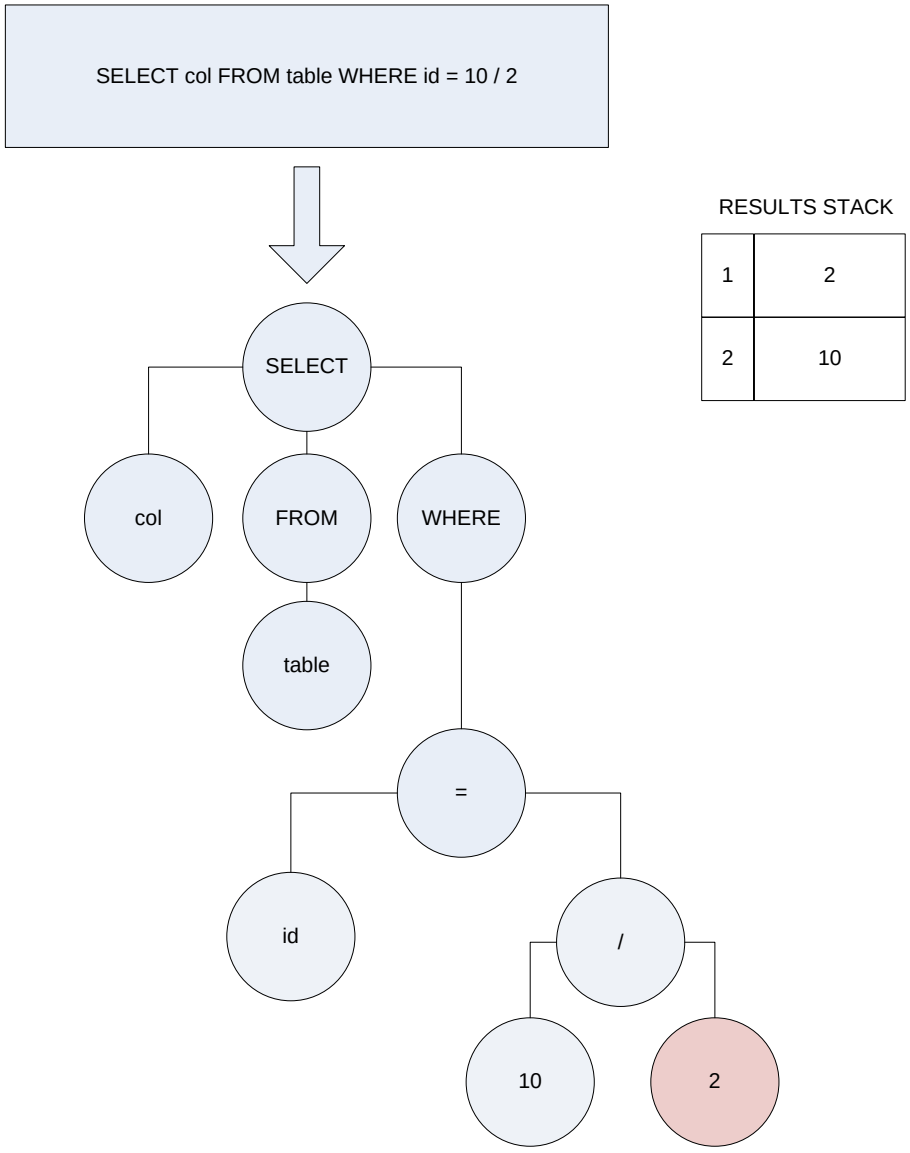
```
SELECT id, col FROM table WHERE id = 10 - 9;
INSERT INTO (id, col) VALUES (1, 10 / 2);
```

Programista strategii wyboru nie musi się jednak tym przejmować – wartość jest obliczana zanim przekazana zostanie do strategii.

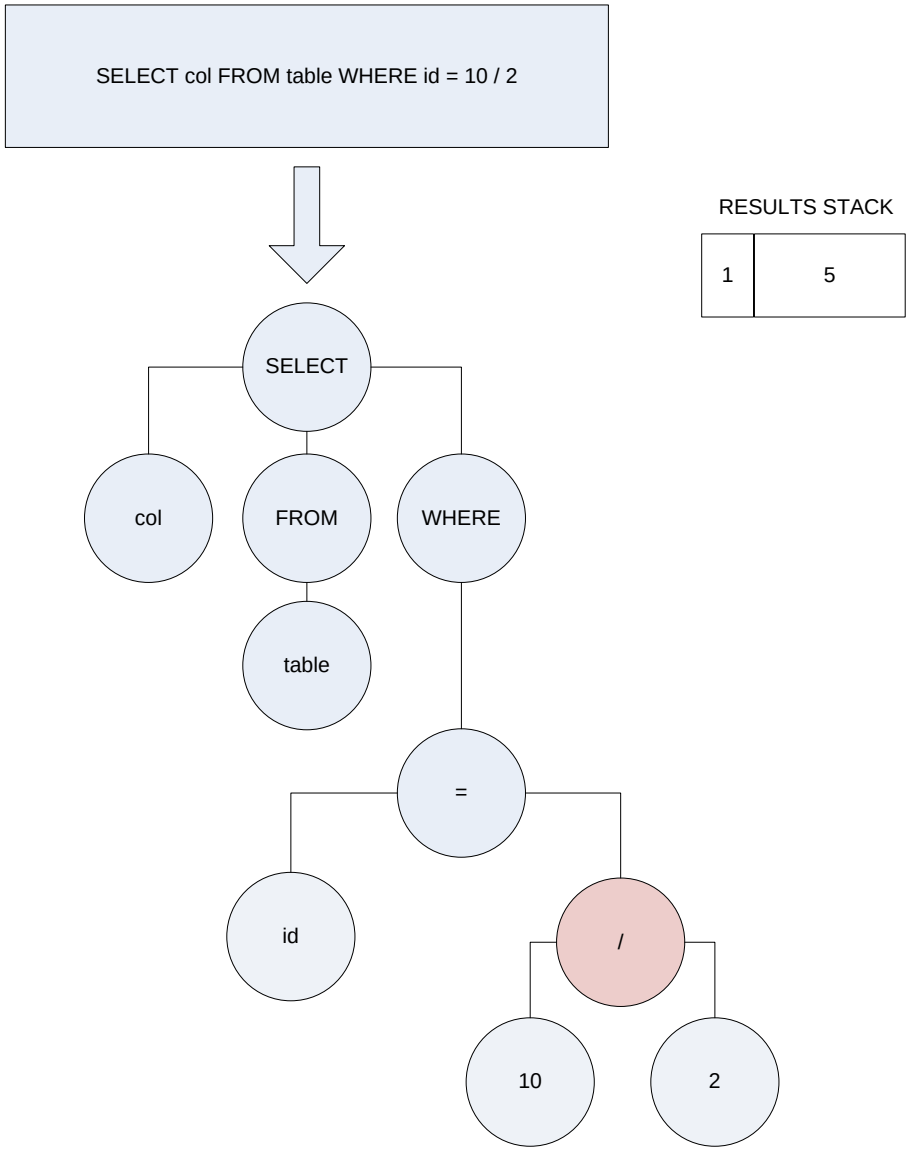
Kalkulacja wyrażeń arytmetycznych wykonywana jest przy użyciu stosu rezultatów oraz obiektu klasy *utils.Calculation*. Każda napotkana liczba lub łańcuch znaków dodawana jest do stosu rezultatów. W przypadku natrafienia na wyrażenie arytmetyczne (na przykład dodawanie) 2 obiekty ściągane są ze stosu. Wykorzystanie stosu rezultatów pokazane zostało na Rysunek 17, Rysunek 18 oraz Rysunek 19. Najpierw na stosie umieszczana jest liczba 10, będąca lewą stroną wyrażenia. Następnie liczba 2 będąca prawą stroną wyrażenia. Następnie wyrażenie dzielenia pobiera ze stosu dwie ostatnie liczby i wykonuje operację. Wynik operacji umieszczany jest na stosie.


SELECT col FROM table WHERE id = 10 / 2
RESULTS STACK
1
10
SELECT
col
FROM
WHERE
table
=
id
/
10
2

Rysunek 17


SELECT col FROM table WHERE id = 10 / 2
RESULTS STACK
1
2
2
10
SELECT
col
FROM
WHERE
table
=
id
/
10
2

Rysunek 18



Rysunek 19

5.1.1.3 Przepisywanie zapytań

Po przetworzeniu zapytania *SQLParser* zwraca listę shardów, na których zapytanie powinno się uruchomić oraz szereg danych opisujących zapytanie. Najważniejsze jest jednak przepisane zapytanie – czyli zapytanie SQL powstałe w wyniku przerobienia oryginalnego zapytania. Przepisywanie zapytań jest w wielu przypadkach konieczne, aby uzyskać poprawne wartości. Omówienie potrzeby przepisywaniu zapytań znaleźć można w rozdziale 4.3.

Po przetworzeniu oryginalnego zapytania tworzone są obiekty Java odpowiadające poszczególnym częściom zapytania. Przykładowo dla zapytania SELECT tworzony jest obiekt klasy *net.sf.jsqlparser.statement.select.Select*, a dla zapytania INSERT obiekt klasy *net.sf.jsqlparser.statement.insert.Insert*. Obiekt *Select* zawiera wszystkie informacje o zapytaniu: wyrażenia kolumnowe, klauzulę FROM, WHERE, GROUP BY itp. Dla każdego takiego elementu tworzony jest obiekt odpowiedniej klasy. Przykładowo dla FROM jest to *net.sf.jsqlparser.statement.select.FromItem*, a dla WHERE obiekt *net.sf.jsqlparser.expression.Expression*. Wszystkie obiekty są zmienne (ang. *mutable*), dzięki czemu parser podczas przetwarzania zapytania może je modyfikować. Dodatkowo każdy z obiektów ma zaimplementowaną metodę *toString()*, która zwraca fragmenty zapytania SQL. Przykładowo obiekt klasy *Select* zwróci całe zapytanie typu SELECT. Właściwości te zostały wykorzystane do przepisywania zapytań. Parser modyfikuje wybrane obiekt a następnie uruchamia metodę *toString()*. Otrzymany łańcuch tekstowy jest zwracany jako przepisane zapytanie.

W przypadku napotkania w zapytaniu na funkcje agregujące AVG do listy kolumn SELECT dodawane są wyrażenia SUM oraz COUNT. Każde kolumnowe wyrażenie użyte w SELECT to obiekt klasy *net.sf.jsqlparser.statement.select.SelectItem*. Lista tego typu obiektów zawarta w obiekcie zapytania jest modyfikowana poprzez dodanie na jej końcu wyrażeń będących funkcjami SUM oraz COUNT. Podobna operacja wykonywana jest po napotkaniu w zapytaniu na klauzule ORDER BY, GROUP BY oraz HAVING. Wyrażenia podane jako parametry tych klauzul dodawane są do listy wyrażeń kolumnowych SELECT. Gdy z kolei parser natrafi na klauzulę OFFSET to usuwa ją z zapytania poprzez modyfikację obiektu klasy *net.sf.jsqlparser.statement.select.Limit*. Obiekt ten posiada 2 parametry: limit (odpowiada klauzuli LIMIT) oraz offset (odpowiada OFFSET). Poprzez wyzerowanie tego ostatniego klauzula OFFSET zniknie z zapytania. Z kolei klauzula HAVING jest usuwana modyfikując obiekt zapytania poprzez przypisanie właściwości o nazwie having wartości null.

