



WYDZIAŁ ELEKTRONIKI,
TELEKOMUNIKACJI
I INFORMATYKI

Imię i nazwisko studenta: Michał Cwynar

Nr albumu: 175132

Poziom kształcenia: Studia drugiego stopnia

Forma studiów: stacjonarne

Kierunek studiów: Informatyka

Specjalność: Aplikacje rozproszone i systemy internetowe

PRACA DYPLOMOWA MAGISTERSKA

Tytuł pracy w języku polskim: Automatyczne śledzenie i grupowanie transakcji bazodanowych w aplikacjach działających na maszynie wirtualnej Java (JVM)

Tytuł pracy w języku angielskim: Automated tracking and grouping of database transactions in Java Virtual Machine (JVM) applications

Opiekun pracy: dr inż. Krzysztof Manuszewski

STRESZCZENIE

Bazy danych są powszechnie stosowane w dzisiejszych aplikacjach. Gwarantują bezpieczeństwo danych za pomocą różnych mechanizmów, których niepoprawne użycie może przyczynić się do znacznego spowolnienia aplikacji. Dlatego kluczowe jest monitorowanie, a także mierzenie czasu wykonywania się transakcji i zapytań bazodanowych. Taka wiedza pozwoli administratorowi baz danych na szybkie zdiagnozowanie problemu w przypadku, gdy zapytanie lub zapytania wchodzące w skład transakcji wykonują się podejrzanie długo. W związku z tym celem niniejszej pracy magisterskiej było zbadanie możliwości takiej instrumentacji kodu bajtowego aplikacji, by być w stanie automatycznie śledzić i grupować transakcje bazodanowe. Omówione zostały również narzędzia już istniejące na rynku, a dzięki aplikacji testowej i eksperymentom możliwe stało się porównanie i przeanalizowanie wszystkich rozwiązań. Okazało się, że autorski agent wprowadza najmniejszy narzut zasobów, ale największy czasu. Głównym powodem jest sposób przesyłania danych do serwera. Natomiast jako jedyny jest w stanie rozróżniać to, czy zapytania są realizowane indywidualnie czy tworzą transakcję, a także umożliwia podejrzenie ich parametrów. W rezultacie rozwiązanie zaproponowane w tej pracy magisterskiej może konkurować z już istniejącymi narzędziami.

Słowa kluczowe: agent, instrumentacja, kod bajtowy, transakcja, baza danych

Dziedzina nauki i techniki, zgodnie z wymogami OECD:

Nauki o komputerach i informatyka

ABSTRACT

Databases are widely used in today's applications. They guarantee data security using various mechanisms, the incorrect use of which may significantly slow down the application. Therefore, it is crucial to monitor and measure the execution time of transactions and database queries. Such knowledge will allow the database administrator to quickly diagnose the problem if a query or queries included in a transaction take a suspiciously long time. Therefore, the aim of this master's thesis was to investigate the possibility of instrumenting the application's bytecode in such a way as to be able to automatically track and group database transactions. Tools already existing on the market were also discussed, and thanks to the test application and experiments, it was possible to compare and analyze all solutions. It turned out that the author's agent introduced the smallest resource overhead, but the largest time overhead. The main reason is how data is transferred to the server. However, it is the only one able to distinguish whether queries are executed individually or form a transaction, and also allows viewing their parameters. As a result, the solution proposed in this master's thesis can compete with already existing tools.

Keywords: agent, instrumentation, bytecode, transaction, database

Field of science and technology in accordance with OECD requirements:

Computer Science and Information technology

SPIS TREŚCI

SPIS TREŚCI.....	5
WYKAZ WAŻNIEJSZYCH OZNACZEŃ I SKRÓTÓW	6
1. WSTĘP I CEL PRACY	7
2. Instrumentacja kodu bajtowego języka programowania Java	9
2.1. Kod bajtowy	9
2.2. Przykłady programów wraz z ich kodem bajtowym	11
2.3. Instrumentacja kodu bajtowego	14
2.3.1. Instrumentacja za pomocą agenta dołączonego statycznie	15
2.3.2. Instrumentacja za pomocą agenta dołączonego dynamicznie	15
2.4. Biblioteka Javassist z przykładami	16
3. Przykłady agentów do śledzenia i grupowania transakcji bazodanowych w aplikacjach	21
3.1. Aplikacja testowa	21
3.2. OpenTelemetry	23
3.3. OneAgent	25
3.4. Datadog	27
4. Projekt własnego agenta	29
4.1. Architektura narzędzia	29
4.1.1. Sposób implementacji agenta	29
4.1.2. Sposób implementacji serwera	34
4.2. Wsparcie dla dowolnej bazy danych	36
4.3. Wyniki	37
5. Eksperymenty i porównania agentów	40
5.1. Plan testów	40
5.2. Wyniki i kierunki rozwoju	40
6. PODSUMOWANIE	43
WYKAZ LITERATURY	44
WYKAZ RYSUNKÓW	45
WYKAZ TABEL	46

WYKAZ WAŻNIEJSZYCH OZNACZEŃ I SKRÓTÓW

JVM - (ang. *Java virtual machine*), maszyna wirtualna Java

1. WSTĘP I CEL PRACY

Bazy danych są nieodłącznym elementem dzisiejszych aplikacji. Są to złożone systemy, które za pomocą różnych mechanizmów [1] gwarantują bezpieczeństwo danych m.in. w sytuacji, w której wielu klientów żąda dostępu do tego samego zasobu w tym samym czasie. Do takich mechanizmów można zaliczyć chociażby blokady współdzieloną i wyłączną. Pierwsza z nich pozwala wielu transakcjom co najwyżej na czytanie tabeli bądź tylko jej rekordów, na których została założona ta blokada. Natomiast druga daje dostęp na wyłączność tylko jednej transakcji do tabeli lub wiersza, dzięki czemu możliwe jest zarówno czytanie jak i modyfikowanie. Inny mechanizm to poziomy izolacji, które zarządzają tymi blokadami, tak aby transakcje mogły wykonywać się równolegle. Najniższy z nich nazywa się `READ_UNCOMMITTED` i nie dostarcza żadnych blokad, przez co możliwy jest nawet odczyt danych zaktualizowanych przez inne transakcje, nawet jeśli zmiany nie zostały jeszcze zatwierdzone, jest to tzw. brudny odczyt. Drugi poziom to `READ_COMMITTED`, który z kolei wprowadza już blokady i pozwala tylko na odczytywanie danych już zatwierdzonych. Kolejny nazywa się `REPEATABLE_READ` i blokuje zasoby na czas trwania całej transakcji uniemożliwiając niepowtarzalne odczyty, czyli sytuację, w której transakcja czyta dwa razy ten sam wiersz, ale dostaje inne dane za każdym razem. Ostatni poziom to `SERIALIZABLE` izolujący całkowicie transakcje od siebie, przez co w praktyce wykonywane są sekwencyjnie. Pomaga to zapobiec zjawisku odczytów widmo występujących, kiedy transakcja pobiera dwukrotnie rekordy z tej samej tabeli, ale dostaje różną ich liczbę. Łatwo więc zauważyć, że niepoprawne użycie tych mechanizmów może znacząco spowolnić aplikację. W związku z tym kluczowe jest monitorowanie, a także mierzenie czasu wykonywania się transakcji i zapytań bazodanowych. Dzięki temu np. administrator baz danych będzie w stanie szybko zdiagnozować problemy w przypadku, gdy zapytanie lub zapytania wchodzące w skład transakcji wykonują się podejrzanie długo. Jest to wtedy jasny sygnał świadczący o potrzebie optymalizacji poprzez np. zmianę poziomu izolacji transakcji.

Celem tej pracy magisterskiej jest zbadanie czy za pomocą procesu instrumentacji kodu bajtowego aplikacji działającej na maszynie wirtualnej Java (JVM), można automatycznie śledzić i grupować transakcje bazodanowe, które wykonuje ta aplikacja na swojej bazie danych. Oprócz tego zadaniem jest również mierzenie ich czasu. Głównym problemem do rozwiązania będzie znalezienie sposobu na rozróżnienie tego, czy zapytania są realizowane indywidualnie czy jednak tworzą transakcję, a następnie ich pogrupowanie. Założenie jest takie, że dwie transakcje są takie same jeśli różnią się tylko i wyłącznie w wartościach parametrów. Do celów można również zaliczyć zapoznanie się z już istniejącymi narzędziami na rynku i porównanie ich za pomocą różnych eksperymentów do rozwiązania zaproponowanego w tej pracy magisterskiej.

W ramach szczegółowego omówienia autorskiego narzędzia, niniejsza praca podzielona została na rozdziały, które w postępujący sposób wyjaśniają wszystkie zagadnienia związane ze zrozumieniem tego sposobu. W związku z tym układ pracy jest następujący:

Rozdział 2, "Instrumentacja kodu bajtowego języka programowania Java" - szczegółowy opis, wraz z praktycznymi przykładami, procesu instrumentacji kodu bajtowego klas, a także prezentacja biblioteki Javassist.

Rozdział 3, "Przykłady agentów do śledzenia i grupowania transakcji bazodanowych w aplikacjach" - kompleksowe omówienie narzędzi istniejących już na rynku zarówno otwartoźródłowych np.

OpenTelemetry jak i komercyjnych takich jak OneAgent czy Datadog.

Rozdział 4, "Projekt własnego agenta" - szczegółowe przedstawienie zaproponowanego rozwiązania wraz z wyjaśnieniem wszystkich jego komponentów poparte diagramami architektury i schematami blokowymi.

Rozdział 5, "Eksperymenty i porównania agentów" - porównanie i przeprowadzenie analizy wszystkich omówionych narzędzi za pomocą różnych eksperymentów.

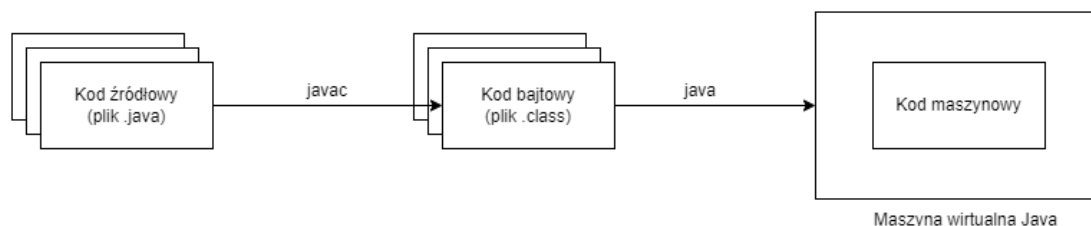
Rozdział 6, "Podsumowanie" - opis autorskiego narzędzia w odniesieniu do wyników eksperymentów oraz postawionych celów całej pracy magisterskiej, a także wskazanie potencjalnych kierunków rozwoju.

2. INSTRUMENTACJA KODU BAJTOWEGO JĘZYKA PROGRAMOWANIA JAVA

W niniejszym rozdziale zostały przedstawione pojęcia związane z kodem bajtowym języka programowania Java wraz z jego instrumentacją. Dodatkowo zaprezentowano także przykłady na różnych typach danych, aby dobrze zrozumieć omawiane zagadnienia, gdyż są to niezwykle ważne narzędzia wykorzystywane przy implementacji agentów. Oprócz tego została opisana jedna z najpopularniejszych bibliotek do instrumentacji kodu bajtowego, czyli Javassist.

2.1. Kod bajtowy

W języku programowania Java kod źródłowy aplikacji przechowywany jest w plikach z rozszerzeniem .java. W celu uruchomienia programu należy go najpierw skompilować za pomocą kompilatora. Można to osiągnąć z wykorzystaniem polecenia `javac`, które przetłumaczy klasy zapisane w plikach z rozszerzeniem .java na odpowiadający im kod bajtowy, który zostanie zapisany z rozszerzeniem .class. Ostatecznie aplikacja jest gotowa do wykonania za pomocą polecenia `java`, które uruchamia maszynę wirtualną Java tłumaczącą kod bajtowy na kod maszynowy zrozumiały dla procesora. Na poniższym rysunku został zobrazowany proces uruchamiania programów napisanych w języku programowania Java.



Rysunek 2.1: Proces uruchamiania programów napisanych w języku programowania Java. Źródło własne.

Na rysunku 2.1 widać, że kod bajtowy jest etapem pośrednim w tym procesie. Dzięki temu, posiadając jedynie pliki z rozszerzeniem .class, można uruchomić aplikację na dowolnym komputerze z zainstalowaną maszyną wirtualną Java bez dostępu do plików źródłowych. W związku z tym ten język programowania jest niezależny od platformy na jakiej jest uruchamiany.

Co ciekawe, plik z rozszerzeniem .class ma z góry zdefiniowaną strukturę, która została przedstawiona w tabelce 2.1. Można tam znaleźć opisy występujących po sobie pól, a także informację o liczbie zajmowanych przez nie bajtów. Pierwsze pole to "magiczna" liczba, od której zaczyna się każdy poprawny plik .class. Jej wartość w systemie szesnastkowym to CAFEBABE. Następne 4 bajty tworzą liczbę odpowiadającą wersji np. dla Javy 8 wersja major to 52, a minor to 0. Kolejne pole określa liczbę stałych w puli, a następne je definiuje. W tym miejscu przechowywane są pełne nazwy klas, sygnatury metod i stałe ciągi znaków czy liczb. Szóste pole to numer określający flagi dostępu np. 1 odpowiada za ACC_PUBLIC, czyli klasę publiczną, a 1024 za ACC_ABSTRACT, czyli klasę abstrakcyjną. Można łączyć flagi np. liczba 1025 definiuje dostęp jako publiczny i abstrakcyjny. Kolejne dwa pola informują o nazwach klasy pochodnej i bazowej. Dla przypomnienia każda klasa w języku programowania Java bezpośrednio lub pośrednio dziedziczy po klasie Object. Następne w kolejności są informacje dotyczące liczby interfejsów, pól czy metod klasy wraz z ich definicjami, czyli nazwami, sygnaturami,

typami danych czy flagami dostępu. W przypadku funkcji podaje się także zawartość ich ciał, czyli tzw. kod bajtowy składający się z instrukcji zrozumiałych dla maszyny wirtualnej Java. Przedostatnie pole odpowiada za liczbę, a ostatnie za definicję atrybutów np. adnotacje.

Tabela 2.1: Struktura pliku .class.

liczba bajtów	opis pola
4	magiczna liczba (ang. <i>magic number</i>)
2	wersja minor (ang. <i>minor version</i>)
2	wersja major (ang. <i>major version</i>)
2	liczba stałych w puli stałych (ang. <i>constant pool</i>)
*	poindeksowane stałe tworzące pulę stałych
2	liczba określająca flagi dostępu (ang. <i>access flags</i>)
2	indeks do nazwy klasy przechowywanej w puli stałych
2	indeks do nazwy klasy bazowej przechowywanej w puli stałych
2	liczba implementowanych interfejsów
*	definicja interfejsów
2	liczba pól klasy
*	definicja pól
2	liczba metod klasy
*	definicja metod
2	liczba atrybutów klasy
*	definicja atrybutów

* – liczba bajtów zdefiniowana przez poprzednie pole

Zatem kod bajtowy jest zestawem instrukcji zrozumiałych dla JVM. Warte podkreślenia jest to, że jego nazwa pochodzi od tego, że każde polecenie zapisane jest tylko za pomocą pojedynczego bajtu zwanego kodem operacji. W związku z tym mamy tylko 256 możliwych komend z czego 202 już wykorzystano, 3 zarezerwowano przez JVM, a 51 jest jeszcze wolnych dla nowych instrukcji [2]. Do każdego kodu operacji przypisany jest odpowiadający mu mnemonik, którego pierwsza lub ostatnia litera odnosi się do typu operandów, na których działa. W tabelce 2.2 zostały przedstawione możliwe przedrostki i przyrostki wraz z odpowiadającymi im typami argumentów, na których działa polecenie zaczynające się lub kończące na tej literze.

Tabela 2.2: Przedrostki i przyrostki mnemoników wskazujące na typ operandów.

przedrostek/przyrostek	typ operandów
i	typ całkowitoliczbowy 32-bitowy o nazwie <i>int</i>
l	typ całkowitoliczbowy 64-bitowy o nazwie <i>long</i>
s	typ całkowitoliczbowy 16-bitowy o nazwie <i>short</i>
b	typ całkowitoliczbowy 8-bitowy o nazwie <i>byte</i>
c	typ reprezentujący pojedynczy znak 16-bitowy o nazwie <i>char</i>
f	typ zmiennoprzecinkowy 32-bitowy o nazwie <i>float</i>
d	typ zmiennoprzecinkowy 64-bitowy o nazwie <i>double</i>
a	typ referencyjny 32-bitowy

Z kolei w tabelce 2.3 zaprezentowano przykłady kodów operacji w systemie binarnym i szesnastkowym wraz z odpowiadającymi im mnemonikami.

Tabela 2.3: Przykłady kodów operacji w systemie binarnym i szesnastkowym wraz z ich mnemonikami.

kod operacji w systemie binarnym	kod operacji w systemie szesnastkowym	mnemonik
0000 0011	03	iconst_0
0000 1000	08	iconst_5
0011 1011	3b	istore_0
0011 1110	3e	istore_3
0001 1010	1a	iload_0
0001 1101	1d	iload_3
0001 0000	10	bipush
1111 1110	fe	impdep1
1111 1111	ff	impdep2
1100 1010	ca	breakpoint

Ostatnie trzy podane mnemoniki w tabelce 2.3 są zarezerwowane dla programów typu Debugger w celu implementacji różnych funkcjonalności takich jak punkty przerwania.

2.2. Przykłady programów wraz z ich kodem bajtowym

Zaprezentowane kody bajtowe poniższych programów zostały wygenerowane dzięki narzędziu `javap`. Jest to deassembler, który przekształca pliki `.class` w formie binarnej na mnemoniki wraz z ich operandami i dodatkowymi informacjami. Ten program to potężne narzędzie udostępniające wiele opcji, w tym przypadku została użyta jedna: `-v`, która podaje również dane na temat rozmiaru stosu, liczby zmiennych lokalnych i argumentów funkcji.

Pierwszy omówiony przykład to pusta pętla wykonująca się 101 razy z iteratorem typu `int` pokazana w pseudokodzie 1. Kod bajtowy tego programu wygenerowany za pomocą narzędzia `javap` został zamieszczony na rysunku 2.2.

Algorytm 1 Pseudokod prezentujący pustą pętlę wykonującą się 101 razy z iteratorem typu `int`

```

1: void f() {
2:   int i;
3:   for i = 0; i <= 100; i++ do
4:     ;
5:   end for
6: }
```

Na rysunku 2.2 można zaobserwować, że przed samymi mnemonikami i ich operandami, znajdują się trzy dodatkowe informacje dla maszyny wirtualnej Java w kontekście tego jakie zasoby są potrzebne, aby uruchomić tę metodę 1. Pierwsza z nich to stos (ang. *stack*), mówi o tym, ile operandów będzie co najwyżej na stosie w czasie wykonywania tej metody. Druga z nich to liczba zmiennych lokalnych (ang. *locals*), która w tym przypadku wynosi dwa. Może wydawać się to dziwne, dlatego że w pseudokodzie 1 widać tylko jedną zmienną lokalną, czyli iterator pętli. Natomiast nie można zapomnieć o tym, że w języku programowania Java wszystkie niestacyjne metody są wirtualne i mają jeden niejawny

```

Code:
  stack=2, locals=2, args_size=1
    0: iconst_0
    1: istore_1
    2: iload_1
    3: bipush          100
    5: if_icmpgt        14
    8: iinc             1, 1
   11: goto              2
   14: return

```

Rysunek 2.2: Kod bajtowy odpowiadający programowi 1. Źródło własne.

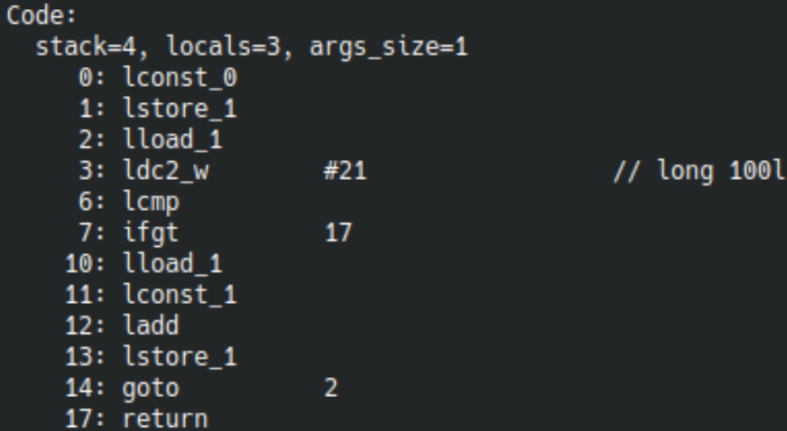
parametr, jakim jest słowo kluczowe *this*, czyli referencję do obiektu, którego funkcja została wywołana. Dodatkowo JVM traktuje parametry jako zmienne lokalne, dlatego właśnie ich liczba wynosi dwa. Trzecia informacja to liczba argumentów metody (ang. *args_size*), co już zostało poruszone, chodzi o słowo kluczowe *this*.

JVM jest przykładem maszyny zarówno stosowej jak i rejestrowej czego dowodem jest stos 32-bitowych operandów i zmienne lokalne, które można traktować jak rejestry również 32-bitowe. W związku z tym w rozważanym przypadku słowo kluczowe *this* jest rejestrem o numerze 0, a iterator pętli o indeksie 1. Zadaniem pierwszej instrukcji jest umieszczenie stałej 0 na stosie. Dla małych wartości od -1 do 5 JVM ma wbudowane funkcje do zapisu takich liczb bez potrzeby przekazywania żadnego operandu. Cel następnego mnemonika to pobranie stałej z wierzchołka stosu i przypisanie jej do rejestru numer 1, w którym przechowywany jest iterator pętli. Ponownie ta instrukcja jak i również następna nie potrzebują operandu, chyba że występowałyby więcej niż cztery zmienne lokalne. Trzecie polecenie to umieszczenie wartości z rejestru numer 1 z powrotem na stosie, czyli wierzchołkiem jest cyfra 0. Czwarta instrukcja przyjmuje operand o rozmiarze bajtu, czyli liczbę 100 i zapisuje ją na stosie, oczywiście na 32 bitach. Kolejny mnemonik odpowiada za pobranie i porównanie dwóch operandów będących na stosie. Jeśli pierwszy z nich będzie większy od wartości w wierzchołku to sterowanie zostanie przekazane do instrukcji spod indeksu 14., na co wskazuje argument, zapisany na dwóch bajtach, omawianej funkcji. W przeciwnym razie następny mnemonik będzie wykonany i też tak się stanie w rozważanym przypadku, gdyż 0 nie jest większe od 100. Zatem kolejna operacja to inkrementacja zmiennej lokalnej zapisanej w rejestrze numer 1, czyli iteratora pętli, o wartość 1. W następnym kroku instrukcja o nazwie *goto* przekazuje sterowanie do mnemoniku spod indeksu 2., o czym mówi operand zapisany na dwóch bajtach. W związku z tym można zaobserwować, że operacje od indeksu 2. do 11. tworzą pętlę, która polega na porównaniu zmiennej lokalnej o numerze 1 z liczbą 100, jeśli będzie ona większa to sterowanie zostanie przekazane do instrukcji spod indeksu 14., w przeciwnym razie wykonana będzie kolejna operacja. Łatwo więc zauważyć, że jak iterator pętli osiągnie wartość 101, to mnemonik o nazwie *return* zostanie wykonany, który sprawi, że metoda się zakończy. Warto zwrócić też uwagę na indeksy kodu bajtowego widoczne po lewej stronie mnemoników i ich operandów. Dzięki temu można się dowiedzieć, ile bajtów zajmują te operacje wraz z ich argumentami, w tym przypadku jest to 15, z czego na mnemoniki przeznaczone jest po 1 bajcie, a operandy mogą zajmować co najwyżej 2 bajty.

Przykład 2 jest podobny do poprzedniego 1, z tym że iterator pętli nie jest typu *int*, a *long*. Kod bajtowy odpowiadający temu przypadkowi, wygenerowany za pomocą narzędzia *javap*, został zamieszczony na rysunku 2.3.

Algorytm 2 Pseudokod prezentujący pustą pętlę wykonującą się 101 razy z iteratorem typu *long*

```
1: void f() {  
2: long i;  
3: for i = 0; i <= 100; i++ do  
4:   ;  
5: end for  
6: }
```



The image shows a screenshot of Java bytecode code. It starts with 'Code:' followed by stack and local variable information: 'stack=4, locals=3, args_size=1'. The instructions are: 0: lconst_0, 1: lstore_1, 2: lload_1, 3: ldc2_w #21 // long 100l, 6: lcmp, 7: ifgt 17, 10: lload_1, 11: lconst_1, 12: ladd, 13: lstore_1, 14: goto 2, 17: return.

Rysunek 2.3: Kod bajtowy odpowiadający programowi 2. Źródło własne.

Pierwsza widoczna różnica w porównaniu do kodu bajtowego na rysunku 2.2 jest taka, że mnemoniki zaczynają się od litery *l*, dlatego że ich operandy, czyli iterator pętli jak i również liczba 100 są traktowane jako typ *long*. Kolejna różnica jest zauważalna w parametrach mówiących o zasobach jakie są wymagane, aby wykonać tę metodę. Jak już zostało powiedziane, rejestry do przechowywania zmiennych lokalnych są 32-bitowe. To samo dotyczy stosu, który udostępnia miejsca w pamięci zajmujące również 32 bity. W poprzednim przypadku iterator pętli był typu *int*, który potrzebuje akurat 32 bity, dlatego żeby go przechować potrzebny był tylko jeden rejestr. W sytuacji, w której ta zmienna jest typu *long*, wymagane są dwa rejestry, żeby ją zapisać. Tak samo jeśli chodzi o stos operandów. Iterator pętli i liczba 100, z racji tego że są traktowane jako typ *long*, zajmą po 2 miejsca, dlatego rozmiar stosu jest ustawiony na 4 (ang. *stack*), a liczba zmiennych lokalnych na 3 (ang. *locals*) na rysunku 2.3. Warto tutaj wspomnieć o tym, że tylko typy *long* i *double* potrzebują dwóch rejestrów czy dwóch miejsc na stosie operandów. Wszystkie pozostałe typy mieszczą się na 32 bitach. Nawet jeśli zmienna lokalna byłaby typu *short* to zajmowałaby jeden rejestr i jedno miejsce na stosie, natomiast JVM dbałby o to, żeby tylko 16 z 32 bitów było wykorzystywanych np. za pomocą mnemoniku *i2s*, które zeruje niepotrzebne bity.

Sposób działania pierwszych trzech mnemoników jest taki sam jak w poprzednim przykładzie 2.2. Natomiast czwarta instrukcja do umieszczenia liczby 100 na stosie się różni. Wcześniej w tym miejscu użyta była funkcja o nazwie *bipush*, która przyjmuje operand, w tym przypadku 100, w postaci jednego bajtu i zapisuje go na stosie na 32 bitach. Nie ma podobnej instrukcji, która mogłaby zrzuć taki operand od razu na typ *long*, czyli na 64 bity. W związku z tym JVM musi sobie poradzić inaczej. Otóż,

wykorzystuje do tego celu wspomnianą już pulę stałych, czyli jedno z pól z pliku .class, które służy do przechowywania pełnych nazw klas, sygnatur metod i stałych ciągów znaków czy liczb niezbędnych do uruchomienia kodu klasy, której ten plik .class dotyczy. Każda zapisana tam stała jest indeksowana i można ją pobrać za pomocą następujących operacji:

- *ldc* (operandem jest indeks stałej mieszczący się na 1 bajcie),
- *ldc_w* (operandem jest indeks stałej mieszczący się na 2 bajtach),
- *ldc2_w* (operandem jest indeks stałej tylko typu *long* lub *double* mieszczący się na 2 bajtach).

W rozważanym przypadku użyta jest wersja o nazwie *ldc2_w*, z racji tego, że celem jest umieszczenie liczby 100 typu *long* na stosie. Operandem jest indeks z puli stałych, pod którym jest zapisana ta wartość. Następne polecenie do porównania iteratora pętli i liczby 100 też się różni. W poprzednim przykładzie instrukcja warunkowa przyjmowała te wartości w postaci dwóch operandów i od razu decydowała o kolejności przepływu sterowania programem. W tym przypadku jest trochę inaczej, ze względu oczywiście na typ operandów, a mianowicie te porównanie podzielone jest na dwa etapy. Po pierwsze użyta jest funkcja o nazwie *lcmp*, która pobiera te argumenty ze stosu. Jak pierwszy z nich jest większy od drugiego, czyli od wierzchołka, to umieszczana jest wartość 1 na stosie, jeśli operandy są równe to będzie to 0, w przeciwnym razie -1. W aktualnym przypadku na stosie pojawi się -1, dlatego że 0 nie jest większe od 100. Drugi etap tego porównania to instrukcja spod indeksu 7., która dokonuje decyzji o kolejności przepływu sterowania programem na podstawie rezultatu z poprzedniej operacji. Jeśli w wierzchołku stosu jest liczba większa od 0, porównanie się powiedzie, w przeciwnym razie sterowanie zostanie przekazane następnej instrukcji. Z racji tego, że w wierzchołku stosu znajduje się wartość -1 to porównanie się nie powiedzie. Kolejne cztery operacje polegają na umieszczeniu iteratora pętli i liczby 1 na stosie, dodaniu ich i zapisaniu wyniku. Jest to więc odpowiednik inkrementacji zmiennej lokalnej zapisanej tym razem w dwóch rejestrach zaczynając od tego z numerem 1 o wartość 1. W następnym kroku instrukcja o nazwie *goto* przekazuje sterowanie do mnemoniku spod indeksu 2., o czym mówi operand zapisany na dwóch bajtach. W związku z tym ponownie można zauważyć, że operacje od indeksu 2. do 14. tworzą pętlę, która polega na porównaniu zmiennej lokalnej o numerze 1 z liczbą 100, jeśli będzie ona większa to sterowanie zostanie przekazane do instrukcji spod indeksu 17., w przeciwnym razie wykonana będzie kolejna operacja. Łatwo więc zauważyć, że jak iterator pętli osiągnie wartość 101, to mnemonik o nazwie *return* zostanie wykonany, który sprawi, że metoda się zakończy.