5.1.1.4 Zapytania DDL

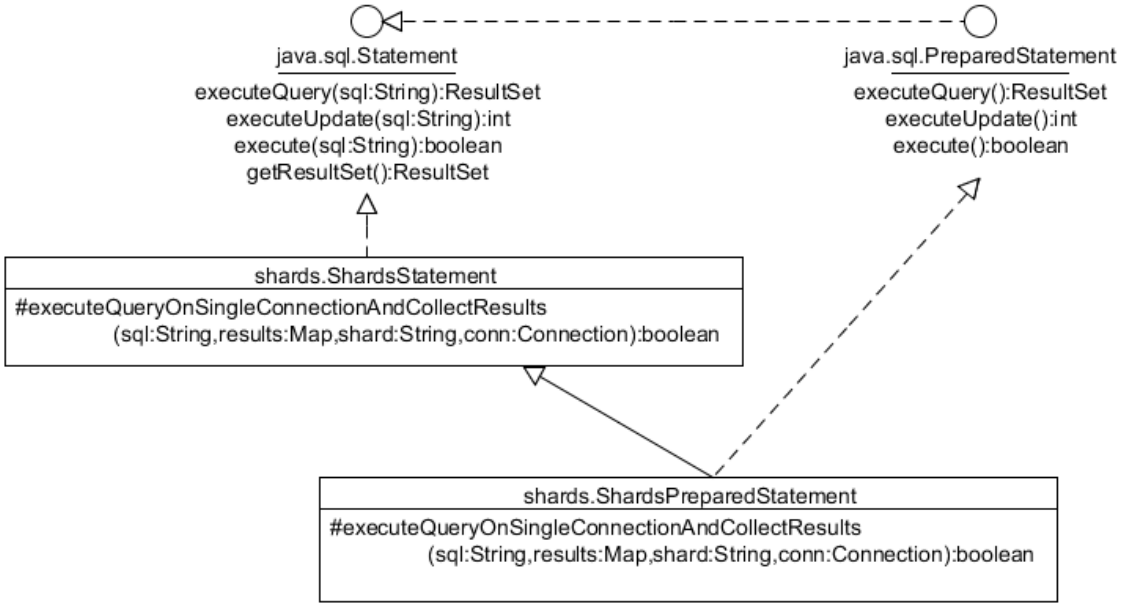
W przypadku wszystkich zapytań DDL parser zawsze zwraca wszystkie shardy. Zapytania CREATE czy DROP muszą być bowiem uruchomione na każdej bazie danych. W wyniku tego faktu nie ma tu konieczności przetwarzania zapytań oraz ich przepisywania.

5.1.2 HavingExpressionVisitor

Podczas przepisywania zapytań klauzula HAVING jest traktowana nadzwyczajnie. Nie może być przekazana bezpośrednio do shardów, w wyniku czego jest usuwana podczas przepisywania zapytania (więcej informacji można znaleźć w rozdziale 4.3.4). Zamiast tego filtrowanie zgromadzonych wyników odbywa się w obiekcie klasy *shards.HavingExpressionVisitor* – specjalnym obiekcie wizytatora. Na wejściu obiekt otrzymuje pierwszy rekord wyników oraz oryginalne zapytanie. Wizytator przetwarza zapytanie sprawdzając czy rekord ma znaleźć się w wynikowym zbiorze. Operacja ta powtarzana jest dla każdego rekordu.

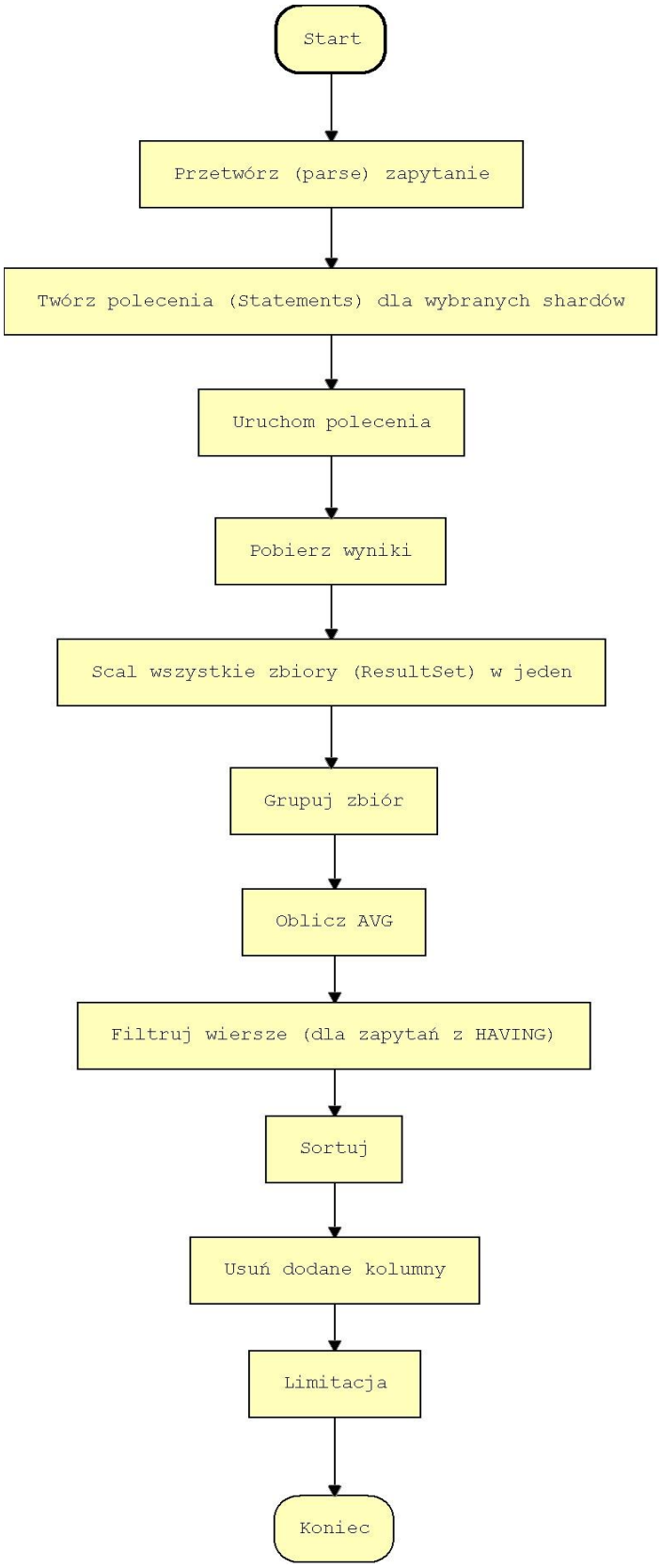
5.2. Implementacja ShardsStatement

Sterownik JDBC definiuje kilka interfejsów, które muszą być zaimplementowane przez dostawcę sterownika. Jednym z nich jest *java.sql.Statement*, czyli obiekt polecenia, za pomocą którego programista uruchamia zapytania SQL. W prototypowym sterowniku będącym efektem tej pracy interfejs ten implementowany jest przez klasę *shards.ShardsStatement*. Rysunek 20 przedstawia diagram klas. Klasa *ShardsStatement* służy do uruchamiania zapytań ad-hoc, czyli za pomocą jednego obiektu polecenia programista może uruchamiać wiele zapytań. Druga implementacja tego interfejsu to *shards.ShardsPreparedStatement*. Klasa ta dodatkowo implementuje interfejs *java.sql.PreparedStatement*, który służy do uruchamiania zapytań uprzednio przygotowanych. Jeden obiekt *ShardsPreparedStatement* służy do uruchomienia jednego określonego na początku zapytania. Zapytanie to może być dodatkowo parametryzowane. Aby nie powielać kodu klasa *ShardsPreparedStatement* dziedziczy po *ShardsStatement* nadpisując jedną z jej metod. Metoda ta nosi nazwę *executeQueryOnSingleConnectionAndCollectResults* i jak sama nazwa wskazuje służy ona do uruchomienia zapytania na pojedynczym połączeniu (shardzie) i zebraniu wyników.



Rysunek 20 Diagram klas poleceń

Obie klasy korzystają z tego samego algorytmu, który w ogólności przedstawia Rysunek 21. Wybrane operacje zostały opisane w kolejnych podrozdziałach.



Rysunek 21 Algorytm uruchomienia zapytania

5.2.1 Uruchamianie poleceń

W obecnej wersji sterownika zapytania uruchamiane są szeregowo – w pętli po kolei na każdym z wybranych shardów. W celach optymalizacyjnych będzie można jednak w przyszłości zrównoleglić wykonywanie – np. za pomocą puli wątków. W takich przypadku sumaryczny czas zapytania będzie krótszy.

5.2.2 Grupowanie

Algorytm grupowania realizowany jest wg następującego algorytmu: najpierw wykonywane jest sortownie po kolumnach, po których ma nastąpić grupowanie. Następnie w pętli porównywane są sąsiednie rekordy. Jeśli wartości w tych w kolumnach są równe to następuje agregacja rekordów w jeden. W przeciwnym wypadku porównywani są kolejni sąsiedzi.

Algorytm agregacji polega na złączeniu dwóch rekordów w jeden. Dla wszystkich znalezionych w zapytaniu funkcji agregujących (np. SUM lub COUNT) pobierane są wartości argumentów ze zbioru wyników. Wykonywana jest operacja (np. dodawanie dwóch wartości dla funkcji SUM). Wynik operacji trafia do kolumny, z której został pobrany argument. Tak utworzony wiersz dodawany jest do wynikowego zbioru.

Przykładowo dla zapytania

```
SELECT dept_no, sum(salary) FROM emp GROUP BY dept_no
```

ze wszystkich wybranych shardów zwrócone zostaną następujące rekordy:

Nr rekordu	**Dept_no**	**Sum(salary)**
1	1	4000
2	2	2500
3	1	5000
4	2	4000

Algorytm najpierw posortuje wyniki po kolumnie *dept_no*, która występuje w klauzuli GROUP BY:

Nr rekordu	**Dept_no**	**Sum(salary)**

1	1	4000
3	1	5000
2	2	2500
4	2	4000

Następnie porównywane są 2 sąsiadujące rekordy o numerach 1 oraz 3. W tym wypadku wartość kolumny *dept_no* jest identyczna więc następuje agregacja. Pobierane są argumenty dla funkcji agregującej: 4000 oraz 5000. Wykonywana jest operacja dodawania i w miejsce tej kolumny wstawiany jest wyniki: 9000. Zbiór wyników po tej operacji wygląda następująco:

Nr rekordu	**Dept_no**	**Sum(salary)**
1	1	9000
2	2	2500
4	2	4000

Następnie zmodyfikowany rekord porównywany jest ze swoim sąsiadem o numerze 2. Wartości kolumny *dept_no* są różne więc nie następuje agregacja. Algorytm przechodzi więc do porównania kolejnych rekordów o numerach 2 oraz 4. Ze względu na tą samą wartości w kolumnie *dept_no* rekordy są łączone w jeden i wynik całej operacji grupowania wygląda następująco:

Nr rekordu	**Dept_no**	**Sum(salary)**
1	1	9000
2	2	6500

5.2.3 Obliczanie wartości średniej

W przypadku funkcji AVG zapytanie jest przepisywane, aby zamiast AVG zawierało funkcje COUNT oraz SUM. Więcej na ten temat przeczytać można w rozdziale 4.3.1. Następnie uruchamiane jest na wybranych shardach, a wyniki grupowane są wg algorytmu podanego w rozdziale 5.2.2 – obliczana jest ilość oraz suma. Następnie dla wszystkich wierszy obliczana jest średnia wg wzoru:

```
średnia = suma / ilość
```

i wstawiana do odpowiedniej kolumny. Dwie dodane kolumny dla funkcji SUM oraz COUNT na koniec procesu są usuwane z wyników.

5.2.4 Sortowanie

Sortowanie wyników także wykonywane jest po stronie sterownika. Do tego celu wykorzystywane są znane z języka Java obiekty *java.util.Comparator*. Specjalna implementacja dostarczana przez sterownik o nazwie *RowComparator* porównuje ze sobą całe wiersze, a dokładnie wszystkie kolumny wpisane w klauzuli ORDER BY.

To kończy omówienie sposobu implementacji prototypu. W ostatniej części rozdziału zaprezentowano wyniki testów wydajnościowych, które ukazują praktyczne efekty poczynionych starań.

5.3. Testy wydajnościowe

Do testowania wykorzystano trzy bazy danych – nieshardowaną, shard1 oraz shard2. Każda z nich zawierała tabelę przedstawioną na Listing 28.

```
create table items (
 id bigserial primary key,
 name text,
 category bigint not null,
 price numeric(10,2)
);
```

Listing 28 Kod tworzący tabelę w testowych bazach danych

Do pierwszej bazy wstawiono wygenerowane przez skrypt z Listing 29 rekordy. Pozostałe bazy zostały wypełniony za pomocą analogicznych skryptów tak, żeby finalnie uzyskać następującą konfigurację:

- Tabela z bazy nieshardowanej – milion rekordów
- Tabela z bazy shard1 – pół miliona rekordów, kategorie 0, 2, 4, 6, 8...
- Tabela z bazy shard2 – pół miliona rekordów, kategorie 1, 3, 5, 7…

```
CREATE OR REPLACE FUNCTION fill_database()
```

```
 RETURNS void AS
$$
 declare
   n integer;
 begin
         n = 0;
         FOR i IN 1..1000000
         LOOP
           n = n + 1;
           IF n > 10 THEN n:=1; END IF;
           INSERT INTO items (name, category, price) VALUES
('item' || i, n,
i * 1.23);
         END LOOP;
 end;
$$
LANGUAGE 'plpgsql';

select fill_database();

vacuum full verbose analyze items;
```

Listing 29 Skrypt inicjalizujący nieshardowaną testową bazę danych

Dane są rozdzielane do odpowiednich baz danych za pomocą strategii wyboru sharda opartej o funkcję haszującą. Kluczem tej strategii jest kolumna „category”.