Podsumowując, kod bajtowy z przykładu 2.3 zajął o trzy bajty więcej niż z rysunku 2.2. Ponadto różnica w zasobach była również zauważalna. Operandy na stosie zajmowały dwa razy więcej miejsca, a iterator pętli był zapisany na dwóch rejestrach. Różnicą w tych dwóch programach był tylko typ zmiennej lokalnej. W związku z tym można powiedzieć, że wybrany typ danych ma istotny wpływ na zasoby pamięciowe oraz używane operacje na poziomie kodu bajtowego.

2.3. Instrumentacja kodu bajtowego

Instrumentacja to inaczej modyfikacja kodu bajtowego metod klas przed lub już po ich załadowaniu do maszyny wirtualnej Java za pomocą tzw. agentów. Celem jest dostarczenie dodatkowych funkcjonalności lub zmiana już istniejących. Instrumentacja wykorzystywana jest w wielu rodzajach aplikacji. Jednym z najpopularniejszych zastosowań są narzędzia służące do monitorowania, które modyfikują kod bajtowy, tak aby zapewnić różne metryki. Pierwszym przykładem są agenty do profilowania aplikacji, które dbają o ich wydajność, mierząc m.in. czas i częstotliwość wykonywania się metod czy

wykorzystanie pamięci. Takim narzędziem jest np. JProfiler. Drugie zastosowanie to agenty, których celem jest mierzenie pokrycia kodu (ang. *Code Coverage*) jak np. JaCoCo. Ta grupa rozwiązań informuje o tym, jaki procent kodu źródłowego został uruchomiony w czasie wykonania testów automatycznych. Kolejny i ostatni przykład dotyczy agentów przeznaczonych do logowania ważnych informacji z aplikacji np. parametrów metod. Jak widać instrumentacja to ciekawe i szeroko stosowane narzędzie [3].

Jak już zostało wspomniane modyfikację kodu bajtowego można przeprowadzić przed i po załadowaniu klas do maszyny wirtualnej Java. Pierwszy sposób wiąże się ze statycznym, a drugi dynamicznym, dołączeniem agenta do aplikacji źródłowej, której kod bajtowy ma być modyfikowany.

2.3.1. Instrumentacja za pomocą agenta dołączonego statycznie

Ten sposób w środowisku maszyny wirtualnej Java wymaga utworzenia dwóch plików z rozszerzeniem .jar (ang. *Java Archive*) dla agenta i aplikacji źródłowej, której kod bajtowy będzie modyfikowany. Jak sama nazwa wskazuje plik .jar to archiwum w formacie ZIP, które zawiera skompilowane pliki .class, a także pliki statyczne jak np. zdjęcia czy raporty z podsumowaniem przeprowadzonych testów automatycznych. W takim archiwum może się znaleźć także plik z rozszerzeniem .mf, czyli manifest zawierający metadane np. atrybut o nazwie Main-Class wskazujący na ścieżkę do klasy z metodą o nazwie *main*.

Po utworzeniu plików .jar, możliwe jest uruchomienie aplikacji wraz z agentem, dzięki poleceniu `java` z odpowiednimi parametrami, które można zaobserwować poniżej:

```
java -javaagent:agent.jar -jar aplikacja.jar
```

Pierwszy parametr —*javaagent* umożliwia podanie ścieżki do pliku .jar agenta, a drugi —*-jar* uruchamia aplikację, czyli metodę o nazwie *main*, która jest punktem startowym prawie każdego programu w języku programowania Java. Natomiast w przypadku agentów metoda odpowiadająca za punkt wejścia wygląda trochę inaczej, a mianowicie może przyjąć jeden z dwóch poniższych wariantów:

```
public static void premain(String agentArgs, Instrumentation inst)
```

```
public static void premain(String agentArgs)
```

Podczas uruchamiania agenta podanego za pomocą parametru —*javaagent*, JVM najpierw postara się znaleźć pierwszą metodę o nazwie *premain* z dwoma argumentami, jeśli nie będzie ona zdefiniowana, JVM wybierze wtedy drugi wariant tej metody. Parametr o nazwie *agentArgs* umożliwia podanie dodatkowych argumentów agentowi. Natomiast *inst* to klasa *Instrumentation* pochodząca z pakietu o nazwie *instrument* [4], która udostępnia zestaw metod pozwalających modyfikować kod bajtowy. Warto zaznaczyć, że *premain* zostanie wykonany przed funkcją *main*. To co jest jeszcze istotne, aby poprawnie uruchomić agenta, to dodanie w jego pliku manifestowym atrybutu o nazwie *Premain-Class*, który powinien zawierać ścieżkę do klasy z metodą o nazwie *premain*.

Ogromną zaletą instrumentacji kodu bajtowego za pomocą agenta dołączonego statycznie jest brak jakiegokolwiek dodatkowej konfiguracji oraz możliwość uruchomienia aplikacji źródłowej wraz z agentem.

2.3.2. Instrumentacja za pomocą agenta dołączonego dynamicznie

Dynamiczne dołączenie agenta z kolei przeznaczone jest do modyfikowania kodu bajtowego aplikacji źródłowych już uruchomionych na maszynie wirtualnej Java. Zatem znajduje zastosowanie w sytuacjach, w których program jest wdrożony już na jakimś środowisku np. deweloperskim czy nawet

produkcyjnym i wystąpiła potrzeba dodania jakiejś funkcjonalności związanej z metrykami czy logami. W takim przypadku ten sposób to odpowiednie narzędzie do wprowadzenia takich zmian. Mimo to instrumentacja kodu bajtowego za pomocą agenta dołączonego dynamicznie w środowisku maszyny wirtualnej Java wymaga również utworzenia dwóch plików z rozszerzeniem .jar (ang. *Java Archive*). Pierwszy z nich to tak jak w poprzednim sposobie agent, natomiast drugi to program, którego zadaniem jest dołączenie agenta do już uruchomionej aplikacji źródłowej. W pseudokodzie 3 został przedstawiony sposób dodania agenta do dowolnej działającej maszyny wirtualnej Java.

Algorytm 3 Kod pozwalający na dodanie agenta do dowolnej aplikacji już uruchomionej na maszynie wirtualnej Java [5]

```
1: VirtualMachine jvm = VirtualMachine.attach(pid);
2: jvm.loadAgent(agent.jar);
3: jvm.detach();
```

W pseudokodzie 3 można zaobserwować, że kluczem jest wykorzystanie obiektu klasy *VirtualMachine* i jego metod. Funkcja o nazwie *attach* pozwala na połączenie się do dowolnej maszyny wirtualnej Java identyfikowanej przez unikatowy identyfikator procesu (ang. *Process ID* lub *PID*), który przekazywany jest jako parametr tej metody. Następna funkcja o nazwie *loadAgent* dołącza agenta do maszyny, z którą uzyskano połączenie. Parametrem jest ścieżka do pliku .jar agenta. Z kolei ostatnia funkcja o nazwie *detach* zamyka te połączenie. Taką aplikację można uruchomić za pomocą tego samego polecenia co w poprzednim sposobie, ale już bez opcji *-javaagent*.

java -jar aplikacja.jar

Jeśli chodzi o samego agenta to w nim również trzeba dokonać kilku zmian, aby był w stanie dynamicznie zmodyfikować kod bajtowy metod aplikacji źródłowej. Po pierwsze, metoda odpowiedzialna za punkt wejścia agenta wygląda trochę inaczej niż w poprzednim sposobie, a mianowicie inna jest nazwa tej funkcji, która również może przyjąć jeden z dwóch poniższych wariantów:

```
public static void agentmain(String agentArgs, Instrumentation inst)

public static void agentmain(String agentArgs)
```

Zasada wyboru jednego z powyższych wariantów przez maszynę wirtualną Java jest taka sama jak w przypadku metod o nazwie *premain*, czyli priorytet to znalezienie definicji pierwszego wariantu z dwoma argumentami, a następnie z jednym. Oczywiście, funkcja o nazwie *agentmain* zostanie wykonana po funkcji *main*. Po drugie, aby poprawnie dodać agenta do aplikacji, wymagane jest umieszczenie w jego pliku manifestowym atrybutu tym razem o nazwie *Agent-Class*, który powinien zawierać ścieżkę do klasy z metodą o nazwie *agentmain*.

Niewątpliwą zaletą instrumentacji za pomocą agenta dołączonego dynamicznie jest możliwość dodania go do dowolnej już wdrożonej aplikacji np. na jakimś środowisku. Dzięki temu istnieje możliwość wprowadzenia różnych funkcjonalności bez potrzeby restartu aplikacji. Natomiast do wad można zaliczyć wymóg stworzenia dodatkowego programu, którego celem jest tylko dołączenie agenta do odpowiedniej maszyny wirtualnej Java, na której działa pożądana aplikacja.

2.4. Biblioteka *Javassist* z przykładami

Jak już zostało wspomniane język programowania Java dostarcza pakiet o nazwie *instrument* umożliwiający modyfikowanie kodu bajtowego. Jedną z klas jest *Instrumentation* wraz z funkcją o na-

zwie *addTransformer*. Przyjmuje ona parametr będący interfejsem o nazwie *ClassFileTransformer*, który posiada dwie domyślne metody o nazwie *transform*. Wywołanie funkcji *addTransformer* rejestruje klasę implementującą ten interfejs, dzięki czemu metody *transform* wykonywane są dla każdej zdefiniowanej klasy w aplikacji. Ich sygnatury można zobaczyć poniżej:

```
byte[] transform(ClassLoader loader, String className, Class classBeingRedefined,  
                ProtectionDomain protectionDomain, byte[] classfileBuffer)
```

```
byte[] transform(Module module, ClassLoader loader, String className, Class classBeingRedefined,  
                ProtectionDomain protectionDomain, byte[] classfileBuffer)
```

Parametry tych metod opisują dane kolejnych klas wykorzystywanych w aplikacji, po kolei są to:

- klasa o nazwie *Module* reprezentująca moduł, z którego pochodzi klasa,
- klasa o nazwie *ClassLoader* odpowiedzialna za załadowanie klasy do maszyny wirtualnej Java,
- nazwa klasy,
- klasa o nazwie *Class* reprezentująca klasę, dla której wywoływana jest funkcja *transform*,
- klasa o nazwie *ProtectionDomain* opisująca uprawnienia do pliku klasy,
- tablica bajtów reprezentująca plik *.class* klasy.

Zatem metoda *transform* to miejsce, w którym proces instrumentacji się odbywa. Ostatni parametr tej funkcji to plik z rozszerzeniem *.class* klasy w postaci tablicy bajtów. W związku z tym można go dowolnie modyfikować np. dodać albo zmienić funkcjonalności metod klasy, a następnie zwrócić podmieniając tym samym istniejącą do tej pory definicję tej klasy. Jednak nie jest to wygodny sposób, dlatego że wiąże się z modyfikacją tablicy bajtów, a więc też znajomością kodów operacji czy podawaniem operandów w bajtach. Na szczęście, istnieje wiele bibliotek, które ułatwiają znacznie te zadanie, np. *ByteBuddy*, *ByteMan*, *ASM* czy *BCEL*. Jednak do jednych z najpopularniejszych i najprzyjemniejszych należy *Javassist*. Ta biblioteka umożliwia wykonanie tego zadania na poziomie samego języka programowania Java, co sprawia, że jest bardzo prosta w użyciu, gdyż nie wymaga nawet znajomości struktury pliku *.class* czy mnemoników.

W celu zaprezentowania działania biblioteki *Javassist* został stworzony agent, który statycznie zmodyfikował kod bajtowy funkcji przedstawionej w pseudokodzie 1, która na potrzeby przykładu została umieszczona w klasie o nazwie *Application*. Z kolei implementacja interfejsu o nazwie *ClassFileTransformer* wraz z redefinicją jednej z metod *transform* została przedstawiona w pseudokodzie 4.

Pierwszym poleceniem funkcji *transform* w pseudokodzie 4 jest instrukcja warunkowa sprawdzająca czy klasa, dla której wywoływana jest ta metoda, nazywa się *Application*. Jeśli nie, to zwracana jest od razu istniejąca już do tej pory definicja klasy. W przeciwnym razie sterowanie przechodzi do instrukcji pochodzących z biblioteki *Javassist* znajdujących się wewnątrz bloku instrukcji warunkowej. W pierwszej linii następuje pobranie obiektu klasy o nazwie *ClassPool* z domyślnymi ustawieniami. *ClassPool* przedstawia pulę ze wszystkimi zdefiniowanymi klasami w aplikacji przy pomocy tablicy hashowanej, w której klucze to ich nazwy, a wartości to klasy je reprezentujące o nazwie *CtClass*. Dlatego drugim poleceniem jest pobranie z puli obiektu przedstawiającego klasę o nazwie *Application*, a trzecim jej metody o nazwie *f* z pseudokodu 1. Klasa *CtMethod* dostarcza wiele funkcji umożliwiających instrumentację poprzez podanie kodu w języku programowania Java jako łańcuch znaków w parametrze, który jest automatycznie tłumaczony na kod bajtowy. Można to zaobserwować w liniach nr 7 i 8, w których dodawane jest wyświetlanie tekstu na początku i końcu tej metody. Ostatnia instrukcja bloku to podmienienie istniejącej już definicji klasy *Application* na nową.

Algorytm 4 Klasa implementująca interfejs o nazwie `ClassFileTransformer` wraz z metodą *transform*

```
1: class Transformer implements ClassFileTransformer {
2:     byte[] transform(ClassLoader loader, String className, Class classBeingRedefined, Protec-
        tionDomain protectionDomain, byte[] classfileBuffer) {
3:         if (className.equals("Application")) {
4:             ClassPool classPool = ClassPool.getDefault();
5:             CtClass ctClass = classPool.get("Application");
6:             CtMethod ctMethod = ctClass.getDeclaredMethod("f");
7:             ctMethod.insertBefore("System.out.println(\"Początek metody\");");
8:             ctMethod.insertAfter("System.out.println(\"Koniec metody\");");
9:             return ctClass.toBytecode();
10:        }
11:        return classfileBuffer;
12:    }
13: }
```

Ostatnim krokiem w celu poprawnego zaimplementowania agenta, który w sposób statyczny zmodyfikuje kod bajtowy, jest oczywiście zdefiniowanie funkcji *premain*, która została umieszczona na potrzeby przykładu w klasie o nazwie `Agent`. Można to zaobserwować w pseudokodzie 5.

Algorytm 5 Klasa agenta z metodą *premain*

```
1: class Agent {
2:     public static void premain(String agentArgs, Instrumentation inst) {
3:         inst.addTransformer(new Transformer());
4:     }
5: }
```

W tej metodzie została również wywołana funkcja *addTransformer* w celu rejestracji klasy implementującej interfejs `ClassFileTransformer` 4. Tym samym agent wraz z aplikacją, czyli klasą o nazwie `Application`, jest gotowy do uruchomienia według instrukcji dotyczącej instrumentacji kodu bajtowego za pomocą agenta dołączonego statycznie 2.3.1.

Rezultatem programu jest oczywiście wyświetlenie tekstu "Początek metody", pętla wykonująca się 101 razy, a także napis "Koniec metody". Jako dowód został wygenerowany kod bajtowy zmodyfikowanej przez agenta funkcji o nazwie `f` z pseudokodu 1 za pomocą narzędzia `javap`.

Na rysunku 2.4 można zaobserwować, że ta modyfikacja faktycznie przebiegła pomyślnie. Instrukcje od indeksu 8. do 19. to są te same mnemoniki czy operandy odpowiadające za wykonanie pętli 101 razy, które są widoczne na rysunku 2.2. Natomiast polecenia przed i po to dodatkowe funkcjonalności dodane przez agenta, czyli wyświetlenie napisów "Początek metody" i "Koniec metody".

Innym przykładem zastosowania biblioteki `Javassist` jest agent, który zamiast dostarczania nowych funkcjonalności, zmienia już istniejące. Jedyną różnicą względem poprzedniego przykładu to logika metody *transform* klasy implementującej interfejs `ClassFileTransformer`, którą można zobaczyć w pseudokodzie 6.

W pseudokodzie 6 można zaobserwować, że jedyną zmianą to zastąpienie dwóch metod *insertBefore* i *insertAfter* na jedną *setBody*, która nadpisuje dotychczasową funkcjonalność funkcji na nową przekazaną w parametrze. Proces uruchamiania agenta wraz z aplikacją nie różni się od poprzedniego

```

Code:
  stack=5, locals=4, args_size=1
    0: getstatic      #25          // Field java/lang/System.out:Ljava/io/PrintStream;
    3: ldc             #27          // String Początek metody
    5: invokevirtual   #33          // Method java/io/PrintStream.println:(Ljava/lang/String;)V
    8: iconst_0
    9: istore_1
   10: iload_1
   11: bipush          100
   13: if_icmpgt       22
   16: iinc            1, 1
   19: goto            10
   22: goto            25
   25: aconst_null
   26: astore_3
   27: getstatic      #25          // Field java/lang/System.out:Ljava/io/PrintStream;
   30: ldc             #35          // String Koniec metody
   32: invokevirtual   #33          // Method java/io/PrintStream.println:(Ljava/lang/String;)V
   35: return

```

Rysunek 2.4: Kod bajtowy odpowiadający zmodyfikowanemu programowi 1 przez agenta 4. Źródło własne.

Algorytm 6 Klasa implementująca interfejs o nazwie `ClassFileTransformer` wraz z metodą *transform*

```

1: class Transformer implements ClassFileTransformer {
2:     byte[] transform(ClassLoader loader, String className, Class classBeingRedefined, Protec-
        tionDomain protectionDomain, byte[] classfileBuffer) {
3:         if (className.equals("Application")) {
4:             ClassPool classPool = ClassPool.getDefault();
5:             CtClass ctClass = classPool.get("Application");
6:             CtMethod ctMethod = ctClass.getDeclaredMethod("f");
7:             ctMethod.setBody("System.out.println(\"Definicja metody\");");
8:             return ctClass.toBytecode();
9:         }
10:        return classfileBuffer;
11:    }
12: }

```

przykładu.

```

Code:
  stack=2, locals=1, args_size=1
    0: getstatic      #20
    3: ldc             #22
    5: invokevirtual   #28
    8: return

```

Rysunek 2.5: Kod bajtowy odpowiadający zmodyfikowanemu programowi 1 przez agenta 6. Źródło własne.

Rezultatem aplikacji jest tylko wyświetlenie tekstu "Definicja metody", co widać na rysunku 2.5, który przedstawia kod bajtowy zmodyfikowanej funkcji o nazwie `f` z pseudokodu 1. Widać, że dotychczasowa funkcjonalność tej metody, czyli pętla wykonująca się 101 razy, została całkowicie zastąpiona przez agenta na nową, jaką jest wyświetlenie napisu "Definicja metody".

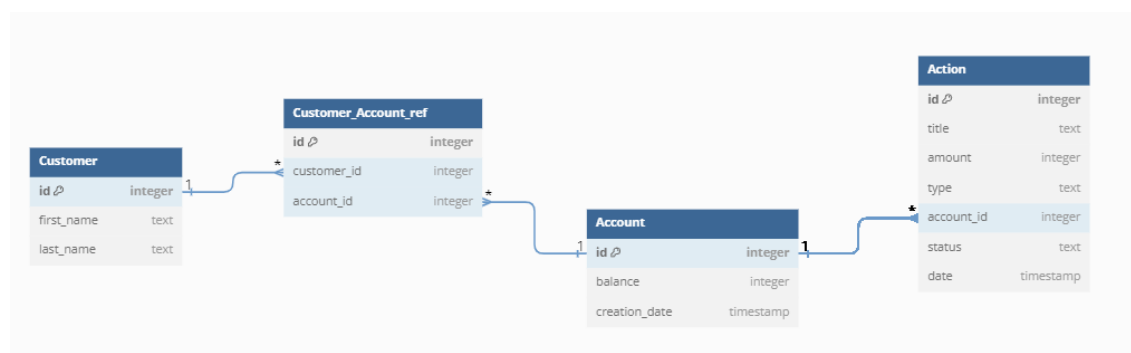
Podsumowując, biblioteka Javassist to potężne narzędzie, które pomaga w implementacji agentów do różnych celów. Jednym z nich jest także tytułowe śledzenie i grupowanie transakcji bazodanowych w aplikacjach. Przykłady takich agentów zostały omówione w następnym rozdziale.

3. PRZYKŁADY AGENTÓW DO ŚLEDZENIA I GRUPOWANIA TRANSAKCJI BAZODANOWYCH W APLIKACJACH

W niniejszym rozdziale zostały przedstawione istniejące rozwiązania agentów, które za pomocą instrumentacji kodu bajtowego profilują aplikację w celu dostarczenia informacji o jej wydajności m.in. mierząc czas wykonywania się zapytań czy transakcji bazodanowych. W związku z tym w pierwszym podrozdziale została zaprezentowana aplikacja testowa, która w kolejnych podrozdziałach będzie modyfikowana przez kolejne przykłady agentów np. OpenTelemetry, OneAgent czy Datadog. Dzięki temu w łatwy i przejrzysty sposób będzie można porównać istniejące rozwiązania.

3.1. Aplikacja testowa

Jest to prosta aplikacja bankowa napisana w języku programowania Java w wersji 17 wraz z użyciem platformy Spring Boot i silnika szablonów stron internetowych Thymeleaf. Dane aplikacji przechowywane są w bazie danych PostgreSQL w postaci czterech tabel, które można zaobserwować na diagramie przedstawionym na rysunku 3.1.



Rysunek 3.1: Diagram bazy danych aplikacji testowej. Źródło własne.

Customer, pierwsza z tabel przedstawionych na diagramie 3.1, służy przechowywaniu takich danych o klientach jak imię i nazwisko. Kolejna Account zawiera informacje o koncie bankowym w postaci salda i daty utworzenia. Tabela Customer_Account_ref łączy klienta z jego kontem bankowym. Natomiast ostatnia z nich Action zawiera dane na temat różnych przelewów bankowych w postaci tytułu, kwoty, typu (dostępne opcje to przelew krajowy, przelew walutowy i BLIK), konta, na które przelewane są środki, statusu czy daty wykonania przelewu.

Aplikacja umożliwia wykonanie różnych zapytań czy transakcji bazodanowych za pomocą interfejsu użytkownika napisanego przy wykorzystaniu silnika szablonów stron internetowych Thymeleaf, który można zobaczyć na rysunku 3.2.

Interfejs użytkownika dostarcza funkcjonalności za pomocą różnych formularzy. Niektóre z nich składają się tylko z przycisku, a inne wymagają również podania danych. W każdym z przypadków żądanie użytkownika przekierowywane jest do odpowiedniego punktu końcowego udostępnianego dzięki platformie Spring Boot. Obsługa żądania zrealizowana jest w języku programowania Java w wersji 17 wraz z wykorzystaniem interfejsu JDBC (ang. *Java DataBase Connectivity*) w celu nawiązania komuni-

Select all accounts

Select all customers

Select all actions

Create a new account

Create a new action

Create a random transaction

Title:

Amount:

Type:

Account id:

Status:

Create a new action

Rysunek 3.2: Interfejs użytkownika aplikacji testowej. Źródło własne.

kacji z bazą danych. W tabelce 3.1 zostały przedstawione kolejne nazwy przycisków, odpowiadające im punkty końcowe oraz przykładowe zapytania, które są wykonywane w czasie obsługi danego żądania. Warto zauważyć, że ostatni przycisk podany w tabelce 3.1 odpowiada za realizację dwóch zapytań.

Tabela 3.1: Przykładowe zapytania wykonywane w czasie obsługi żądania związanego z danym przyciskiem i jego punktem końcowym.

nazwa przycisku	punkt końcowy	przykładowe zapytania
Select all accounts	/selectAllAccounts	SELECT * FROM account
Select all customers	/selectAllCustomers	SELECT * FROM customer
Select all actions	/selectAllActions	SELECT * FROM action
Create a new account	/createNewAccount	SELECT * FROM account, INSERT INTO account (<i>id</i> , <i>balance</i> , <i>creation_date</i>) VALUES(85, 100, '2024-04-06 01:32:12+02')

Jak już zostało wspomniane aplikacja umożliwia również wykonanie transakcji bazodanowych za pomocą dwóch ostatnich przycisków umieszczonych w interfejsie użytkownika na rysunku 3.2. Transakcja to sekwencja jednego lub większej liczby zapytań traktowana jako jedna operacja. Innymi słowy nie ma możliwości, że tylko część transakcji zostanie zrealizowana, albo wykonają się wszystkie zapytania wchodzące w jej skład, albo żadne z nich. Co więcej, istnieje nawet zbiór właściwości zwany ACID gwarantujący poprawne działanie transakcji w bazach danych. Nazwa to akronim od:

- atomowość (ang. *atomicity*) - mówi o tym, że transakcja jest niepodzielna, albo wykona się w całości, albo wcale,
- spójność (ang. *consistency*) - gwarantuje, że wprowadzone zmiany przez transakcje nie naruszają integralności danych, czyli że zostaną zachowane wszystkie dotychczasowe więzy integralności (ang. *constraints*),
- izolacja (ang. *isolation*) - zapewnia, że każda transakcja jest niezależna od innej, istnieją różne poziomy izolacji w zależności od użytej bazy danych mówiące o tym w jakim stopniu jedna transakcja ma współpracować z drugą w kontekście możliwości odczytywania lub modyfikowania wierszów, na których działają,
- trwałość (ang. *durability*) - informuje o tym, że po zakończeniu transakcji wszystkie zmiany wprowadzone przez nią są zapisane.

W tej aplikacji testowej wyróżniamy dwa rodzaje transakcji składające się z dwóch lub trzech zapytań w zależności od wybranego typu przy wypełnianiu ostatniego formularza odpowiedzialnego za wysyłanie przelewów. Przycisk związany z tym formularzem przekierowuje na punkt końcowy `/create-NewAction`. W przypadku przelewu krajowego i BLIK transakcja składa się z dwóch zapytań, przykłady zostały zaprezentowane poniżej:

- `INSERT INTO Action (title, amount, type, account_id, status, date)`
`VALUES('Przelew za zakupy', 500, 'Przelew krajowy', 10, 'done', '2024-04-07 12:30:17.0')`
`UPDATE Account SET balance=balance+500 WHERE id=10`
- `INSERT INTO Action (title, amount, type, account_id, status, date)`
`VALUES('Przelew za obiad', 50, 'BLIK', 5, 'done', '2024-04-07 12:32:24.0')`
`UPDATE Account SET balance=balance+50 WHERE id=5`

Natomiast w przypadku przelewu walutowego transakcja składa się z trzech zapytań np.:

- `INSERT INTO Action (title, amount, type, account_id, status, date)`
`VALUES('Prowizja za przelew walutowy: Przelew za wakacje', 2, 'Przelew walutowy', 9, 'done', '2024-04-07 12:35:07.0')`
`INSERT INTO Action (title, amount, type, account_id, status, date)`
`VALUES('Przelew za wakacje', 2500, 'Przelew walutowy', 9, 'done', '2024-04-07 12:35:07.0')`
`UPDATE Account SET balance=balance+2500 WHERE id=9`

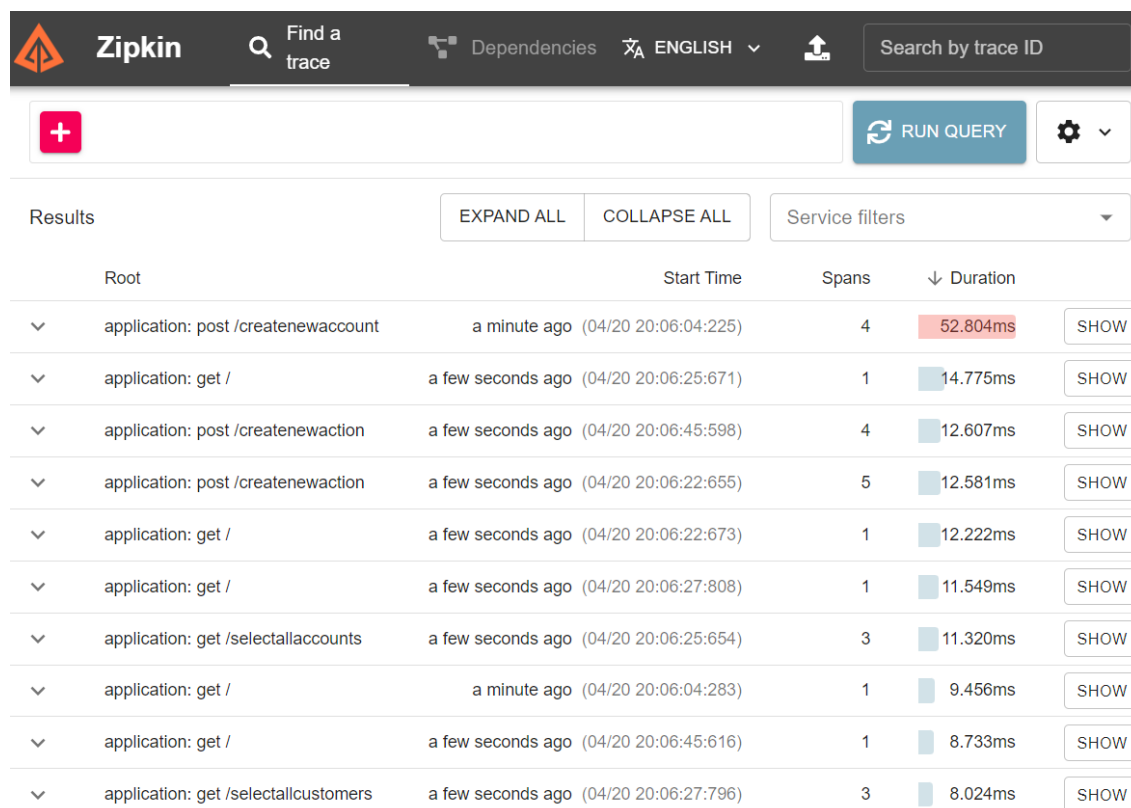
W interfejsie użytkownika znajduje się jeszcze przycisk przekierowujący na punkt końcowy `/create-RandomTransaction`, który odpowiada za natychmiastowe stworzenie transakcji o typie wylosowanym z takim samym prawdopodobieństwem bez potrzeby wypełniania ostatniego formularza.