5.3.1 Test narzutu sterownika

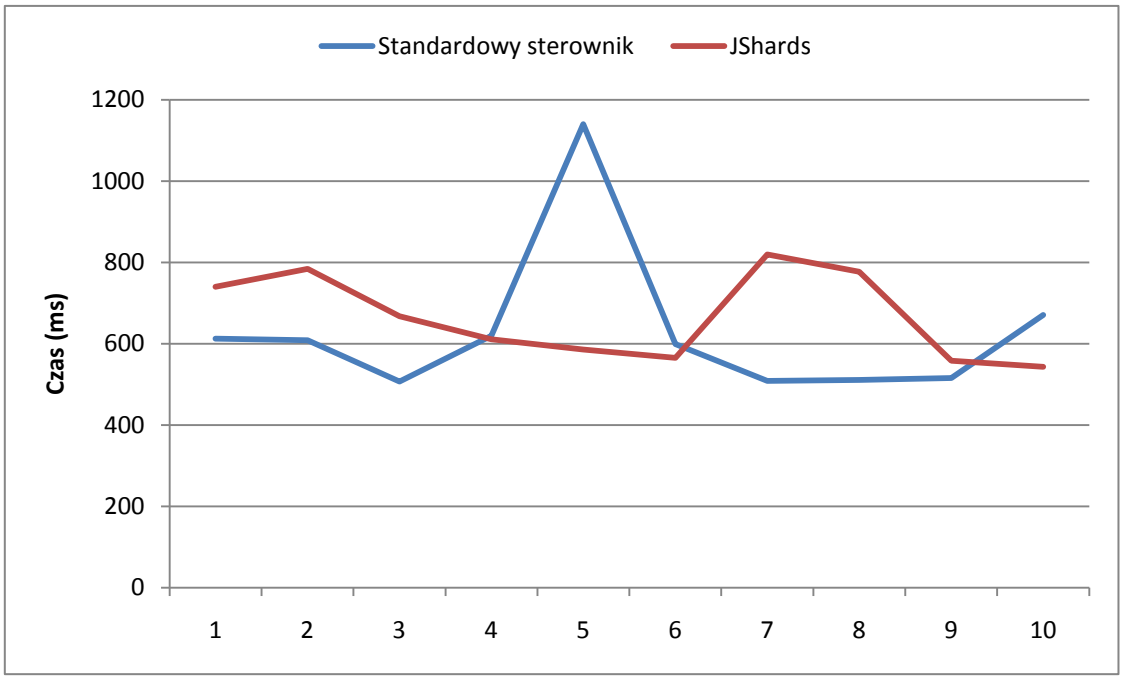
W celu zmierzenia narzutu sterownika wykorzystano dwie maszyny fizyczne. Zarówno klient, jak i serwer, na którym znajdowała się baza testowa, wyposażone były w czterordzeniowe procesory i 3 GB RAM. Komunikacja między klientem i serwerem odbywała się poprzez sieć lokalną o przepustowości 100 Mb/s.

Jedna z konfiguracji korzystała ze standardowego sterownika JDBC dostarczonego dla bazy PostgreSQL. Druga natomiast wykorzystywała prototyp sterownika JShards.

Na każdej konfiguracji wykonano trzy różne zapytania. Każde z nich powtórzono dziesięciokrotnie. Czasy wykonania poszczególnych zapytań wyrażone w milisekundach zawarto w Tabela 1, 2 oraz 3 oraz na Wykresach Wykres 1, 2 i 3. Dla każdego wyniku wyliczono procentowy narzut sterownika. Ponadto dla każdego zapytania obliczono wartości średnie z wyników cząstkowych.

Standardowy sterownik	JShards	Narzut
613	740	20,72%
609	784	28,74%
507	668	31,76%
620	611	-1,45%
1140	586	-48,60%
599	565	-5,68%
509	820	61,10%
511	777	52,05%
516	558	8,14%
671	543	-19,08%
629,5	**665,2**	**12,77%**

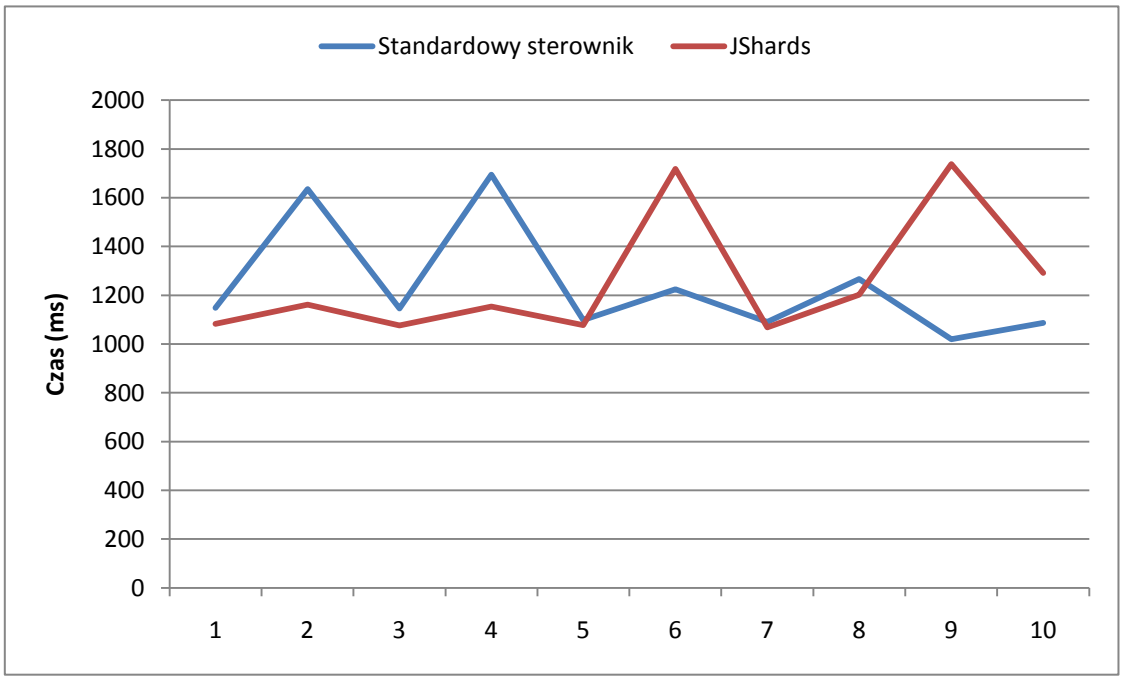
Tabela 1 Wyniki testu narzutu sterownika JShards dla zapytania SELECT * FROM items WHERE category = 1



Wykres 1 Wyniki testu narzutu sterownika JShards dla zapytania SELECT * FROM items WHERE category = 1 (mniej = lepiej)

Standardowy sterownik	JShards	Narzut
1149	1083	-5,74%
1636	1161	-29,03%
1146	1076	-6,11%
1695	1154	-31,92%
1098	1078	-1,82%
1224	1718	40,36%
1090	1068	-2,02%
1267	1202	-5,13%
1020	1738	70,39%
1087	1291	18,77%
1241,2	**1256,9**	**4,77%**

Tabela 2 Wynika testu narzutu sterownika JShards dla zapytania SELECT * FROM items WHERE category = 1 OR category = 2

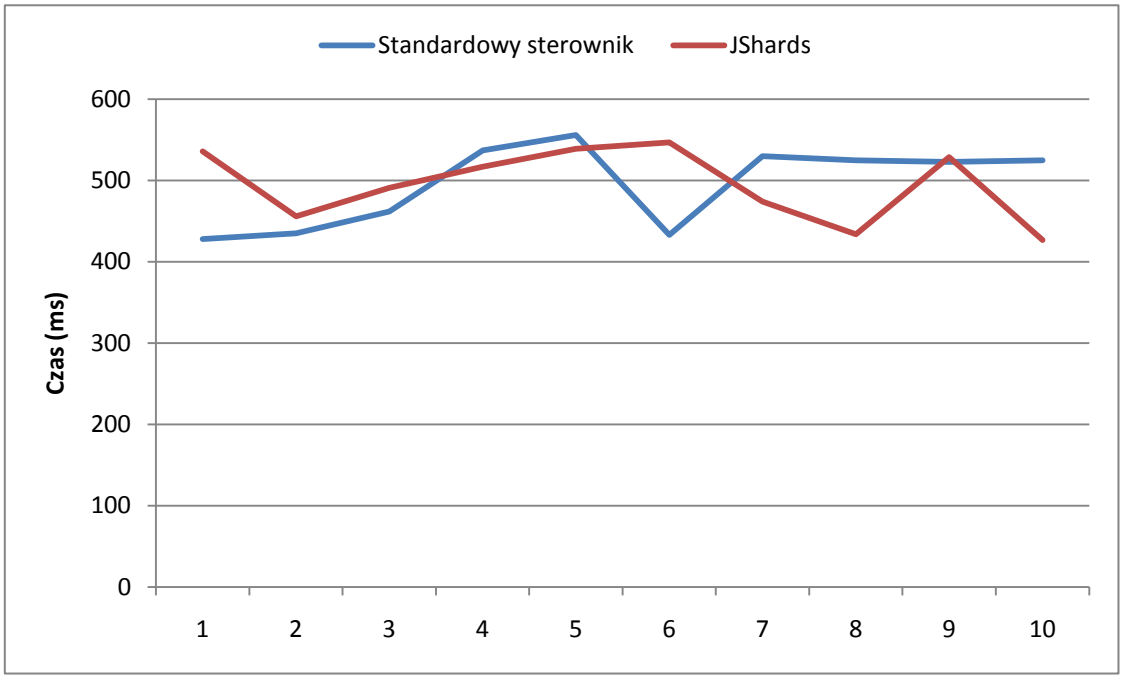


Wykres 2 Wynika testu narzutu sterownika JShards dla zapytania SELECT * FROM items WHERE category = 1 OR category = 2 (mniej = lepiej)

Standardowy sterownik	JShards	Narzut
428	536	25,23%
435	456	4,83%
462	491	6,28%
537	517	-3,72%
556	539	-3,06%
433	547	26,33%
530	474	-10,57%
525	434	-17,33%
523	529	1,15%

	525	427	-18,67%
	495,4	**495**	**1,05%**

Tabela 3 Wynik testu narzutu sterownika JShards dla zapytania SELECT count(category) FROM items WHERE category BETWEEN 1 AND 3 GROUP BY category



Wykres 3 Wynik testu narzutu sterownika JShards dla zapytania SELECT count(category) FROM items WHERE category BETWEEN 1 AND 3 GROUP BY category (mniej = lepiej)

Na podstawie przeprowadzonych testów obliczono średni narzut sterownika. Wynosi on 6,20%. Narzut ten jest spowodowany koniecznością wykonywania dodatkowych operacji omówionych w rozdziałach 4 i 5.

5.3.2 Testy przepustowości

Do testów przepustowości wykorzystano trzy fizyczne komputery. Komputer klienta wyposażony był w czterordzeniowy procesor o częstotliwości 2,4 GHz oraz 3 GB pamięci operacyjnej. Oba serwery posiadały identyczną konfiguracje – czterordzeniowy procesor o częstotliwości 3 GHz i 3 GB pamięci operacyjnej. Serwery zostały skonfigurowane zgodnie z założeniami przedstawionymi na początku podrozdziału, tzn.:

- Serwer 1a – jednolita baza danych zawierająca milion rekordów – wykorzystywany w teście bez shardów
- Serwer 1b – shard1 zawierał pół miliona rekordów, kategorie 0, 2, 4, 6, 8...

- Serwer 2 – shard2 zawierał pół miliona rekordów, kategorie 1, 3, 5, 7…

Testy polegały na uruchomieniu dwóch zestawów zapytań zarówno w konfiguracji nieshardowanej, na serwerze 1a, jak i przy użyciu dwóch shardów – serwery 1b i 2. Każde zapytanie było wykonywane przez od jednego do czterech wątków. Test dla każdej konfiguracji „zestaw zapytań – ilość wątków" był z kolei powtarzany czterokrotnie w celu uśrednienia rezultatów.

Podobnie, jak poprzednio w tabelach umieszczono czasy wykonania poszczególnych zestawów wyrażone w milisekundach. Tabele wzbogacono o procentową różnicę między czasami wykonań i wartości uśrednione.

Pierwszy zestaw złożony był z zapytań zawartych na Listing 30. Wyniki znajdują się w Tabela 4, 5, 6 i 7 oraz na Wykresie Wykres 4.

```
SELECT * FROM items WHERE category = 1
SELECT * FROM items WHERE category = 1 OR category = 2
SELECT count(category) FROM items WHERE category BETWEEN 1 AND 3
GROUP BY category
```

Listing 30 Zapytania wykorzystane w pierwszym zestawie testującym

Standardowy sterownik	JShards	Przyrost
1653	1404	15,06%
1636	1406	14,06%
1610	1495	7,14%
1624	1445	11,02%
1630,75	**1437,5**	**11,82%**

Tabela 4 Wyniki testu przy 1 wątku na każde zapytanie (3 wątki równolegle)

Standardowy sterownik	JShards	Przyrost
3693	1918	48,06%
3630	1930	46,83%
3743	1995	46,70%
3731	2003	46,31%
3699,25	**1961,5**	**46,98%**

Tabela 5 Wyniki testu przy 2 wątkach na każde zapytanie (6 wątków równolegle)

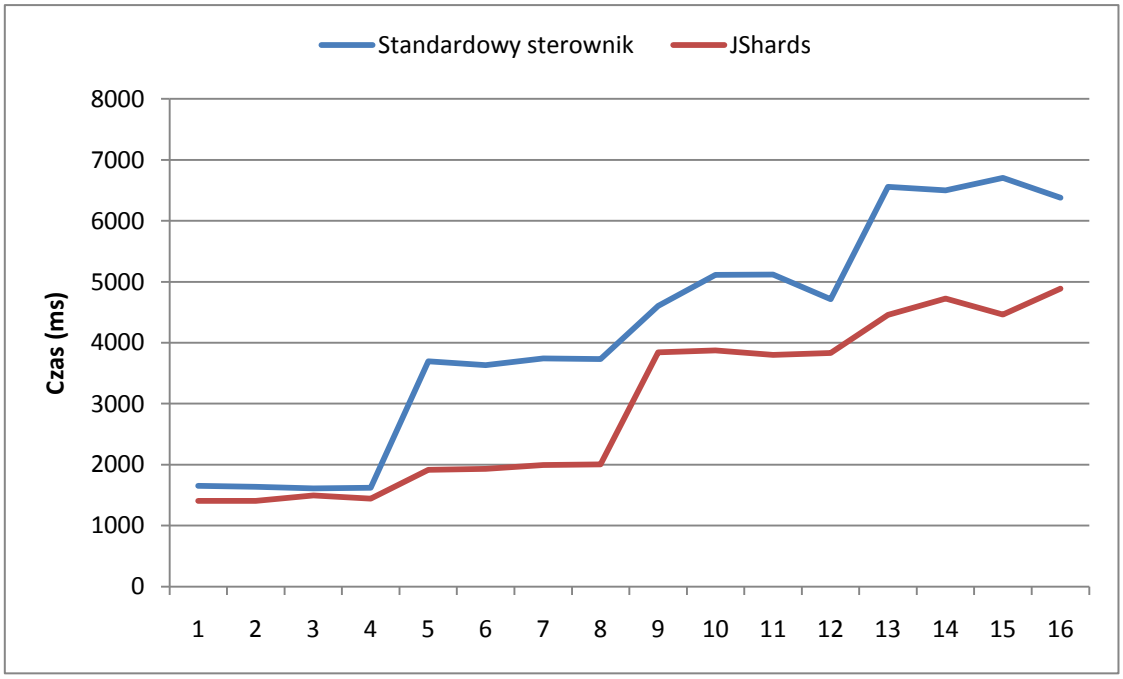
Standardowy sterownik	JShards	Przyrost

4605	3842	16,57%
5111	3874	24,20%
5119	3803	25,71%
4716	3831	18,77%
4887,75	**3837,5**	**-21,31%**

Tabela 6 Wyniki testu przy 3 wątkach na każde zapytanie (9 wątków równolegle)

Standardowy sterownik	JShards	Przyrost
6556	4458	32,00%
6501	4724	27,33%
6704	4460	33,47%
6380	4886	23,42%
6535,25	**4632**	**29,06%**

Tabela 7 Wyniki testu przy 4 wątkach na każde zapytanie (12 wątków równolegle)



Wykres 4 Wyniki testu dla pierwszego zestawu testującego (mniej = lepiej)

Drugi zestaw natomiast złożony był z zapytań zawartych na Listing 31. Wyniki znajdują się w Tabelach 8, 9, 10 i 11. Zobrazowano je także na Wykres 5.

```
SELECT * FROM items WHERE category = 1
SELECT * FROM items WHERE category = 1 OR category = 2
SELECT count(category) FROM items WHERE category BETWEEN 1 AND 3
GROUP BY category
UPDATE items SET price = 13.12 WHERE category = 1 and id = 3
```

```
INSERT INTO items (id, name, category, price) values (default,
'test', 1, 12.11)
INSERT INTO items (id, name, category, price) values (default,
'test', 2, 14.11)
```

Listing 31 Zapytania wykorzystywane w drugim zestawie testującym

Standardowy sterownik	JShards	Przyrost
2017	1089	46,01%
1986	1139	42,65%
2011	1080	46,30%
1968	1079	45,17%
1995,5	**1096,75**	**45,03%**

Tabela 8 Wyniki testu przy 1 wątku na każde zapytanie (6 wątków równolegle)

Standardowy sterownik	JShards	Przyrost
3661	1945	46,87%
3720	1985	46,64%
3743	2000	46,57%
3706	1978	46,63%
3707,5	**1977**	**46,68%**

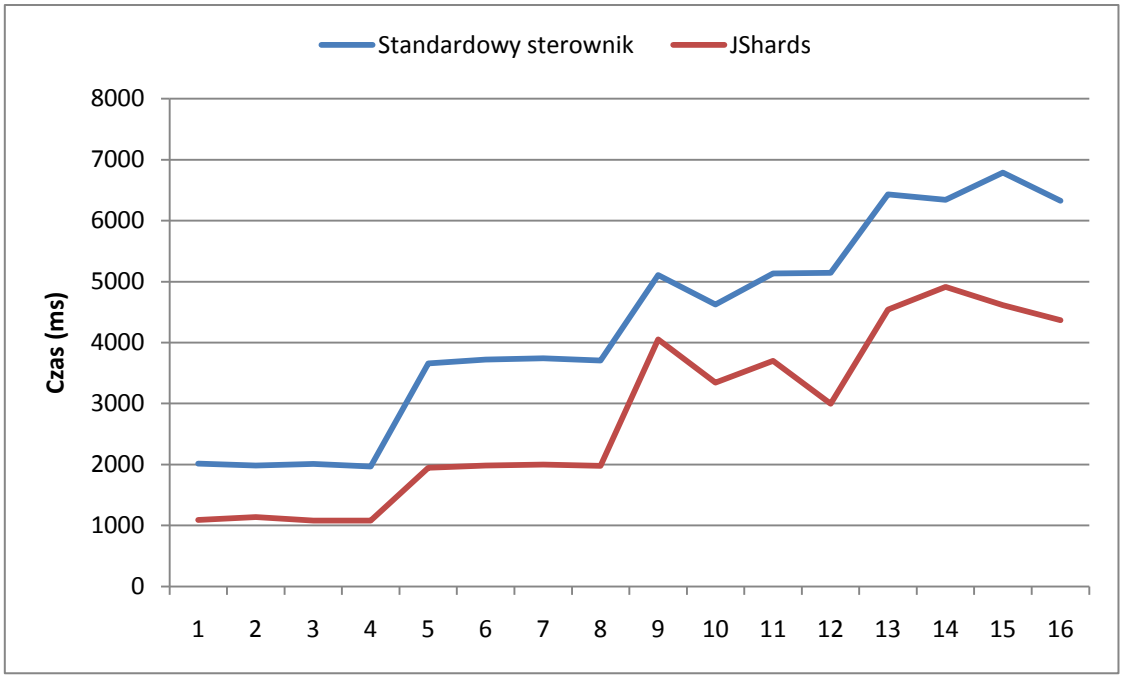
Tabela 9 Wyniki testu przy 2 wątkach na każde zapytanie (12 wątków równolegle)

Standardowy sterownik	JShards	Przyrost
5110	4050	20,74%
4623	3342	27,71%
5135	3700	27,95%
5145	2996	41,77%
5003,25	**3522**	**29,54%**

Tabela 10 Wyniki testu przy 3 wątkach na każde zapytanie (18 wątków równolegle)

Standardowy sterownik	JShards	Przyrost
6431	4542	29,37%
6341	4913	22,52%
6786	4615	31,99%
6327	4367	30,98%
6471,25	**4609,25**	**28,72%**

Tabela 11 Wyniki testu przy 4 wątkach na każde zapytanie (24 wątków równolegle)



Wykres 5 Wyniki testu dla drugiego zestawu testującego (mniej = lepiej)

Na podstawie przeprowadzonych testów można szacować, że dzięki zastosowaniu sterownika JShards średni przyrost przepustowości wynosi 32,39%. Wynik ten dotyczy zastosowania jedynie dwóch shardów. Przy większej ich ilości osiągniemy jeszcze lepsze rezultaty.

Warto jednak zaznaczyć, iż zapytania skonstruowane były tak, żeby w pełni wykorzystać architekturę shardów – zawsze zawierały odwołanie do klucza wyboru sharda. W innym przypadku przyrost wydajności nie byłby tak znaczący.

Po zapoznaniu się ze szczegółami implementacyjnymi prototypu sterownika, warto przyjrzeć się sposobowi jego wykorzystania w praktyce. Temu zagadnieniu poświęcony będzie rozdział 6.

6. Podręcznik użytkownika

Rozdział ten ma na celu opisanie sposobu użycia narzędzia JShards. Na początku omówiona zostanie składnia i semantyka pliku konfiguracyjnego. W szczególności pokazany będzie sposób konfiguracji przykładowej strategii wyboru sharda.

Następnie zilustrowany będzie najczęstszy sposób użycia sterownika – wykorzystanie go we własnej aplikacji Java. Ponadto opisany zostanie sposób użycia JShards we współpracy z zewnętrzną aplikacją do zarządzania bazami danych. Jako przykład wykorzystano narzędzie SQuirreL SQL Client[13].

Rozdział zamknie omówienie sposobu tworzenia własnej strategii wyboru sharda.

6.1. Plik konfiguracyjny

Plik konfiguracyjny stanowi serce sterownika JShards. Tutaj definiowane są szczegóły konfiguracyjne połączeń z rozproszonymi bazami danych. Tutaj także wybiera i konfiguruje się strategie wyboru sharda. Przyjrzyjmy się teraz zawartości przykładowego pliku z Listing 32.

```
---
drivers:
  - "org.postgresql.Driver"

connections:
  -
    name: shard1
    url: "jdbc:postgresql://dev:5432/shards"
    user: shards
    password: shards
  -
    name: shard2
    url: "jdbc:postgresql://dev:5432/shards2"
```

[13] Do pobrania z http://squirrel-sql.sourceforge.net/

```
    user: shards
    password: shards

strategy:
  -
    table: shards
    class: "shards.RangeShardsSelectionStrategy"
    params:
      column: id
      ranges:
        -
          from: 0
          to: 2
          shard: shard1
        -
          from: 3
          shard: shard2
```

Listing 32 Przykładowy plik konfiguracyjny

Plik ten zapisany jest w formacie YAML. Podzielony jest na trzy główne sekcje: drivers, connections oraz strategy.

W sekcji pierwszej definiowane są w pełni kwalifikowane nazwy klas sterowników JDBC. Sterowniki te są rejestrowane w klasie DriverManager i zostaną wykorzystane do utworzenia połączeń z poszczególnymi rozproszonymi bazami danych.

W sekcji drugiej definiuje się parametry połączeń do poszczególnych shardów. Każde połączenie charakteryzowane jest przez cztery parametry:

- name – identyfikator sharda; wykorzystywany dalej przy konfiguracji strategii, jak i w całej aplikacji, w szczególności podczas wybierania podzbioru shardów
- url – adres URL do rzeczywistej bazy danych; może wskazywać na dowolny serwer bazodanowy, na którym działa system zarządzania bazą danych kompatybilny z jednym ze sterowników wyszczególnionych w sekcji „drivers”

- user – identyfikator użytkownika mającego uprawnienia do łączenia się z bazą danych identyfikowaną przez url
- password – hasło użytkownika mającego uprawnienia do łączenia się z bazą danych identyfikowaną przez url

Zestaw tych czterech parametrów określa shard. Każdy shard, którego chcemy używać musi być zdefiniowany w tej sekcji.