3.2. OpenTelemetry

OpenTelemetry to pierwszy przykład otwartoźródłowego agenta, który za pomocą instrumentacji kodu bajtowego, dostarcza wielu informacji dotyczących wydajności aplikacji. Kod źródłowy tego narzędzia jest publicznie dostępny i można go znaleźć na platformie GitHub [6]. OpenTelemetry posiada także kompleksowe wsparcie dla zewnętrznych serwisów jak np. Zipkin czy Jaeger pozwalających na zwizualizowanie zebranych danych. Wystarczy skorzystać z odpowiednich parametrów, żeby połączyć agenta z takim serwisem [7].

W celu zaprezentowania działania OpenTelemetry wykorzystana została aplikacja testowa omówiona w podrozdziale 3.1. Do agenta został podłączony również serwis Zipkin w celu zwizualizowania

zgromadzonych informacji. Przykład polegał na wykonaniu kilku różnych interakcji z interfejsem użytkownika aplikacji testowej. Na rysunku 3.3 można zaobserwować zebrane dane w czasie tych interakcji przez OpenTelemetry, które zostały przesłane do serwisu Zipkin w celu ich prezentacji.



Root	Start Time	Spans	Duration	
application: post /createnewaccount	a minute ago (04/20 20:06:04:225)	4	52.804ms	SHOW
application: get /	a few seconds ago (04/20 20:06:25:671)	1	14.775ms	SHOW
application: post /createnewaction	a few seconds ago (04/20 20:06:45:598)	4	12.607ms	SHOW
application: post /createnewaction	a few seconds ago (04/20 20:06:22:655)	5	12.581ms	SHOW
application: get /	a few seconds ago (04/20 20:06:22:673)	1	12.222ms	SHOW
application: get /	a few seconds ago (04/20 20:06:27:808)	1	11.549ms	SHOW
application: get /selectallaccounts	a few seconds ago (04/20 20:06:25:654)	3	11.320ms	SHOW
application: get /	a minute ago (04/20 20:06:04:283)	1	9.456ms	SHOW
application: get /	a few seconds ago (04/20 20:06:45:616)	1	8.733ms	SHOW
application: get /selectallcustomers	a few seconds ago (04/20 20:06:27:796)	3	8.024ms	SHOW

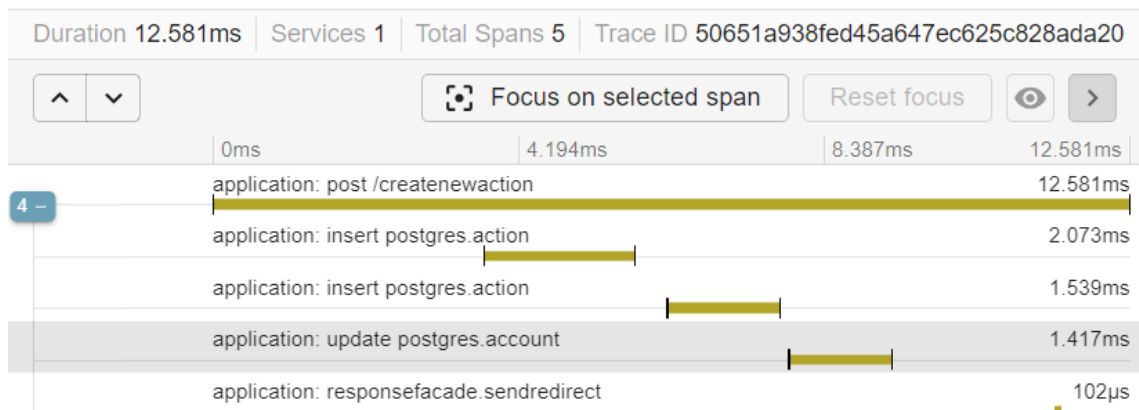
Rysunek 3.3: Serwis Zipkin prezentujący zgromadzone dane przez OpenTelemetry w czasie interakcji z interfejsem użytkownika aplikacji testowej. Źródło własne.

Na rysunku 3.3 można zauważyć, że agent zarejestrował wykonane punkty końcowe związane z żądaniami użytkownika, a także m.in. czas ich realizacji. Dodatkowo w trzeciej kolumnie zaprezentowana została informacja o liczbie operacji wchodzących w skład obsługi danego żądania. Istnieje możliwość podejrzenia szczegółów zarejestrowanego punktu końcowego za pomocą przycisku znajdującego się w ostatniej kolumnie. Przykład został przedstawiony na rysunku 3.4.

W szczegółach żądania widocznych na rysunku 3.4 można zauważyć pięć operacji wraz z kolejnością i czasem w jakim się wykonały. Pierwszą jest punkt końcowy z informacją o jego metodzie HTTP, w tym przypadku POST. Kolejne trzy operacje to są zapytania do bazy danych wysłane w trakcie obsługi tego żądania. Z podrozdziału 3.1 wiadomo, że punkt końcowy /createNewAction w przypadku przelewu walutowego wykonuje transakcję składającą się z trzech zapytań, które są właśnie widoczne w postaci tych operacji. Z rysunku 3.4 można również się dowiedzieć, że pierwsze zapytanie wykonywało się najdłużej, następnie drugie, a z kolei trzecie najkrócej. Natomiast ostatnia operacja z rysunku 3.4 to przekierowanie użytkownika na stronę startową aplikacji testowej. Istnieje również możliwość podejrzenia szczegółów każdej z tych operacji, w których można znaleźć więcej informacji jak np. dane o przeglądarce internetowej, w której użytkownik uruchomił aplikację testową. Natomiast w przypadku zapytań bazodanowych w szczegółach można odkryć ich rozwiniętą wersję, ale pozbawioną parametrów. Dla drugiej operacji z rysunku 3.4 te rozwinięte zapytanie wygląda w następujący sposób:

```
INSERT INTO Action (title, amount, type, account_id, status, date) VALUES(?, ?, ?, ?, ?, ?)
```


application: post /createnewaction



Rysunek 3.4: Operacje wykonane podczas realizacji punktu końcowego /createNewAction. Źródło własne.

Niestety, w szczegółach nie ma oryginalnych parametrów zapytań. Co więcej, nie ma także żadnej informacji o tym, czy zapytania wykonywane w trakcie obsługi tego punktu końcowego z rysunku 3.4 faktycznie tworzą transakcję bazodanową czy jednak są realizowane indywidualnie. Jest to niewątpliwa wada tego agenta, gdyż wiedza o tym może pomóc w szybkim zdiagnozowaniu problemów w przypadku, gdy zapytanie lub zapytania wchodzące w skład transakcji wykonują się podejrzanie długo.

Podsumowując, OpenTelemetry umożliwia rozległe monitorowanie wydajności aplikacji od samych punktów końcowych do zapytań do bazy danych. Udostępnia również szczegółowe dane na temat każdej z zarejestrowanych operacji. Natomiast nie jest w stanie zebrać informacji o oryginalnych parametrach w zapytaniach, a także poinformować o tym, czy tworzą transakcję czy są wykonywane indywidualnie.

3.3. OneAgent

Kolejny przykład to komercyjne narzędzie o nazwie OneAgent zarządzane przez firmę Dynatrace [8]. Ten agent składa się z wielu wyspecjalizowanych procesów zbierających różne informacje o wydajności maszyny, na której OneAgent został zainstalowany, a także aplikacjach na niej uruchomionych [9]. Najważniejsze metryki są następnie udostępniane na stronie internetowej dostarczonej przez Dynatrace.

Przykład służący zaprezentowaniu działania narzędzia OneAgent, w kontekście śledzenia i grupowania transakcji bazodanowych, został przeprowadzony w taki sam sposób jak dla poprzedniego agenta. To co odróżnia narzędzie OneAgent od poprzednika jest możliwość rejestrowania zapytań i transakcji bazodanowych niezależnie od punktów końcowych. Ponadto aktualnie omawiany agent przyporządkowuje operacje wykonywane na bazie danych do trzech różnych grup, są to:

- zapytania do modyfikowania danych,
- zapytania do odczytywania danych,
- transakcje.

Na rysunku 3.5 można zobaczyć te grupy wraz z dodatkowymi informacjami, czyli medianą czasów odpowiedzi, współczynnikiem niepowodzeń oraz liczbą wykonanych zapytań z tej grupy na minutę.

Pora przyjrzeć się każdej grupie z osobna. Na rysunku 3.6 została pokazana pierwsza z nich, która przedstawia zarejestrowane zapytania do modyfikowania danych, czyli w tym przypadku do wstawiania

Statement types

Contains 3 Request types.

Select a type to analyze executed database statements in detail

Name ↕	Response time [Median] ↕	Failure rate [Average] ↕	Throughput ↕	Details
SQL Modifications	10.6 ms	20 %	5 /min	✓
SQL Queries or Procedures	17.4 ms	0 %	4 /min	✓
SQL Transactions	4.28 ms	0 %	4 /min	✓

Rysunek 3.5: Grupy, do których OneAgent przyporządkowuje zarejestrowane zapytania. Źródło własne.

i aktualizowania rekordów w bazie danych. Oprócz tego prezentowany jest także średni czas całkowity i odpowiedzi, a także średni czas odpowiedzi dla 10% najwolniejszych zapytań. Ostatnia kolumna dostarcza natomiast szczegółowych informacji i dodatkowych funkcjonalności jak wizualizacja danych na wykresie.

3 database statements

Create analysis view

Filter on statement, database name, vendor, process and hostname.				
Statement ↕	Total time ▼	Response time ↕	Slowest 10 % ↕	Actions
Insert into account	271 µs/min	32.5 ms	32.5 ms	✓
Insert into action	239 µs/min	14.3 ms	25.5 ms	✓
Update account	60.9 µs/min	3.66 ms	4.86 ms	✓

Rysunek 3.6: Zapytania wchodzące w skład pierwszej grupy. Źródło własne.

W tym przypadku, tak samo jak w poprzednim, istnieje również możliwość podejrzenia w szczegółach rozwiniętej wersji zapytań, ale też niestety pozbawionej parametrów. Dla pierwszego zapytania z rysunku 3.6 jego rozwinięta wersja wygląda w następujący sposób:

```
INSERT INTO account (id, balance, creation_date) VALUES(?, ?, ?)
```

Z kolei na rysunku 3.7 została przedstawiona druga grupa, czyli zapytania do odczytywania danych, których rozwinięta wersja jest analogiczna do tej przedstawionej na rysunku.

Natomiast rysunek 3.8 ukazuje ostatnią grupę, czyli transakcje bazodanowe zarejestrowane przez narzędzie OneAgent.

Co ciekawe, na rysunku 3.8 nie widać żadnych transakcji czy zapytań je tworzących. Okazuje się, że agent zarejestrował tylko instrukcję o nazwie Commit, która odpowiedzialna jest za zapisanie

3 database statements

[Create analysis view](#)

Filter on statement, database name, vendor, process and hostname.				
Statement	Total time	Response time	Slowest 10 %	Actions
SELECT * FROM account	1.06 ms/min	63.7 ms	120 ms	
SELECT * FROM action	128 µs/min	15.4 ms	15.4 ms	
SELECT * FROM customer	59.7 µs/min	7.16 ms	7.16 ms	

Rysunek 3.7: Zapytania wchodzące w skład drugiej grupy. Źródło własne.

1 database statement

[Create analysis view](#)

Filter on statement, database name, vendor, process and hostname.				
Statement	Total time	Response time	Slowest 10 %	Actions
SQL Commit	70.7 µs/min	2.12 ms	2.82 ms	

Rysunek 3.8: Zapytania wchodzące w skład trzeciej grupy. Źródło własne.

wszystkich zmian wprowadzonych przez transakcję, a tym samym też jej zakończenie. W związku z tym ponownie nie ma żadnej możliwości dowiedzenia się jakie zapytania tworzą transakcje bazodanowe.

Podsumowując, narzędzie o nazwie OneAgent zarządzane przez firmę Dynatrace umożliwia nie tylko monitorowanie aplikacji, ale także maszyn, na których został zainstalowany. Funkcjonalność śledzenia i grupowania transakcji bazodanowych jest jedną z wielu jakie dostarcza ten agent. Zaletą tego rozwiązania jest automatyczne przyporządkowanie zapytań do różnych grup, natomiast do wad można zaliczyć brak informacji o oryginalnych parametrach, a także przede wszystkim nieumiejętność rejestrowania transakcji bazodanowych.

3.4. Datadog

Ostatnim przykładem jest również komercyjny agent o nazwie Datadog zarządzany przez firmę o tej samej nazwie [10]. Kod źródłowy tego narzędzia jest publicznie dostępny i można go znaleźć na platformie GitHub [11]. Działa podobnie jak poprzedni agent, czyli poprzez uruchomienie wielu wyspecjalizowanych procesów zbierających różne informacje o wydajności maszyny i aplikacjach na niej uruchomionych. Najważniejsze metryki są następnie udostępniane na stronie internetowej.

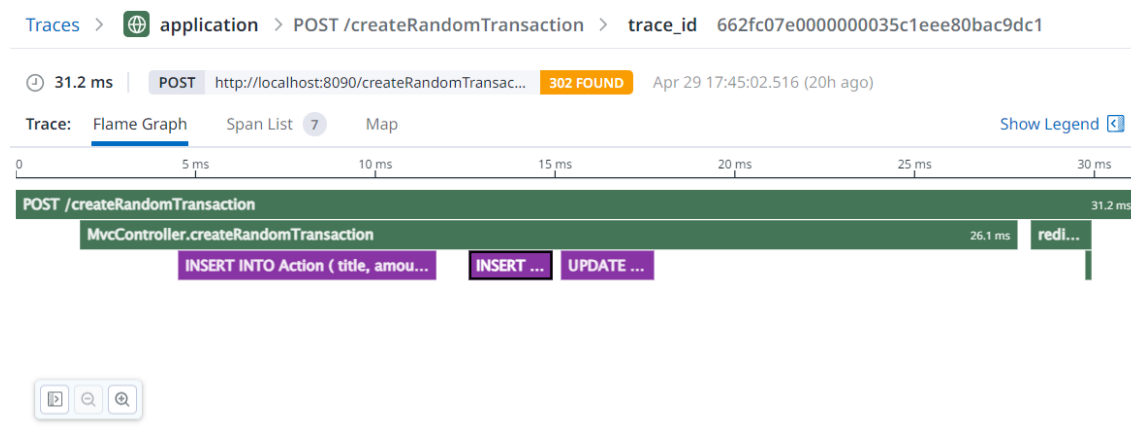
Przykład służący zaprezentowaniu działania narzędzia Datadog, w kontekście śledzenia i grupowania transakcji bazodanowych, został przeprowadzony w taki sam sposób jak dla poprzednich agentów. Okazało się, że Datadog również pozwala na rejestrowanie zapytań i transakcji bazodanowych niezależnie od punktów końcowych podobnie do poprzednika, ale bez przyporządkowywania ich do grup. Na rysunku 3.9 można zobaczyć zebrane dane przez narzędzie Datadog w czasie kilku różnych interakcji z interfejsem użytkownika aplikacji testowej.

Resources 6 resources		Search Resources	Options
RESOURCE NAME	REQUESTS		
☆ <code>SELECT * FROM customer</code>	2	<div><div></div></div>	
☆ <code>INSERT INTO account (id, balance, creation_date) VALUES (?)</code>	2	<div><div></div></div>	
☆ <code>SELECT * FROM account</code>	4	<div><div></div></div>	
☆ <code>INSERT INTO Action (title, amount, type, account_id, status, date) VALUES (?)</code>	4	<div><div></div></div>	
☆ <code>SELECT * FROM action</code>	2	<div><div></div></div>	
☆ <code>UPDATE Account SET balance = balance + ? WHERE id = ?</code>	2	<div><div></div></div>	

Rysunek 3.9: Zarejestrowane zapytania bazodanowe przez narzędzie Datadog. Źródło własne.

Na rysunku 3.9 widać zarejestrowane i przedstawione już w rozwiniętej wersji zapytania bazodanowe, a także ich liczbę wywołań. Dodatkowo istnieje możliwość wyświetlenia większej ilości informacji o tych zapytaniach jak np. współczynnik niepowodzeń poprzez wybranie odpowiednich opcji w prawym górnym rogu rysunku 3.9.

Datadog umożliwia również zajrzenie w szczegóły konkretnych wywołań zapytań, w których można znaleźć wykres przedstawiający ich powiązanie z punktem końcowym i innymi zapytaniami realizowanymi w trakcie obsługi tego żądania na osi czasu. Na rysunku 3.10 można zobaczyć taki wykres dla jednego z wywołań ostatniego zapytania z rysunku 3.9.



Rysunek 3.10: Wykres przedstawiający powiązanie zapytań z punktem końcowym na osi czasu. Źródło własne.

Na wykresie 3.10 widać kolejne etapy przedstawione na osi czasu obsługi punktu końcowego `/createRandomTransaction` wraz z zapytaniami bazodanowymi wykonanymi w trakcie realizacji tego żądania. Kolejne operacje są tożsame z tymi przedstawionymi również na rysunku 3.4, które były wyjaśniane podczas omawiania podobnej funkcjonalności obecnej w OpenTelemetry. Tak samo jak w poprzednich agentach tak i w tym nie ma żadnej informacji o oryginalnych parametrach, a także o tym czy zapytania z rysunku 3.10 tworzą transakcję czy są wykonywane indywidualnie.

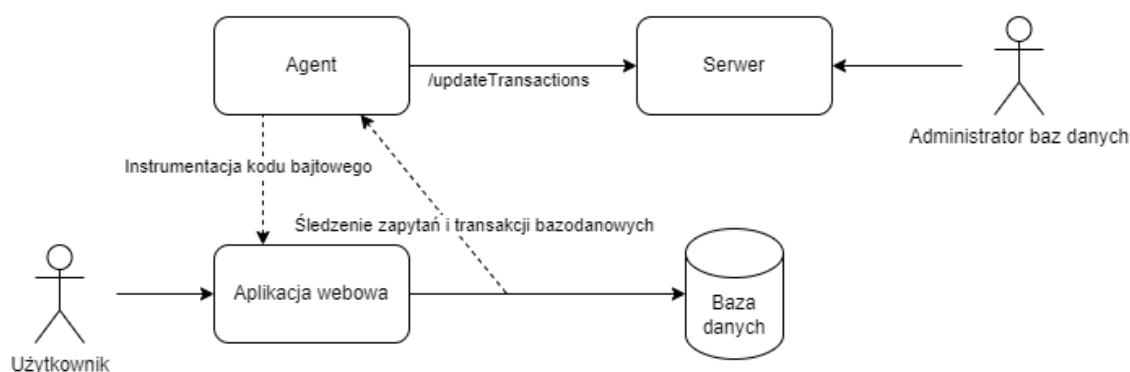
Podsumowując, narzędzie o nazwie Datadog zarządzane przez firmę o tej samej nazwie umożliwia nie tylko monitorowanie aplikacji, ale także maszyn, na których został zainstalowany podobnie do poprzednika. Zaletą tego agenta jest tworzenie powiązań między zapytaniami bazodanowymi a punktami końcowymi w postaci czytelnych wykresów. Natomiast wady są analogiczne do dwóch wcześniejszych przykładów agentów.

4. PROJEKT WŁASNEGO AGENTA

Aby pokazać możliwości automatycznej instrumentacji kodu, zrealizowano dedykowane temu celowi narzędzie. W niniejszym rozdziale zostało zaprezentowane autorskie rozwiązanie odpowiedzialne za automatyczne śledzenie i grupowanie transakcji bazodanowych. W związku z tym w pierwszym podrozdziale został wytłumaczony w szczegółach sposób implementacji tego narzędzia i jego komponentów poprzedzony diagramem architektury. W drugim podrozdziale została poruszona kwestia wsparcia tego rozwiązania dla dowolnej bazy danych. Na koniec zaprezentowano jego działanie na podstawie aplikacji testowej omówionej w podrozdziale 3.1 i osiągnięte wyniki porównano z już istniejącymi na rynku agentami przedstawionymi w poprzednim rozdziale. Wszystkie omówione klasy i funkcje można znaleźć w repozytorium GitHub pod tym linkiem <https://github.com/Winetq/research-project>, do którego dostęp posiada każda osoba.

4.1. Architektura narzędzia

Narzędzie składa się z dwóch komponentów: agenta i serwera. Na rysunku 4.1 został przedstawiony diagram architektury wraz ze wszystkimi składowymi.



Rysunek 4.1: Diagram architektury przedstawiający proponowane narzędzie. Źródło własne.

Diagram z rysunku 4.1 prezentuje w uproszczony sposób zasadę działania tego narzędzia. Można na nim zaobserwować, że agent dzięki procesowi instrumentacji kodu bajtowego jest w stanie śledzić zapytania i transakcje bazodanowe, które wykonuje aplikacja webowa jak np. aplikacja testowa z podrozdziału 3.1 do swojej bazy danych. Zgromadzone dane agent wysyła następnie do drugiego komponentu za pomocą punktu końcowego /updateTransactions. Z kolei zadaniem serwera jest wyświetlenie odebranych danych w przystępny sposób na stronie internetowej. W rezultacie np. administrator baz danych otrzymuje dostęp do informacji, które pozwolą mu na szybkie zdiagnozowanie problemów w przypadku, gdy zapytanie lub zapytania wchodzące w skład transakcji wykonywały się podejrzanie długo. W następnych podrozdziałach został szerzej wyjaśniony sposób implementacji obu komponentów, czyli agenta i serwera.

4.1.1. Sposób implementacji agenta

Ten komponent został stworzony z wykorzystaniem m.in. biblioteki Javassist, a także pakietu o nazwie *instrument*, który, jak wiadomo z podrozdziału 2.4, udostępnia interfejs o nazwie *ClassFile-*

Transformer wraz z metodami *transform*, dzięki którym istnieje możliwość modyfikacji kodu bajtowego. Wyzwanie polega na tym, aby znaleźć odpowiednie klasy, dla których proces instrumentacji się odbędzie. Aplikacja testowa z podrozdziału 3.1, w celu nawiązania komunikacji z PostgreSQL, korzysta z przystosowanego do tej bazy danych sterownika tzw. JDBC, który udostępnia klasy implementujące interfejsy z pakietu *sql*. Dlatego najlepszym miejscem na wydobycie potrzebnych agentowi informacji są metody pochodzące z tych interfejsów, które są definiowane w klasach sterownika. Dzięki takiemu podejściu możliwe jest stworzenie narzędzia nieograniczającego się tylko do jednego sterownika, w tym przypadku PostgreSQL, dlatego że każdy inny sterownik obsługujący dowolną inną bazę danych np. MySQL również dostarcza klasy implementujące interfejsy z pakietu *sql*.

W związku z powyższymi interfejsami, których metody zostały zmodyfikowane, to *PreparedStatement* i *Connection*. Pierwszy z nich udostępnia funkcje odpowiedzialne za wykonanie zapytań takie jak *executeQuery*, *executeUpdate* czy *execute*, które umożliwią ich pobranie, a także mierzenie ich czasu. Natomiast drugi dostarcza metody takie jak *commit* lub *rollback* pozwalające na zatwierdzenie lub wycofanie transakcji, ale również na modyfikację flagi *autoCommit*. Gdy jest ona ustawiona na wartość prawdziwą to każde zapytanie jest traktowane jako transakcja składająca się tylko z pojedynczego zapytania. W przeciwnym razie zapytania są dodawane do transakcji, która kończy się wraz z jej zatwierdzeniem lub wycofaniem. Dzięki takiemu mechanizmowi możliwe jest określenie sytuacji, w której transakcja składa się tylko z jednego zapytania, a kiedy z większej ich liczby.

Klasa agenta, która implementuje interfejs o nazwie *ClassFileTransformer* z pakietu o nazwie *instrument* została pokazana w pseudokodzie 7.

Algorytm 7 Klasa implementująca interfejs o nazwie *ClassFileTransformer* wraz z metodą *transform*

```
1: abstract class Transformer implements ClassFileTransformer {  
2:     byte[] transform(ClassLoader loader, String className, Class classBeingRedefined, Protec-  
        tionDomain protectionDomain, byte[] classfileBuffer) {  
3:         if (checkIfImplements(classfileBuffer)) {  
4:             return transformClass(classfileBuffer);  
5:         }  
6:         return classfileBuffer;  
7:     }  
8: }
```

Na uwagę zasługuje funkcja *checkIfImplements* zawarta w instrukcji warunkowej w linii nr 3. Sprawdza ona czy klasa, dla której aktualnie wykonuje się metoda *transform*, implementuje interfejs *PreparedStatement* lub *Connection* stosując w tym celu algorytm przeszukiwania wszerek (ang. *breadth-first search*, BFS) i odpowiednie do tego zadania funkcje pochodzące z biblioteki *Javassist* jak np. *getInterfaces* do pobrania interfejsów danej klasy. Dzięki takiemu podejściu możliwe jest znalezienie wszystkich interfejsów implementowanych przez aktualną klasę, a także przez jej klasy pochodne. To co wystarczy zrobić to sprawdzić czy wśród odnalezionych interfejsów występuje *PreparedStatement* lub *Connection*. Jeśli tak nie jest to metoda *transform* zwraca już istniejącą do tej pory definicję klasy, co można zobaczyć w linii nr 6. W przeciwnym razie funkcja *checkIfImplements* przekazuje sterowanie do następnego polecenia, którym jest metoda *transformClass*. Nie bez przyczyny klasa *Transformer* jest abstrakcyjna, dlatego że posiada dwie klasy pochodne definiujące właśnie metodę abstrakcyjną jaką jest *transformClass*. Jedna z nich odpowiedzialna jest za modyfikację kodu bajtowego funkcji pochodzących z klasy implementującej interfejs *PreparedStatement*, a druga z klasy implementującej *Connection*. W związku z tym polecenie z linii nr 4 z pseudokodu 7 przekierowuje sterowanie do

metody *transformClass* z odpowiedniej klasy pochodnej w zależności od tego, czy sprawdzana klasa implementuje interfejs *PreparedStatement* czy *Connection*.

Klasa pochodna zajmująca się instrumentacją metod z interfejsu *PreparedStatement* modyfikuje wspomniane już trzy funkcje *executeQuery*, *executeUpdate* i *execute* w taki sam sposób. Kod, który jest do nich dodawany w funkcji *transformClass*, można zobaczyć na listingu 4.1.

Listing 4.1: Kod dodawany do metod z interfejsu *PreparedStatement*

```
CtMethod ctMethod = ctClass.getDeclaredMethod(method, null);
ctMethod.insertBefore("{ start = System.nanoTime();" +
    "parameters = new ArrayList();" +
    "for (int i = 1; i <= preparedParameters.getParameterCount(); i++)" +
    "parameters.add(preparedParameters.toString(i, true)); }");
ctMethod.insertAfter("{ finish = System.nanoTime();" +
    "timeElapsed = finish - start;" +
    "agent.DataStore.processData(preparedQuery.query.toString(), parameters, connection, " +
    "TimeUnit.NANOSECONDS.toMicros(timeElapsed)); }");
```

Parametrem funkcji *transformClass* jest klasa w postaci tablicy bajtów, dzięki której można stworzyć obiekt ją reprezentujący klasą *CtClass* za pomocą metody *makeClass* pochodzącej ze wspomnianej już klasy *ClassPool* w podrozdziale 2.4. Pierwszym poleceniem widocznym na listingu 4.1 jest pobranie z tego obiektu, czyli ze zmiennej o nazwie *ctClass*, danej metody przedstawionej za pomocą klasy *CtMethod*, która zostanie zmodyfikowana za pomocą dwóch następnych poleceń. Zatem na początku instrumentowanej funkcji wstawiany jest kod odpowiedzialny za wystartowanie pomiaru czasu i zebranie parametrów zapytania. Zmienne o nazwach *start* i *parameters* to pola klasy, które zostały stworzone za pomocą konstruktora klasy *CtField*, a następnie dodane do tej klasy dzięki metodzie *addField()* z biblioteki *Javassist*. JDBC umożliwia wysłanie zapytania jednocześnie z parametrami, ale także możliwe jest ustawienie parametrów później i zastąpienie ich w zapytaniu znakami zapytania. Dzięki tej pętli ten drugi wariant też jest zrealizowany. Dzieje się tak dlatego, że klasa sterownika PostgreSQL implementująca interfejs *PreparedStatement* w zmiennej o nazwie *preparedParameters* przechowuje parametry, które są podawane i ustawiane poza zapytaniem. Wystarczy tylko skorzystać z odpowiednich funkcji tego pola, aby je pobrać. Z kolei na końcu modyfikowanej metody umieszczany jest kod odpowiedzialny za zakończenie pomiaru czasu i obliczenie go, a także za wywołanie funkcji *processData* z klasy *DataStore*. Zmienne o nazwach *finish* i *timeElapsed* są to też pola klasy, które zostały stworzone w ten sam sposób co dwa poprzednie atrybuty. Parametry przekazane do metody *processData* to:

- zapytanie w formie tekstowej pobrane z pola klasy sterownika o nazwie *preparedQuery*, dla którego aktualnie wywoływana jest metoda *executeQuery*, *executeUpdate* czy *execute*,
- lista potencjalnych parametrów ustawionych poza zapytaniem jeśli takie były,
- pole klasy o nazwie *connection* reprezentujący połączenie z bazą danych, w ramach którego te zapytanie jest wykonywane, pozwalające także na pobranie wartości flagi *autoCommit*,
- czas, który upłynął od wystartowania do zakończenia jego pomiaru w mikrosekundach, mówiący o tym jak długo realizowało się te zapytanie.