W sekcji strategy z kolei opisana jest logika działania strategii wyboru shardów, w szczególności zaś konkretne strategie przypisane są do tabel w rozproszonej bazie danych.

Każda pozycja w tej sekcji opisana jest przez trzy parametry:

- table – nazwa tabeli w bazie danych, która będzie podlegała podziałowi horyzontalnemu
- class – w pełni kwalifikowana nazwa klasy strategii wyboru sharda, która będzie wykorzystywana do podziału tabeli
- params – parametr zawierający szczegóły konfiguracyjne strategii reprezentowanej przez klasę określoną w parametrze class.

Najistotniejszym atrybutem w ramach parametru params jest atrybut column. Zawiera on nazwę kolumny tabeli, która stanowić będzie klucz strategii wyboru sharda, czyli dyskryminator, którego wartość będzie determinowała przynależność rekordu do sharda. Atrybut ten będzie charakterystyczny dla właściwie każdej strategii – wyjątek stanowi strategia mirroringu, ponieważ w jej przypadku rekordy trafiają na wszystkie shardy.

Ponadto w ramach parametru params zawarte są właściwości charakterystyczne dla konkretnej strategii wyboru sharda. W przykładowym pliku znajduje się konfiguracja strategii wyboru sharda opartej o przedziały. Za definicję przedziałów odpowiada atrybut ranges, który zawiera trójki złożone z następujących własności:

- from – lewe ograniczenie przedziału
- to – prawe ograniczenie przedziału

- shard – identyfikator sharda, do którego trafi zapytanie, o ile wartość klucza strategii sharda będzie mieściła się w zadanym przedziale. Identyfikator tutaj podany musi należeć do zbioru identyfikatorów shardów zdefiniowanych w sekcji connections

Przedział definiowany przez każdą trójkę jest przedziałem obustronnie domkniętym.

Skoro znamy już sposoby konfiguracji aplikacji, przetestujmy jej działanie w praktyce.

6.2. Wykorzystanie sterownika w aplikacji Java

Sterownik JShards został zaprojektowany tak, aby jego użycie nie różniło się od użycia każdego innego sterownika zgodnego ze specyfikacją JDBC. W tej sekcji omówiono sposób jego integracji z własną aplikacją napisaną w języku Java.

6.2.1 Tworzenie połączenia

Pierwszym krokiem, który należy wykonać jest stworzenie połączenia. Tak naprawdę jest to jedyny moment, w którym cały proces różni się od wykorzystania klasycznego sterownika JDBC. Otóż w klasycznym podejściu połączenie z bazą zdobywamy poprzez podanie jej adresu URL, loginu oraz hasła użytkownika, co przedstawia Listing 33.

```
Connection c =
DriverManager.getConnection("jdbc:postgresql://localhost/db",
"username", "password");
```

Listing 33 Klasyczny sposób łączenia z bazą danych w JDBC

Natomiast w przypadku sterownika JShards, w miejscu adresu URL bazy danych podajemy lokalizację pliku konfiguracyjnego – Listing 34. Login i hasło nie są wykorzystywane.

```
String url = "jdbc:shards:src/test/resources/config.yml";
Connection connection = DriverManager.getConnection(url);
```

Listing 34 Łączenie z bazą danych za pomocą sterownika JShards

Dzięki temu sterownik JShards może wykorzystać informacje o połączeniach z tego pliku i utworzyć połączenia do rzeczywistych baz danych na jego podstawie. Warto zaznaczyć, że

sterownik ten charakteryzowany jest przez prefiks „jdbc:shards. Pola użytkownika i hasła są pomijane, ponieważ rzeczywiste informacje przechowywane są w sekcji connections pliku konfiguracyjnego.

W tej chwili mamy już dostęp do obiektu klasy implementującej interfejs Connection. Na tym etapie możemy zapomnieć już o fakcie korzystania z architektury rozproszonej. Spróbujmy zatem wykonać kilka zapytań.

6.2.2 Zapytania

Aby wykonań jakiekolwiek zapytania do bazy danych, musimy uzyskać obiekt klasy Statement (lub PreparedStatement, o czym za chwilę). Dokonujemy tego w standardowy dla JDBC sposób zaprezentowany na Listing 35.

```
Statement statement = connection.createStatement();
```

Listing 35

Na obiekcie klasy Statement możemy wykonywać klasyczne metody, co przedstawia Listing 36.

```
statement.executeUpdate("DELETE FROM shards");
statement.executeUpdate("INSERT INTO shards (id,value) VALUES (1,
'test')");
statement.executeUpdate("INSERT INTO shards (id,value) VALUES (2,
'atest')");
ResultSet resultSet = statement.executeQuery("SELECT * FROM shards
ORDER BY value");
```

Listing 36

Nic nie stoi na przeszkodzie, aby używać bardziej rozbudowanych zapytań, jak na Listing 37.

```
ResultSet resultSet = statement.executeQuery("SELECT * FROM shards
WHERE id > 1");
ResultSet resultSet = statement.executeQuery("SELECT count(id)
FROM shards GROUP BY id HAVING id IN (2,3,4,5)");
```

```
ResultSet resultSet = statement.executeQuery("SELECT avg(id) as a,
count(value), sum(id) FROM shards GROUP BY value ORDER BY value");
```

Listing 37

Aplikacja umożliwia także wykorzystanie zapytań typu PreparedStatement. Oczywiście w takim wypadku od obiektu Connection musimy uzyskać obiekt klasy PreparedStatement – Listing 38.

```
PreparedStatement                prepareStatement                =
connection.prepareStatement("SELECT count(id) FROM shards WHERE id
= ?");
prepareStatement.setLong(1, 1l);
ResultSet resultSet = prepareStatement.executeQuery();

PreparedStatement                prepareStatement                =
connection.prepareStatement("UPDATE shards SET value = ? WHERE id
= ?");
prepareStatement.setString(1, "changed");
prepareStatement.setLong(2, 3l);
int count = prepareStatement.executeUpdate();
```

Listing 38 Zapytania typu PreparedStatement

W sposób naturalny można także używać zapytań wsadowych, w szczególności wraz z konstrukcją PreparedStatement – Listing 39.

```
PreparedStatement statement = connection.prepareStatement("UPDATE
shards SET value=? WHERE value = ?");
statement.setString(1, "changed");
statement.setString(2, "test");
statement.addBatch();
statement.setString(1, "test");
statement.setString(2, "changed");
statement.addBatch();
int[] result = statement.executeBatch();
```

Listing 39 Zapytania wsadowe

Wszystkie te konstrukcje działają dokładnie tak, jak spodziewalibyśmy się tego po uruchomieniu ich w klasycznej architekturze scentralizowanej. Wykorzystanie architektury shardów jest transparentne dla programisty. Całą pracę związaną z przepisywaniem zapytań, wyborem shardów i łączeniem wyników wykonuje sterownik.

Przezroczystość rozwiązania, poza brakiem konieczności nauki nowej technologii przez programistę, ma jeszcze jedną zaletę – pozwala na bezproblemowe użycie go w każdej zewnętrznej aplikacji wspierającej sterowniki w standardzie JDBC.

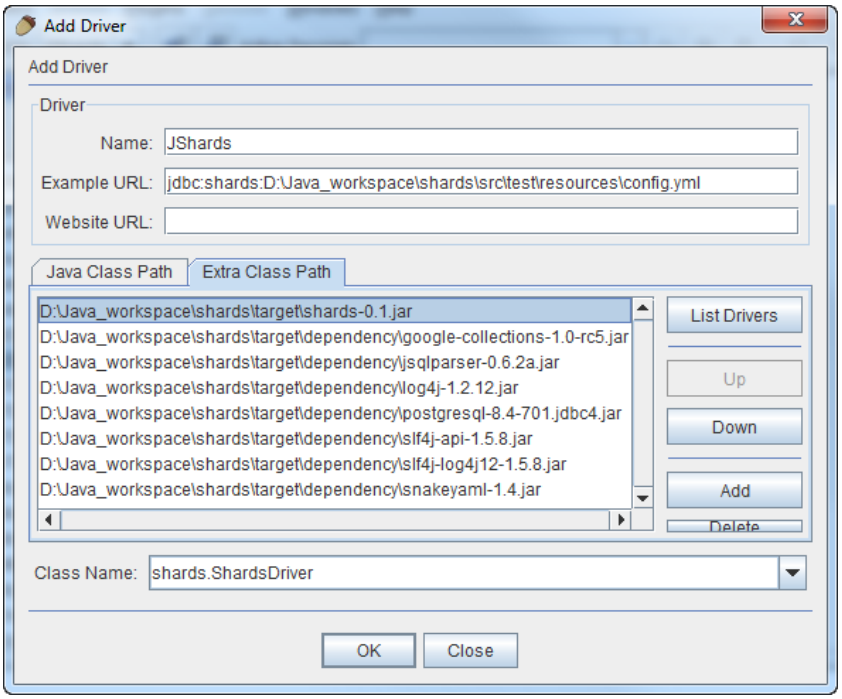
6.3. Wykorzystanie sterownika w zewnętrznej aplikacji na przykładzie SQuirreL SQL Client

Aplikacja JShards może być przydatna nie tylko programistom, ale także administratorom baz danych, którzy chcieliby zarządzać bazami rozproszonymi w architekturze shardów. Dla administratorów tych stworzono wiele narzędzi, które umożliwiają im wygodne zarządzanie bazami poprzez podłączenie odpowiedniego sterownika. Jednym z takich narzędzi jest SQuirreL SQL Client, na przykładzie którego zaprezentowano ten tryb pracy JShards.

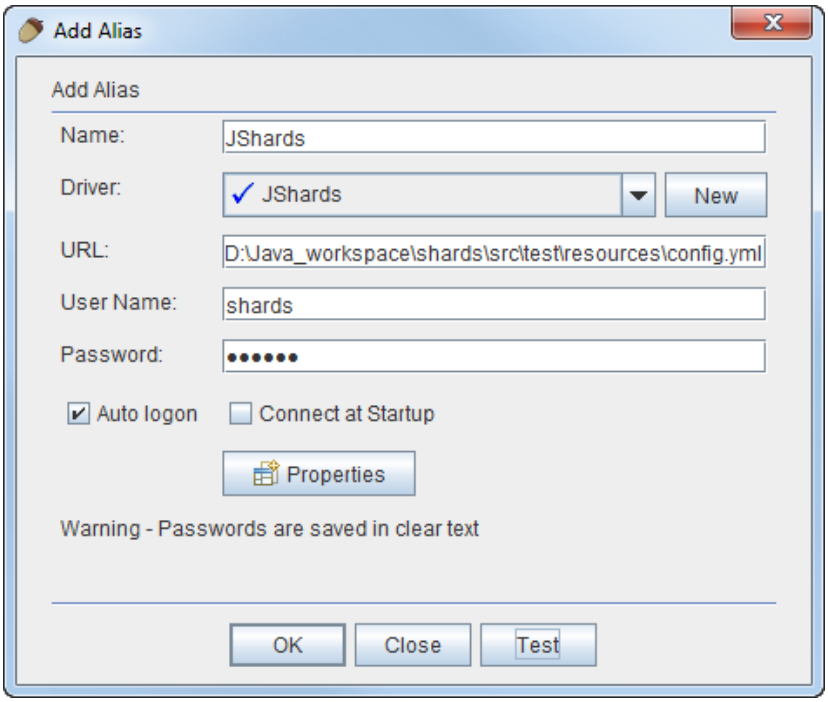
SQuirreL SQL Client jest wieloplatformowym, bezpłatnym narzędziem napisanym w języku Java. Instalacja programu przebiega standardowo. W ramach konfiguracji należy dodać sterownik JShards, pamiętając jednocześnie o wszystkich zależnościach dostarczonych w dystrybucji narzędzia oraz sterownikach wykorzystywanych baz danych. Okno dodawania nowego sterownika przedstawia Rysunek 22.

Konieczne jest także wybranie klasy ShardsDriver w polu Class Name.

Kolejnym krokiem jest dodanie aliasu, jak przedstawiono na Rysunek 13. Warto zwrócić uwagę, że w polu URL znajduje się bezwzględna ścieżka do pliku konfiguracyjnego. W pola User Name oraz Password można wprowadzić dowolną wartość.