Zadaniem metody *processData* jest zapisanie przekazanych parametrów w odpowiedni sposób i wysłanie ich do serwera. Wykorzystana została do tego mapa, gdzie kluczem jest zapytanie pozbawione parametrów, a więc ze znakami zapytania, a wartością lista klas reprezentujących transakcje, których struktura odpowiada kluczowi. Zatem ta klasa przechowuje pozostałe dane, czyli zapytanie z parametrami, czas i status mówiący o tym, czy dana transakcja została zatwierdzona czy wycofana w przypadku, gdy składa się z wielu zapytań. Zasada działania funkcji *processData* została wytłumaczona na podstawie schematu blokowego, który można zobaczyć na rysunku 4.3 na ostatniej stronie tego

podrozdziału.

Pierwszym etapem metody *processData* przedstawionym w pierwszym bloku na rysunku 4.3 jest zastąpienie znaków zapytania w zapytaniu parametrami jeśli takie były i na odwrót. Zostało to osiągnięte dzięki zastosowaniu klasy *ExpressionDeParser* z otwartoźródłowej biblioteki *JSqlParser* [12], która pozwala na wyodrębnienie struktury zapytania. Następnym krokiem jest blok warunkowy decydujący o przepływie sterowania w zależności od wartości flagi *autoCommit*. Gdy jest ona ustawiona na wartość prawdziwą to transakcja składa się tylko z jednego zapytania, a w przeciwnym razie z wielu. W pierwszym przypadku w kolejnym etapie tworzony jest obiekt reprezentujący transakcję. Później występuje kolejny blok warunkowy sprawdzający czy w mapie istnieje już klucz równy zapytaniu ze znakami zapytania, czyli bez parametrów, które zostało stworzone w pierwszym bloku. Jeśli tak to pobierana jest z mapy lista transakcji odpowiadająca temu kluczowi powiększana następnie o nowy obiekt reprezentujący transakcję zainicjalizowany dwa bloki wcześniej. Na koniec aktualizowana jest mapa dla tego klucza powiększoną o nową transakcję listą. Z kolei jeśli nie to najpierw tworzona jest lista, do której dodaje się nową transakcję, a następnie mapa jest powiększana o klucz i nową listę transakcji. Tak czy inaczej ostatnim krokiem tego odgałęzienia jest wysłanie mapy do serwera za pomocą punktu końcowego */updateTransactions* w ramach protokołu HTTP i metody POST. Cel ten został zrealizowany dzięki bibliotece *Jersey* [13]. W drugim przypadku, czyli gdy flaga *autoCommit* ustawiona jest na wartość fałszywą, w pierwszej kolejności tworzona jest pomocnicza mapa, w której klucz to obiekt reprezentujący połączenie z bazą danych, a wartość to klasa odpowiadająca za większą transakcję złożoną z wielu zapytań. Dzięki takiemu powiązaniu możliwa jest w kolejnych blokach obsługa takich transakcji pochodzących od wielu klientów w tym samym czasie, dlatego że każdy potencjalny użytkownik posiada swój unikatowy obiekt reprezentujący połączenie z bazą danych, w ramach którego taka transakcja się wykonuje. W związku z tym następny blok, który jest warunkowy, sprawdza czy w tej pomocniczej mapie istnieje już klucz równy obiektowi reprezentującemu połączenie z bazą danych. Jeśli nie to inicjalizuje się obiekt odpowiadający za większą transakcję, który następnie zapisywany jest w pomocniczej mapie wraz z odpowiadającym mu kluczem. Z kolei jeśli tak to pobierany jest najpierw dotychczasowy obiekt większej transakcji z pomocniczej mapy powiększany następnie o nowe informacje związane z kolejnym zapytaniem wchodzącym w skład aktualnej transakcji. Na koniec aktualizowana jest wartość w pomocniczej mapie dla tego klucza.

Warto wspomnieć również o tym, że obie mapy wspomniane w powyższym opisie schematu blokowego z rysunku 4.3 zaimplementowane są z użyciem klasy *ConcurrentHashMap* w języku programowania Java [14]. Inaczej mówiąc, jest to tablica haszowana, której elementy tworzą początek, czyli głowę, listy dwukierunkowej przechowującej dane w postaci pary klucz-wartość, w obrębie której hasz kluczy jest taki sam. Jednak jak ta lista osiągnie domyślnie rozmiar ośmiu elementów to jest zamieniana w celach optymalizacyjnych na drzewo czerwono-czarne. Dzięki temu złożoność czasowa wszystkich operacji wynosi $O(\log(n))$. Dodatkowo klasa *ConcurrentHashMap* jest rozbudowana o synchronizację za pomocą mechanizmu monitorów dla różnych operacji, dzięki czemu też pozwala na obsługę transakcji pochodzących od wielu klientów w tym samym czasie.

Z kolei druga klasa pochodna zajmuje się instrumentacją wymienionych już wcześniej metod *commit* i *rollback* z interfejsu *Connection*. Kod, który jest do nich dodawany w funkcji *transformClass*, można zobaczyć na listingu 4.2.

Listing 4.2: Kod dodawany do metod z interfejsu *Connection*

```
CtMethod commitMethod = ctClass.getDeclaredMethod(commitMethodName, null);
commitMethod.insertAfter("agent.DataStore.executeTransaction(this, \"COMMIT\");");

CtMethod rollbackMethod = ctClass.getDeclaredMethod(rollbackMethodName, null);
```

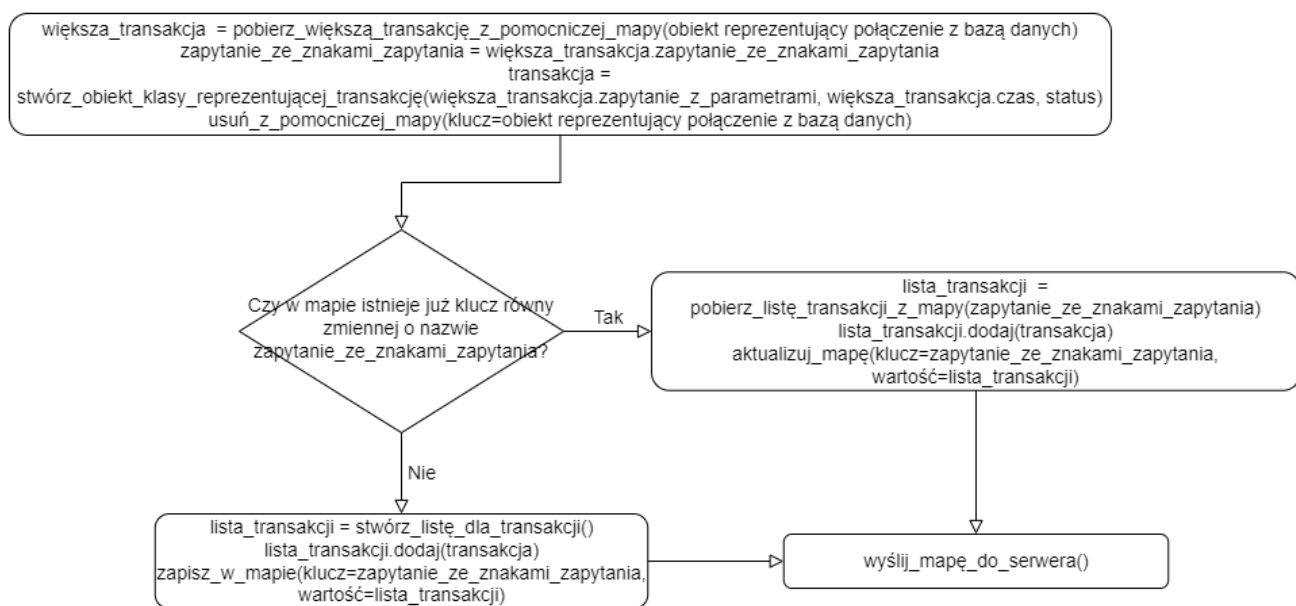


```
rollbackMethod.insertAfter("agent.DataStore.executeTransaction(this, \"ROLLBACK\");");
```

W tym przypadku również ze zmiennej o nazwie `ctClass` pobrane zostają metody `commit` oraz `rollback` reprezentowane przez klasę `CtMethod`. Dla przypomnienia te funkcje oznaczają koniec transakcji poprzez jej zatwierdzenie lub wycofanie. W związku z tym na końcu nich umieszczany jest kod odpowiedzialny za wywołanie funkcji `executeTransaction` z klasy `DataStore`. Parametrami tej metody są:

- obiekt klasy sterownika implementującej interfejs `Connection`, a więc reprezentujący aktualne połączenie z bazą danych,
- tekst przyjmujący wartość `COMMIT` lub `ROLLBACK` w zależności od tego czy nastąpiło zatwierdzenie transakcji czy jej wycofanie.

Zasada działania funkcji `executeTransaction` została wytłumaczona również na podstawie schematu blokowego, który można zobaczyć na rysunku 4.2. Warto jednak zauważyć, że blok warunkowy i wszystkie bloki po nim to jest ten sam fragment schematu przedstawiony również na rysunku 4.3 w prawym dolnym rogu. Zostało to powielone, aby móc zaprezentować dwa osobne diagramy dla obu funkcji. Dzięki temu są bardziej czytelne i przejrzyste. W związku z tym mapa i pomocnicza mapa to są te same struktury danych zarówno na schemacie 4.2 jak i 4.3.



Rysunek 4.2: Schemat blokowy przedstawiający zasadę działania funkcji `executeTransaction`. Źródło własne.

Zatem pierwszy blok z rysunku 4.2 to jedyna nowość w porównaniu do schematu blokowego 4.3. Wywołanie metody `executeTransaction` oznacza zakończenie większej transakcji złożonej z wielu zapytań poprzez jej zatwierdzenie lub wycofanie. W związku z tym pierwszy blok polega na pobraniu obiektu reprezentującego większą transakcję z pomocniczej mapy za pomocą klucza odpowiadającego za połączenie z bazą danych, w ramach którego ta transakcja się zakończyła. Następnie pola tego obiektu są wykorzystane do stworzenia instancji klasy reprezentującej transakcję, tak aby było możliwe bezproblemowe dodanie jej do listy transakcji jakiegoś klucza w oryginalnej mapie. Dzięki temu możliwe jest ponowne użycie niektórych bloków obecnych już na schemacie blokowym 4.3 w przypadku, gdy flaga `autoCommit` ustawiona jest na wartość `prawdziwą`. Na koniec usuwany jest klucz wraz z jego wartością z pomocniczej mapy, dlatego że połączenie jak i sama transakcja się już zakończyły.

Z podrozdziału 2.4 wiadomo, że ostatnim krokiem w celu poprawnego zaimplementowania agenta jest oczywiście zdefiniowanie funkcji *premain* albo *agentmain*. Jednak w tym przypadku wybór padł na tę pierwszą i można to zaobserwować w pseudokodzie 8.

Algorytm 8 Klasa agenta z metodą *premain* rejestrująca klasy pochodne PreparedStatementTransformer i ConnectionTransformer

```
1: class Agent {
2:     public static void premain(String agentArgs, Instrumentation inst) {
3:         inst.addTransformer(new PreparedStatementTransformer());
4:         inst.addTransformer(new ConnectionTransformer());
5:     }
6: }
```

W tej metodzie została również wywołana dwa razy funkcja *addTransformer* w celu rejestracji tych dwóch klas pochodnych PreparedStatementTransformer i ConnectionTransformer dziedziczących po klasie bazowej przedstawionej w pseudokodzie 7. Tym samym agent jest gotowy do uruchomienia według instrukcji dotyczącej instrumentacji kodu bajtowego za pomocą agenta dołączonego statycznie 2.3.1. Jednak możliwe jest również przekazanie parametrów mówiących o tym, w jaki sposób mają być prezentowane dane zebrane przez agenta. Pierwsza z opcji, która jest domyślną, to na serwerze. Natomiast druga to w konsoli, która oznacza, że mapa z transakcjami wyświetlana jest w konsoli bez wysyłania jej na serwer. Parametr można przekazać w poniższy sposób:

```
java -javaagent:agent.jar=server -jar aplikacja.jar
```

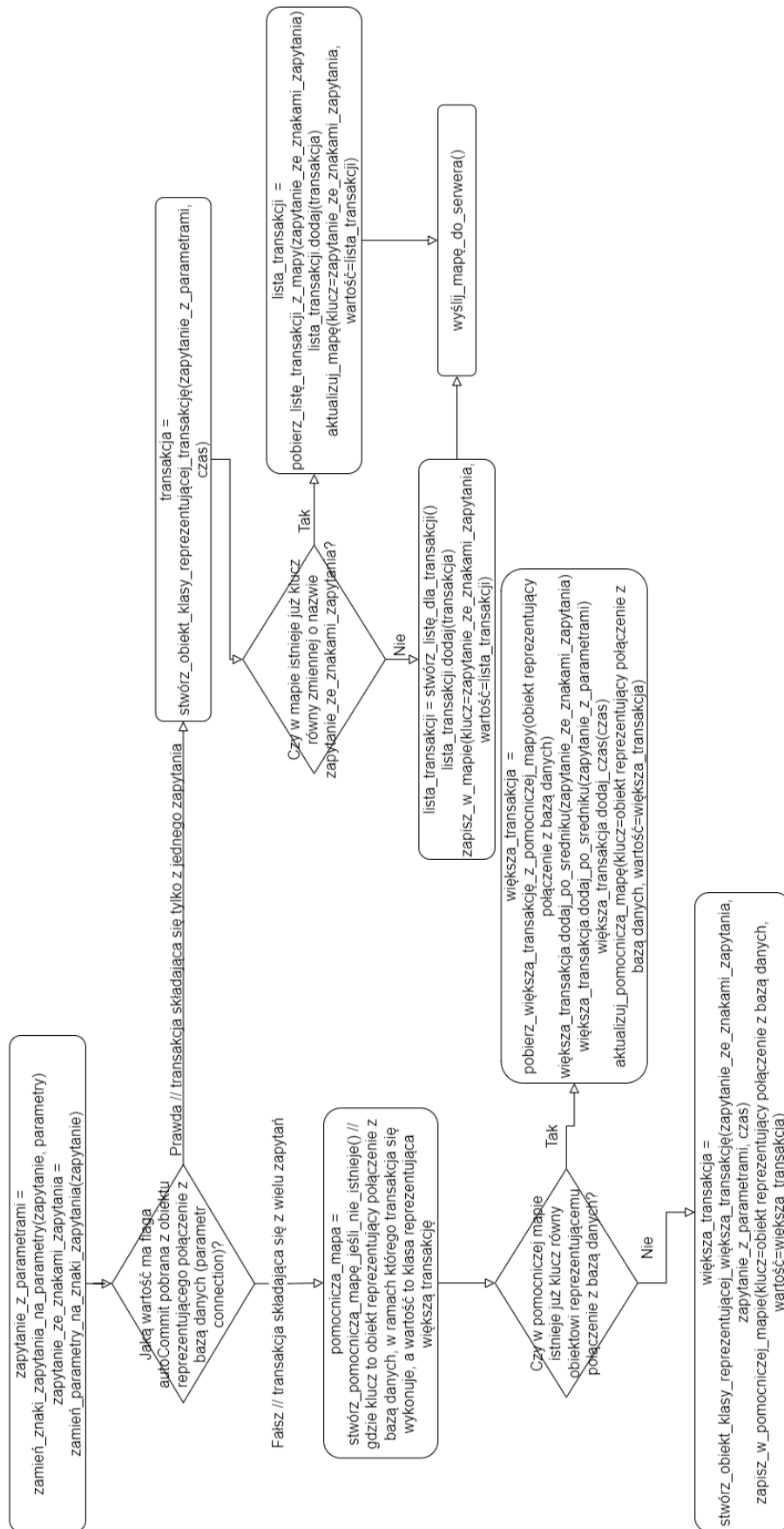
Jak już było wspomniane powyższy parametr jest domyślny, więc też jest opcjonalny, czyli że nie trzeba go przekazywać. W celu wyświetlania mapy z transakcjami w konsoli trzeba już przekazać odpowiedni parametr, który można zobaczyć poniżej:

```
java -javaagent:agent.jar=console -jar aplikacja.jar
```

4.1.2. Sposób implementacji serwera

Serwer to drugi komponent tego narzędzia, którego zadaniem jest odebranie mapy z danymi od agenta i wyświetlenie ich na stronie internetowej. Jest to prosta aplikacja webowa napisana w języku programowania Java w wersji 17 wraz z użyciem platformy Spring Boot i silnika szablonów stron internetowych Thymeleaf. Udostępnia punkt końcowy /updateTransactions realizowany przez metodę HTTP, a mianowicie POST, który pozwala na odebranie zebranych informacji od agenta. Warto wspomnieć, że ten komponent posiada obraz dockerowy w rejestrze Docker Hub udostępniony dla każdego pod tym linkiem <https://hub.docker.com/r/mcwynar/research-project>, który jest tworzony po każdym wypchnięciu zmian do repozytorium dzięki GitHub Actions. W związku z tym zalecanym sposobem uruchomienia serwera jest skorzystanie z tego obrazu za pomocą następującego polecenia:

```
docker run -d -p 8080:8080 mcwynar/research-project:agent_server
```



Rysunek 4.3: Schemat blokowy przedstawiający zasadę działania funkcji *processData*. Źródło własne.