Rysunek 22 Okno dodawania nowego sterownika



Rysunek 23 Okno dodawania aliasu

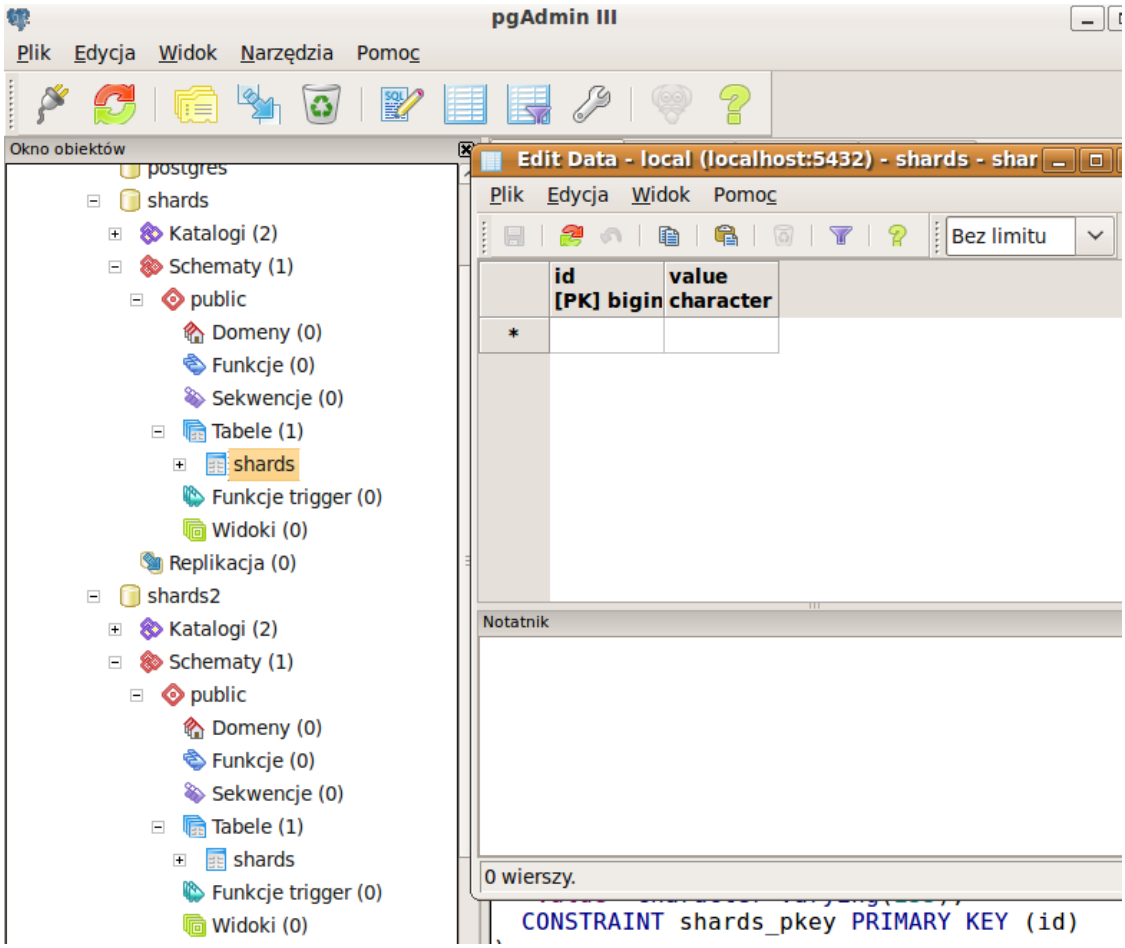
Konfiguracja programu została zakończona. W tej chwili warto przyjrzeć się środowisku bazodanowemu, na którym operować będzie narzędzie.

Aktywne są dwie bazy danych PostgreSQL. Na każdej znajduje się tabela shards o tej samej strukturze, przedstawionej na Listing 40.

```
Id [PK] bigi nt
Value character
```

Listing 40

Sytuację prezentuje Rysunek 24. Sterownik korzysta ze strategii, która zdefiniowana była w pliku konfiguracyjnym w Rozdziale 6.1.



Rysunek 24 Okno aplikacji pgAdmin. Widoczne są dwie bazy i analogiczne tabele na każdej z nich.

Wykonajmy teraz trzy zapytania wstawiające rekordy do bazy danych. Każde z nich wpisujemy po prostu do odpowiedniego pola w aplikacji SQuirreL (Rysunek 25) – w żaden sposób nie jest zaznaczony fakt, że poruszamy się w środowisku rozproszonym.

Jak widać na Rysunek 26, sterownik zadbał o rozdzielenie zapytań do odpowiednich shardów na podstawie wyniku działania strategii wyboru sharda zawartego na Listing 41.

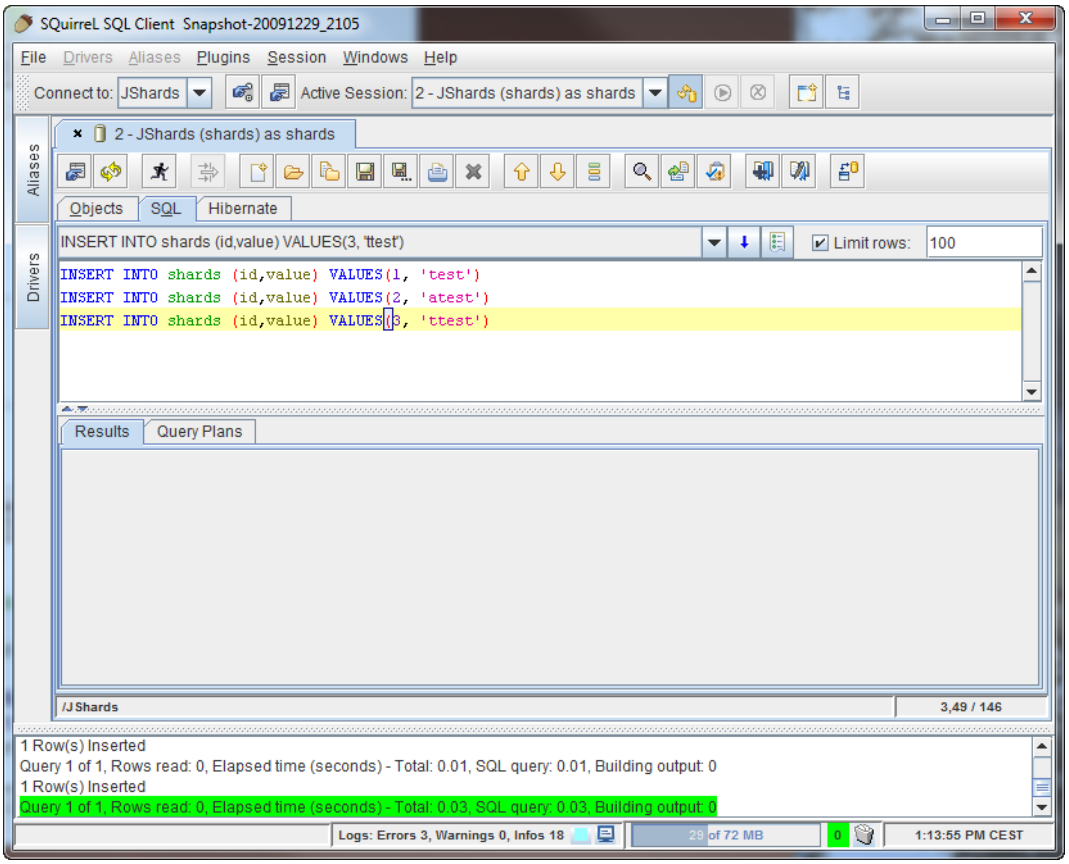
```
Rekordy o id <0;2> trafiają do sharda 1
Rekordy o id <3; +∞) trafiają do sharda 2
```

Listing 41 Wynik działania strategii wyboru sharda

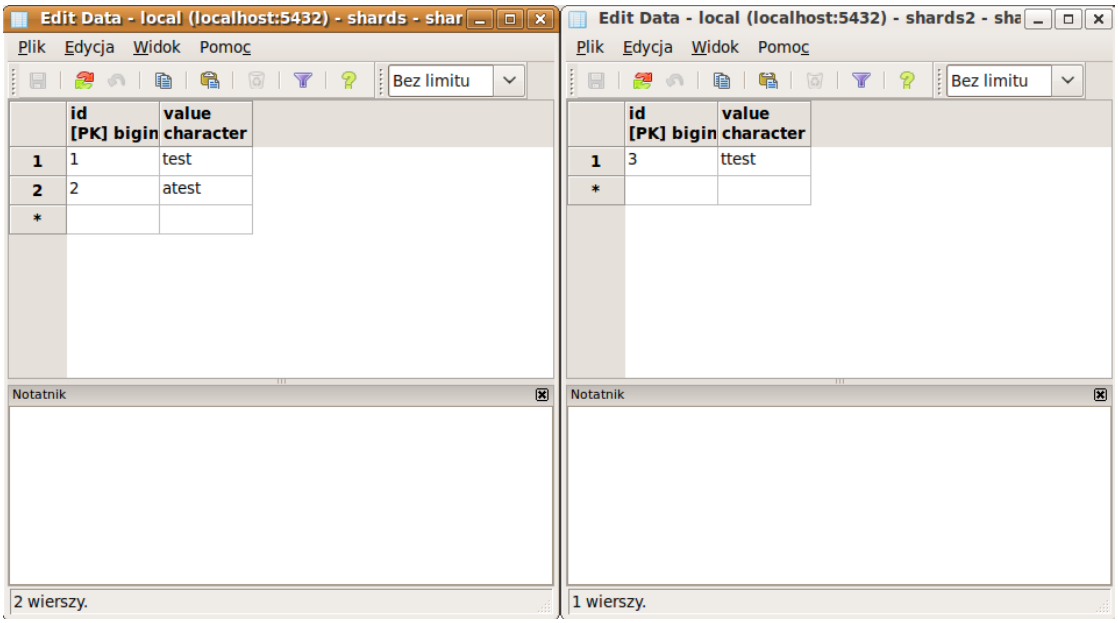
Pora teraz odpytać bazy o rekordy, które się na nich znajdują. Zgodnie z oczekiwaniami zapytanie z Listing 42 zwraca wszystkie trzy rekordy – Rysunek 27.

```
SELECT * FROM shards;
```

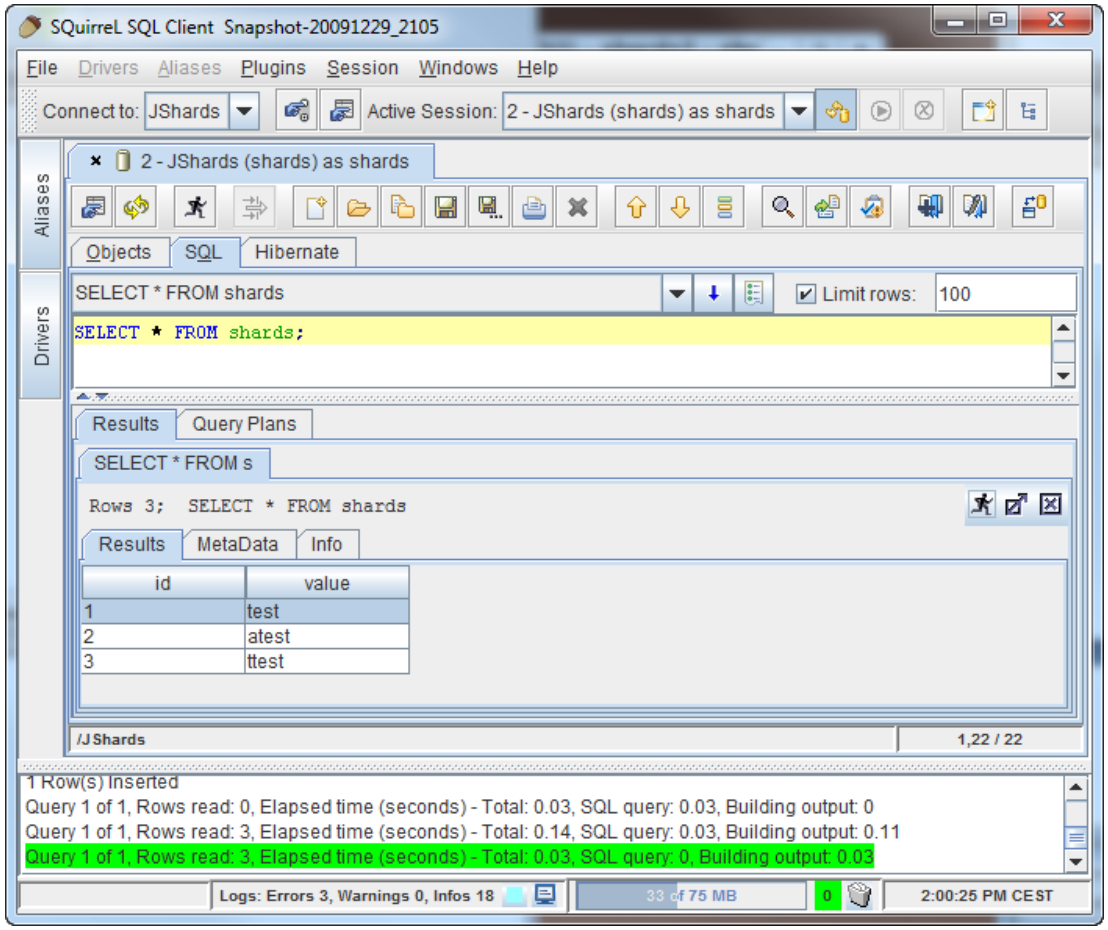
Listing 42



Rysunek 25 Wykonywanie zapytań aktualizujących w SQuirreL SQL Client



Rysunek 26 Zawartość tabel w obu bazach po wykonaniu poleceń aktualizujących



Rysunek 27 Wynik zapytania SELECT * FROM shards w SQuirreL SQL Client

Poznaliśmy już sposoby wykorzystania sterownika JShards. Pora teraz przedstawić proces konstruowania własnej strategii wyboru sharda.

6.4. Tworzenie własnej strategii wyboru sharda

Stworzenie strategii wyboru sharda sprowadza się do zaimplementowania interfejsu ShardsSelectionStrategy. Interfejs ten zawiera jedną metodę, przedstawioną na Listing 43.

```
Set<String> selectShards(ParameterInfo param);
```

Listing 43 Metoda selectShards interfejsu ShardsSelectionStrategy

Metoda ta wywoływana jest dla każdego zapytania tyle razy, ile porównywanych kolumn znajduje się w klauzuli WHERE tego zapytania, jak omówiono w Rozdziale 4.2. Jej zadaniem jest zwrócenie zbioru identyfikatorów shardów, na których zapytanie zostanie wykonane. Decyzja o szczegółach implementacyjnych wyboru należy do programisty tworzącego strategie.

Listing 44 zawiera klasyczną implementację interfejsu ShardsSelectionStrategy.

```
public class ClassicShardsSelectionStrategy implements
ShardsSelectionStrategy {
 private String column;

 public void setColumn(String column) {
       this.column = column;
 }

 public Set<String> selectShards(ParameterInfo param) {
       // implementacja metody odpowiedzialnej za wybór sharda
 }

 private Set<String> allShards() {
       // metoda zwraca wszystkie shardy powiązane z daną
strategią
 }
```

```
}
```

Listing 44 Szkielet implementacji interfejsu ShardsSelectionStrategy

Jednym z istotniejszych jej elementów jest pole column, które zawiera nazwę kolumny stanowiącej klucz strategii wyboru sharda. Z kolei metoda allShards() zwraca zbiór wszystkich shardów, które poprzez konfigurację powiązane będą z budowaną strategią. Zarówno z pola column, jak i metody allShards() korzysta metoda selectShards. Rozważmy zatem jej standardowy szkielet przedstawiony na Listing 45.

```
 public Set<String> selectShards(ParameterInfo param) {
       if (column.equals(param.getColumn())) { // (1)
             if (Type.NUMBER.equals(param.getType())) { // (2)
                Set<String> selectedShards = new …
                // wyliczenia związane z wyborem sharda
                // i dodanie wyników do kolekcji shardów
                return selectedShards;
             } else { // (3)
                throw new WrongShardsQueryException("Given value "
+ param.getValue() + " has to be a number");
             }
       }
       return allShards(); // (4)
 }
```

Listing 45 Szkielet metody selectShards

Pierwszym działaniem podjętym przez tę metodę jest zwykle (1) sprawdzenie, czy kolumna klucza strategii wyboru sharda jest tożsama z kolumną zawartą w obiekcie ParameterInfo, czyli znajdującą się w jednym z warunków wewnątrz klauzuli WHERE zapytania. Jeśli tak, (2) sprawdzany jest typ parametru porównywanego z wartością kolumny, ponieważ powinien być on zgodny z naturą strategii wyboru sharda, np. jeśli strategia operuje na przedziałach liczbowych, wartość porównywana powinna być liczbą. Sukces porównania powoduje przejście do etapu faktycznego wyliczania odpowiednich shardów, porażka natomiast (3) skutkuje wyrzuceniem wyjątku informującego o błędnym typie przekazanej w zapytaniu wartości. Ostatnia ewentualność ma miejsce, jeśli kolumna pochodząca z zapytania nie

pokrywa się z kluczem strategii wyboru sharda. Wtedy (4) zwracane są wszystkie shardy związane ze strategią. Szczegóły tej decyzji omówiono w Rozdziale 4.2.1.

Przedstawiony szkielet nie jest w żadnej mierze jedynym poprawnym rozwiązaniem. Stanowi jednak znakomity punkt wyjścia do tworzenia własnych strategii.

Oczywistym jest, że wyliczenia związane z wyborem sharda stanowią istotę procesu budowy własnej strategii. Z dużą dozą prawdopodobieństwa można założyć, że wartość przekazywana do metody selectShards() będzie porównywana z wartością pewnego pola klasy strategii i na tej podstawie zostanie podjęta decyzja, do których shardów skierować zapytanie. Podobnie, wartość pola column determinuje, która kolumna z klauzuli WHERE zapytania będzie traktowana jako klucz strategii wyboru sharda. Musi zatem istnieć sposób ustalania wartości pól klasy strategii bez konieczności modyfikacji kodu źródłowego.

Taką możliwość zapewnia plik konfiguracyjny. Otóż wszystkie atrybuty zawarte wewnątrz parametru params w sekcji strategy są bezpośrednio mapowane na właściwości klasy strategii wyboru sharda.

Jako przykład rozważmy plik konfiguracyjny z Listing 32, Rozdział 6.1. W tym przypadku klasa RangeShardsSelectionStrategy, której dotyczy konfiguracja, ma dwa pola – Listing 46.

```
String column;
List<ShardRange> ranges;
```

Listing 46

Każde pole posiada odpowiadający mu setter – Listing 47:

```
public void setColumn(String column) {
    this.column = column;
}

public void setRanges(List<Map<String, Object>> ranges) {
    for (Map<String, Object> range : ranges) {
       double from = getDouble(range, "from", Double.MIN_VALUE);
       double to = getDouble(range, "to", Double.MAX_VALUE);
       String shard = getString(range, "shard");
```

```
        addRange(from, to, shard);
    }
}
```

Listing 47

W trakcie analizy pliku konfiguracyjnego uruchomione zostaną obie metody. Skalary zostaną odwzorowane na obiekty odpowiednich typów Java zgodnie z [8]. W tym wypadku nazwa kolumny zostanie odwzorowana na obiekt klasy String. Z kolei lista słowników zostanie odwzorowana na listę map. W związku z tym, zadaniem metody setRanges jest transformacja tej listy na listę obiektów klasy ShardRange. Klasa ShardRange, na obiekty której mapowane są trójki <from, to, shard>, ma trzy pola przedstawione na Listing 48.

```
double from;
double to;
String shard;
```

Listing 48

Takie podejście pozwala na elastyczne inicjowanie pól klas reprezentujących strategie wyboru sharda.

Rozdział ten kończy omówienie zaproponowanego rozwiązania. W kolejnym rozdziale przedstawione zostaną pomysły dalszego rozwoju oraz trudności implementacyjne i koncepcyjne, które napotkano w fazie projektowania i implementacji.

7. Problemy realizacyjne i plany na przyszłość

Koncepcja będąca przedmiotem pracy stanowi złożone zagadnienie. W związku z czym podczas implementacji prototypu natknięto się na kilka problemów, z których najważniejszy to obsługa operacji złączenia w zapytaniach.

Nie wszystkie rozwiązania udało się też zaimplementować w pierwszej wersji prototypu. Zrównoleglenie wykonywania operacji na shardach i obsługa dużych ilości danych poza pamięcią operacyjną, to tylko niektóre plany rozbudowy sterownika JShards.

Wszystkie te zagadnienia stanowią przedmiot niniejszego rozdziału.

7.1. Złączenia

Operacja złączenia tabel (ang. JOIN) jest jedną z najbardziej pracochłonnych operacji w systemach zarządzania bazami danych. W przypadku środowiska rozproszonego, problem staje się jeszcze poważniejszy. Rozważmy dla przykładu zapytanie z Listing 49.

```
SELECT * FROM tabela1 LEFT JOIN tabela2 ON tabela1.id=tabela2.id
```

Listing 49

Zakładając, że zawartość obu tabel znajduje się na kilku – być może wszystkich – shardach, klasyczna implementacja tej operacji wymagałaby przekazania zapytania do wszystkich shardów, dla każdej tabeli. Daje to koszt rzędu wyrażonego Równanie 1.

```
Ilość zapytań = ilość tabel biorących udział w złączeniu * ilość
shardów, na których znajdują się te tabele
```

Równanie 1 Koszt operacji złączenia w środowisku shardów

Dodatkowo złączenie musiałoby się odbywać lokalnie, w pamięci. Wszystko to sprawia, że proces ten miałby niezwykle negatywny wpływ na wydajność całego rozwiązania, a przecież właśnie ten czynnik jest najistotniejszym w koncepcji skalowalnych horyzontalnie baz danych.

Odpowiedzią na ten problem są tabele globalne.

7.1.1 Tabele globalne

Jak czytamy w [9], tabele globalne to relatywnie statyczne, niezmienne tabele, których zawartość jest replikowana na wszystkich dostępnych shardach. Założenie o statycznym charakterze tabel wynika z nakładów, jakie należy ponieść na zapewnienie replikacji między shardami. Dlatego najlepszymi kandydatami na tabele globalne są tabele słownikowe, np. status, kraj, województwo, typ, itp. Z praktyki wynika także, że tabele słownikowe bardzo często występują w złączeniach.

Dzięki temu, że tabele te znajdują się na wszystkich shardach, koszt operacji złączenia z ich udziałem jest zdecydowanie niższy. Rozważmy przykład omówiony wcześniej. Zakładając, że jedna z tabel byłaby tabelą globalną, koszt operacji wyraża Równanie 2.

```
Ilość zapytań = ilość shardów, na których znajduje się te tabela shardowana
```

Równanie 2 Koszt operacji złączenia w środowisku shardów w przypadku wykorzystania tabeli globalnej

Co więcej, operacja złączenia pozostaje w zakresie odpowiedzialności silnika bazodanowego. Aplikacja jest jedynie odpowiedzialna za scalenie wyników zapytań z poszczególnych shardów. To wszystko sprawia, iż wydajność pozostaje na wysokim poziomie, przy jednoczesnym braku konieczności rezygnowania z funkcjonalności złączeń.

Dodatkowym nakładem, który należy ponieść podczas rozbudowywania sterownika JShards jest konieczność zaimplementowania mechanizmu replikacji danych pomiędzy tabelami globalnymi. Do tego celu można z powodzeniem wykorzystać strategię mirroringu opisaną w Rozdziale 4.2.3.

Z problemem złączeń wiąże się bezpośrednio zagadnienie aliasów. Aliasy, zarówno dla tabel, jak i dla kolumn są kolejnym etapem implementacji obsługi złączeń, który planuje się wprowadzić.

7.1.2 Tabele globalne, a wyrażanie binarne

Z obsługą tabeli globalnych wiąże się jeszcze jeden problem. Tabele te obsługiwane są przez strategię mirroringu, więc dowolne zapytanie, które w klauzuli WHERE zawierać będzie

odwołanie do kolumny tabeli globalnej będzie obsłużone przez tę strategię. To z kolei spowoduje – zgodnie z własnością strategii – zwrócenie dowolnego losowo wybranego sharda. Zachowanie to może mieć niespodziewane skutki, co obrazuje przykład z Listing 50.

```
SELECT * FROM tabela_shardowana LEFT JOIN tabela_globalna ON
tabela_shardowana.id=tabela_globalna.id WHERE tabela_shardowana.id
= 1 OR tabela_globalna.value = 'A'
```

Listing 50

Załóżmy, że strategia obsługująca tabelę shardowaną zwróci dla podanego wejścia identyfikator „shard1”. Z kolei strategia mirroringu obsługująca tabelę globalną, zwróci dowolny inny identyfikator sharda, np. „shard2”.

W tej sytuacji operacja sumy teoriomnogościowej zwróci zbiór {„shard1”, „shard2”}, a zatem zapytanie zostanie wykonane na obu shardach, mimo że dla jego wyliczenia wystarczające są dane, znajdujące się tylko na shardzie pierwszym.

Oczywiście sytuacja ta jest niepożądana ze względów wydajnościowych. Dlatego też planuje się implementację mechanizmu pomijania tabel globalnych przy wyliczaniu zbioru shardów, na których ma zostać uruchomione zapytanie.

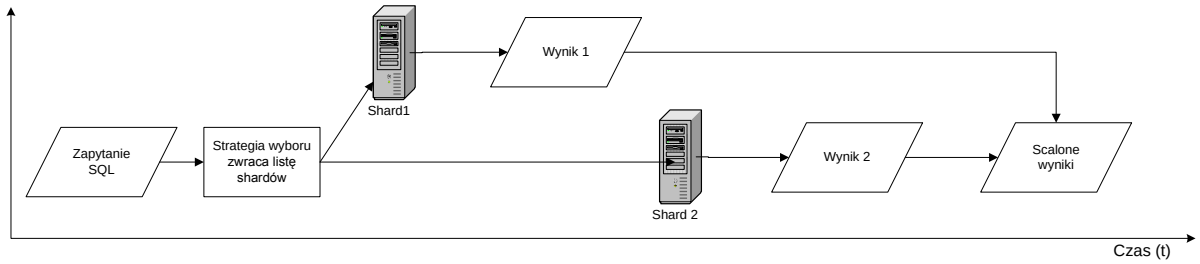
Istnienie wielu baz danych, do których trafiają zapytania, w sposób naturalny przywołuje na myśl zrównoleglenie całego procesu. Jest to kolejny etap na ścieżce rozwoju JShards.

7.2. Równoległe wykonywanie operacji na shardach

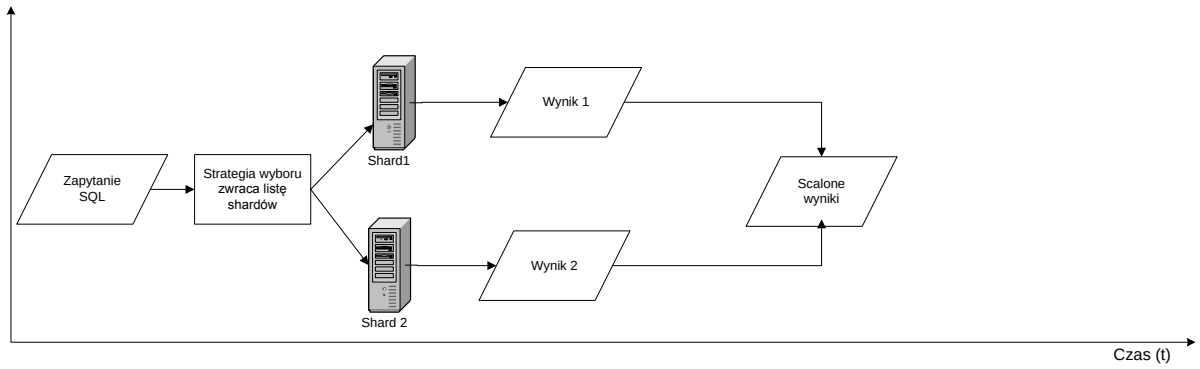
Na chwilę obecną poszczególne zapytania kierowane do shardów wykonywane są szeregowo. Duży wzrost wydajności zapewniłaby możliwość równoległego, wielowątkowego wykonywania operacji na shardach.

W przyszłości planuje się implementację wielowątkowej realizacji przesyłania zapytań do rozproszonych baz danych. Koncepcja ta zakłada hermetyzację operacji na każdym shardzie w osobnym wątku. Klient oczekiwał będzie na zakończenie działania wszystkich wątków, po czym – mając do dyspozycji wyniki cząstkowe – realizować będzie ich scalanie.

Rozwiązanie oparte o szeregowe wykonywanie operacji przedstawia Rysunek 28. Z kolei Rysunek 29 obrazuje koncepcję równoległego przetwarzania zapytań.



Rysunek 28 Szeregowe wykonywanie operacji na shardach



Rysunek 29 Równolegle wykonywanie operacji na shardach

Oczywiście wzrost wydajność będzie proporcjonalny do ilości shardów, do których trafi określone zapytanie. Aby uniknąć spowolnienia narzędzia przez dużą ilość pracujących wątków, planuje się wprowadzenie mechanizmu puli wątków.

Zrównoleglenie operacji wykonywanych na shardach nie rozwiąże jednak innego problemu – konieczności przetwarzania dużych ilości danych w pamięci operacyjnej.

7.3. Wykonywanie operacji na dużych ilościach danych w pamięci operacyjnej

Wyniki większości zapytań, które trafiły do poszczególnych shardów są opracowywane przez sterownik już po ich sprowadzeniu do pamięci operacyjnej. Dotyczy to przede wszystkim operacji grupowania i sortowania. Dopóki dane cząstkowe są niewielkich rozmiarów i

mieszczą się w pamięci operacyjnej, wszystko jest w porządku. Co jednak, gdy rozmiar sprowadzonych danych przekracza rozmiar dostępnej pamięci?

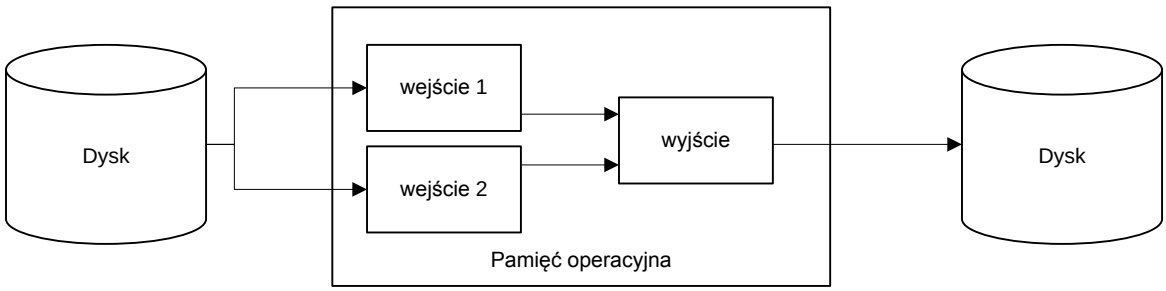
Aby rozwiązać ten problem planuje się zapisywanie cząstkowych wyników do pamięci trwałej, np. na twardy dysk, a następnie sukcesywne ich przetwarzanie. Cały proces nie będzie tak efektywny, jak gdyby wykonywany był w pamięci operacyjnej, jednak niekiedy jest to jedyne wyjście.

Konkretne rozwiązanie przedstawionego problemu przeanalizujemy na przykładzie sortowania. Sortowanie zbioru rekordów, który nie mieści się w pamięci RAM nazywa się sortowaniem zewnętrznym.

Składać się ono będzie z dwóch etapów:

1. Pobieranie porcji danych, która zmieści się w pamięci wewnętrznej, a następnie zapisywanie jej na dysk twardy. W tym momencie mamy pewność, że dane znajdujące się w tej porcji są już posortowane, ponieważ zapytanie skierowane do pojedynczego sharda musiało zawierać klauzulę ORDER BY.
2. Wielofazowe ładowanie rekordów danych z dysku, scalanie ich w sensie algorytmu sortowania przez scalanie i ponowny zapis na dysku.

Etap drugi jest powtarzany dopóki nie uzyska się pojedynczego, posortowanego zbioru danych. Algorytm ten nosi nazwę wielofazowego sortowania przez scalanie. Jego koncepcja została przedstawiona na Rysunek 30, natomiast szczegóły omówiono w [10].



Rysunek 30 Wielofazowe sortowanie przez scalanie

Kolejny podrozdział przedstawia modyfikacje związane z rozszerzeniem gramatyki parsera wyrażeń SQL.

7.4. Poprawa gramatyki parsera SQL

Drobne rozszerzenia i modyfikacje, które planuje się wprowadzić dotyczą przede wszystkim poprawy gramatyki parsera SQL.

Pierwszym krokiem będzie implementacja obsługi wartości boolowskich w zapytaniach, a także wsparcie dla strumieni i dużych obiektów (BLOB, CLOB) w konstrukcjach typu PreparedStatement.

Najistotniejszą jednak modyfikacją będzie wprowadzenie obsługi dialektów. Jak wiadomo, każdy system zarządzania bazą danych posiada swój specyficzny dialekt języka SQL. Różnice są często kosmetyczne, niemniej jednak mogą uniemożliwiać współpracę z daną bazą danych.

Główne obszary, w których uwidaczniają się odmienności to:

- Typy danych – w szczególności typy daty i czasu
- Funkcje – w szczególności funkcje operujące na napisach, np. operacja konkatenacji
- Zapytania – zwłaszcza zawierające polecenia ograniczenia ilości wyników

Szczegółowe różnice w dialektach SQL omówiono w [11].

Sterownik JShards został przystosowany i przetestowany we współpracy z silnikiem bazodanowym PostgreSQL. Implementacja obsługi dialektów pozwoli na wykorzystanie narzędzia z różnorodnymi bazami danych.

8. Podsumowanie

Ze względu na stosunkowo nowatorskie podejście do rozwiązania problemu wydajności aplikacji internetowych, implementacja sterownika JShards była wyzwaniem. Jednakże niedostatek istniejących rozwiązań i istotne potencjalne korzyści, skłoniły autorów do podjęcia próby realizacji tego zadania.

Kluczowym osiągnięciem tej pracy jest sprawny prototyp sterownika, który umożliwił przetestowanie tez postawionych na początku pracy, wedle których to system zarządzania bazą danych stanowi wąskie gardło aplikacji webowych pozostających pod dużym obciążeniem. Przyrost przepustowości na poziomie blisko 33% przy wykorzystaniu jedynie dwóch fizycznych baz danych należy niewątpliwie uznać za sukces. Ponadto można śmiało założyć, że dodanie kolejnych baz danych spowoduje dalszy wzrost przepustowości. Otwarta pozostaje kwestia osiągnięcia punktu krytycznego, czyli momentu, w którym dokładanie następnych shardów nie będzie powodowało wzrostu wydajności.

Z uwagi na niezwykle obszerny temat tej pracy, kilka istotnych problemów pozostaje nierozwiązanych. Kwestią zasadniczą jest obsługa złączeń w zapytaniach. Autorzy proponują wykorzystać mechanizm tabel globalnych, czyli tabel replikowanych na wszystkich shardach.

Następnym etapem prac nad sterownikiem będzie szukanie dalszych zysków wydajności. Z tego powodu rozważa się zrównoleglenie operacji wykonywanych na rozproszonych bazach danych. Duże porcje danych napływające z shardów prowadzą z kolei wprost do kolejnego problemu, który należy rozwiązać – przeniesienia części danych z pamięci operacyjnej do pamięci trwałej i sukcesywnego ich przetwarzania.

Rezultaty zachęciły autorów do dalszej pracy nad stworzonym prototypem. Dodatkowo oprogramowanie to zostanie udostępnione na zasadach open-source. Autorzy wierzą, że ten krok przyczyni się do dalszego zwiększenia jakości i popularności sterownika JShards.

9. Prace cytowane

1. **Andersen, Lance.** *JDBC 4.0 Specification.* Santa Clara, California : Sun Microsystems, 2006.

2. **Lewis, Jonathan.** *Cost-based oracle fundamentals.* New York : APRESS, 2006. 978-1590596364.

3. **Kyte, Thomas.** *Expert Oracle Database Architecture 9i and 10g Programming Techniques and Solutions.* New York : APRESS, 2005. 978-1590595305.

4. RAID. *Wikipedia.* [Online] 2009. http://pl.wikipedia.org/wiki/RAID.

5. **Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides.** *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, Massachusetts : Addison-Wesley, 2000. 978-0201633610.

6. YAML Ain't Markup Language (YAML™) Version 1.2. [Online] http://www.yaml.org/spec/1.2/spec.html.

7. **King, Gavin.** Kod źródłowy klasy org.hibernate.loader.Loader. *JDocs.* [Online] 2010. http://www.jdocs.com/hibernate/3.2.5/org/hibernate/loader/Loader.html.

8. Implicit types. [Online] http://code.google.com/p/snakeyaml/wiki/Documentation#Implicit_types.

9. **codeFutures.** *Database Sharding.* [Online] 2009. http://www.codefutures.com/database-sharding/.

10. **Banachowski, Lech i Stencel, Krzystzof.** *Systemy zarządzania bazami danych.* Warszawa : PJWSTK, 2007. strony 163-188. 83-89244-58-8.

11. SQL Dialects Reference. [Online] 2008. http://en.wikibooks.org/wiki/SQL_Dialects_Reference.