4.2. Wsparcie dla dowolnej bazy danych

Agent został zaimplementowany z myślą o tym, aby był w stanie działać dla dowolnej bazy danych. Dlatego też metody, które zostały poddane instrumentacji pochodzą z interfejsów z pakietu *sql*. Jak już było wspomniane każdy sterownik obsługujący bazę danych dostarcza klasy implementujące te interfejsy, które definiują też te metody. W związku z tym można przypuszczać, że agent przeprowadzi instrumentację niezależnie od użytej bazy danych. Ale czy na pewno?

W celach sprawdzenia została stworzona taka sama aplikacja jak ta opisana w podrozdziale 3.1 z tą różnicą, że zamiast bazy danych PostgreSQL użyto MySQL. Po uruchomieniu tej aplikacji z agentem dołączonym statycznie okazało się, że autorskie narzędzie nie jest w stanie przeprowadzić instrumentacji metod pochodzących z interfejsu *PreparedStatement* z powodu błędu sygnalizującego brak pola o nazwie *preparedParameters*. Klasa sterownika obsługującego bazę danych MySQL nie ma po prostu takiego atrybutu tak samo jak drugiego pola o nazwie *preparedQuery*. Informacje o zapytaniu, dla którego aktualnie wywoływana jest metoda *executeQuery*, *executeUpdate* czy *execute*, oraz o liście potencjalnych parametrów związanych z tym zapytaniem i ustawionym poza nim są przechowywane też w polach, ale o innych nazwach i typach. Kod, który powinien być dodany do tych metod z klasy sterownika obsługującej bazę danych MySQL w funkcji *transformClass*, można zobaczyć na listingu 4.3.

Listing 4.3: Kod dodawany do metod z interfejsu *PreparedStatement* implementowanego przez klasę sterownika obsługującego bazę danych MySQL

```
CtMethod ctMethod = ctClass.getDeclaredMethod(method, null);
ctMethod.addLocalVariable("preparedQuery", pool.get("com.mysql.cj.PreparedQuery"));
ctMethod.addLocalVariable("queryBindings", pool.get("com.mysql.cj.QueryBindings"));
ctMethod.insertBefore("{ start = System.nanoTime();" +
    "parameters = new ArrayList();" +
    "preparedQuery = ((com.mysql.cj.PreparedQuery) query);" +
    "queryBindings = preparedQuery.getQueryBindings();" +
    "for (int i = 0; i < preparedQuery.getParameterCount(); i++)" +
    "parameters.add(queryBindings.getBindValues()[i].getString()); }");
ctMethod.insertAfter("{ finish = System.nanoTime();" +
    "timeElapsed = finish - start;" +
    "agent.DataStore.processData(preparedQuery.getOriginalSql(), parameters, connection, " +
    "TimeUnit.NANOSECONDS.toMicros(timeElapsed)); }");
```

Jedyną różnicą widoczną na listingu 4.3 w porównaniu do 4.1 jest sposób pobierania informacji o zapytaniu oraz o liście parametrów ustawionych poza nim. Tym razem atrybut zawierający zapytanie jest nazwany *query*, ale też klasa go reprezentująca należy do innego pakietu niż poprzednio. Ten atrybut przechowuje również parametry zapytania, które można pobrać za pomocą odpowiednich metod. To kolejna różnica względem klasy sterownika obsługującej PostgreSQL, która te parametry zapisywała w odrębnym atrybucie o nazwie *preparedParameters* co można zaobserwować na listingu 4.1. Poza tym kod przedstawiony na tych dwóch listingach 4.1 i 4.3 wykonuje instrumentację dokładnie w taki sam sposób, z tym że pierwszy z nich przeznaczony jest dla metod klas sterownika obsługującego bazę danych PostgreSQL, a drugi z nich MySQL.

Zatem autorski agent działa na pewno dla baz danych PostgreSQL i MySQL. Może działa też dla innych, pod warunkiem że nazwy atrybutów i ich typy odpowiadają temu co można zobaczyć na listingach 4.1 i 4.3. Jednak niemożliwe jest napisanie wspólnego kodu do przeprowadzenia takiej instrumentacji dla dowolnej bazy danych ze względu na to, że sposób przechowywania zapytań czy ich parametrów lub po prostu nazwy atrybutów i ich typy mogą się różnić w zależności od użytej bazy danych.

4.3. Wyniki

Do zaprezentowania działania tego narzędzia wykorzystana została aplikacja testowa omówiona w podrozdziale 3.1. Przykład polegał na wykonaniu kilku różnych interakcji z interfejsem użytkownika dokładnie w taki sam sposób jak dla innych agentów przedstawionych w poprzednim rozdziale. Na rysunku 4.4 można zobaczyć stronę internetową, którą udostępnia serwer, z zebranymi już danymi przez agenta.

Transactions		
Queries in transaction	Times [μs]	Average time [μs]
<div>> INSERT INTO Action (title, amount, type, account_id, status, date) VALUES (?, ?, ?, ?, ?, ?)</div> <div>> UPDATE Account SET balance = balance + ? WHERE id = ?</div>	10019/716	10019716.00
<div>> SELECT * FROM Account</div>	19659, 1535	10597.00
<div>> INSERT INTO Action (title, amount, type, account_id, status, date) VALUES (?, ?, ?, ?, ?, ?)</div> <div>> INSERT INTO Action (title, amount, type, account_id, status, date) VALUES (?, ?, ?, ?, ?, ?)</div> <div>> UPDATE Account SET balance = balance + ? WHERE id = ?</div>	10019994, 10024769, 10017647	10020803.33
<div>> INSERT INTO Account (id, balance, creation_date) VALUES (?, ?, ?)</div>	4555	4555.00

Automated tracking and grouping of database transactions in Java Virtual Machine (JVM) applications
A project created for Research Project classes at Gdansk University of Technology

Rysunek 4.4: Strona internetowa udostępniona przez serwer wyświetlająca zebrane dane przez agenta. Źródło własne.

Rysunek 4.4 przedstawia klucze mapy, o której była mowa na diagramach 4.2 i 4.3, czyli zapytania pozbawione parametrów, które zastąpione zostały znakami zapytania, jest to tzw. ich struktura. Widać, że agent zarejestrował transakcje składające się zarówno z jednego zapytania jak i z dwóch czy nawet trzech. Oprócz tego w drugiej kolumnie można zauważyć czas podany w mikrosekundach mówiący o tym jak długo wykonywały się transakcje wchodzące w skład danej grupy, czyli klucza. Trzecia kolumna przedstawia z kolei średnią arytmetyczną tych czasów. Dla przykładu, trzecia transakcja od góry na rysunku 4.4 złożona jest z trzech zapytań i wykonała się trzy razy, dlatego że w drugiej kolumnie pokazane są trzy pomiary czasu dla tej grupy, a z kolei w trzeciej kolumnie widać ich średnią.

Ponadto, w porównaniu do poprzednich agentów takich jak OpenTelemetry, OneAgent czy Data-dog, istnieje możliwość podejrzenia oryginalnej transakcji wraz z parametrami. W tym celu wystarczy kliknąć w odpowiedni wiersz, który rozwinie wszystkie zapytania. Można to zauważyć na rysunku 4.5.

Ten rysunek przedstawia stan po kliknięciu ostatniej transakcji, która wykonała się raz, patrząc na liczebność pomiarów czasu. Dlatego też wyświetlone zostało jedno jej wystąpienie. Warto zauważyć, że znaki zapytania zastąpiono oryginalnymi parametrami. Dzięki temu można odczytać, że nowe konto stworzono 2. sierpnia o godzinie 10:04:50 ze stanem wynoszącym 100 i o id równym 19. W przypadku gdy transakcja składa się z wielu zapytań, zostaje również wyświetlona informacja o jej statusie, czyli o tym, czy została zatwierdzona czy wycofana. Taką sytuację można zaobserwować na rysunku 4.6, na którym rozwinęto transakcję złożoną z trzech zapytań.

Druga kolumna wskazuje na trzy realizacje tej grupy transakcji, które właśnie można zobaczyć na rysunku 4.6. Poza oryginalnymi zapytaniami z parametrami, można również dostrzec informacje

Transactions		
Queries in transaction	Times [μs]	Average time [μs]
> INSERT INTO Action (title, amount, type, account_id, status, date) VALUES (?, ?, ?, ?, ?, ?) > UPDATE Account SET balance = balance + ? WHERE id = ?	10019716	10019716.00
> SELECT * FROM Account	19659, 1535	10597.00
> INSERT INTO Action (title, amount, type, account_id, status, date) VALUES (?, ?, ?, ?, ?, ?) > INSERT INTO Action (title, amount, type, account_id, status, date) VALUES (?, ?, ?, ?, ?, ?) > UPDATE Account SET balance = balance + ? WHERE id = ?	10019994, 10024769, 10017647	10020803.33
> INSERT INTO Account (id, balance, creation_date) VALUES (?, ?, ?)	4555	4555.00
> INSERT INTO Account (id, balance, creation_date) VALUES(19, 100, '2024-08-02 10:04:50+02')		
Automated tracking and grouping of database transactions in Java Virtual Machine (JVM) applications A project created for Research Project classes at Gdansk University of Technology		

Rysunek 4.5: Strona internetowa udostępniona przez serwer umożliwiająca podejrzanie transakcji wraz z parametrami. Źródło własne.

Transactions		
Queries in transaction	Times [μs]	Average time [μs]
> SELECT * FROM Account	19659, 1535	10597.00
> INSERT INTO Action (title, amount, type, account_id, status, date) VALUES (?, ?, ?, ?, ?, ?) > INSERT INTO Action (title, amount, type, account_id, status, date) VALUES (?, ?, ?, ?, ?, ?) > UPDATE Account SET balance = balance + ? WHERE id = ?	10019994, 10024769, 10017647	10020803.33
> INSERT INTO Action (title, amount, type, account_id, status, date) VALUES('Prowizja za przelew walutowy: Transfer to user 2', 2, 'Przelew walutowy', 2, 'Done', '2024-08-02 10:05:00.0') INSERT INTO Action (title, amount, type, account_id, status, date) VALUES('Transfer to user 2', 22, 'Przelew walutowy', 2, 'Done', '2024-08-02 10:05:00.0') UPDATE Account SET balance=balance+22 WHERE id=2 Status: COMMIT > INSERT INTO Action (title, amount, type, account_id, status, date) VALUES('Prowizja za przelew walutowy: Transfer to user 1', 2, 'Przelew walutowy', 1, 'Done', '2024-08-02 10:05:48.0') INSERT INTO Action (title, amount, type, account_id, status, date) VALUES('Transfer to user 1', 18, 'Przelew walutowy', 1, 'Done', '2024-08-02 10:05:48.0') UPDATE Account SET balance=balance+18 WHERE id=1 Status: COMMIT > INSERT INTO Action (title, amount, type, account_id, status, date) VALUES('Prowizja za przelew walutowy: Transfer to user 3', 2, 'Przelew walutowy', 3, 'Done', '2024-08-02 10:10:26.0') INSERT INTO Action (title, amount, type, account_id, status, date) VALUES('Transfer to user 3', 72, 'Przelew walutowy', 3, 'Done', '2024-08-02 10:10:26.0') UPDATE Account SET balance=balance+72 WHERE id=3 Status: ROLLBACK		
Automated tracking and grouping of database transactions in Java Virtual Machine (JVM) applications A project created for Research Project classes at Gdansk University of Technology		

Rysunek 4.6: Strona internetowa udostępniona przez serwer wyświetlająca status transakcji złożonej z wielu zapytań. Źródło własne.

o statusie tych transakcji. Zmiany wprowadzone przez pierwszą i drugą z nich zostały zatwierdzone, ponieważ ich status to COMMIT. Natomiast ostatnia transakcja została wycofana ze względu na status równy ROLLBACK.

Narzędzie zaprezentowane w tym rozdziale koncentruje się tylko i wyłącznie na tytułowym śledzeniu i grupowaniu transakcji bazodanowych. Agenty przedstawione w poprzednim rozdziale miały znacznie szersze zastosowanie i chociażby pozwalały na monitorowanie punktów końcowych i powiązanie ich z wykonanymi zapytaniami. Natomiast ich największą wadą była nieumiejętność rejestrowania transakcji bazodanowych, czyli określenie tego, czy zapytania są realizowane indywidualnie czy może jednak tworzą transakcję. Jak już było wspomniane, wiedza o tym może pomóc w szybkim

zdiagnozowaniu problemów w przypadku, gdy zapytanie lub zapytania wchodzące w skład transakcji wykonują się podejrzanie długo. Jest to jasny sygnał świadczący o potrzebie optymalizacji tej transakcji poprzez np. zmianę poziomu jej izolacji lub rodzaju blokady (ang. *locks*) np. ze współdzielonej (ang. *shared lock*) na wyłączną (ang. *exclusive lock*). Natomiast narzędzie przedstawione w tym rozdziale wychodzi naprzeciw tym problemom i w jasny i czytelny sposób prezentuje zapytania i transakcje zrealizowane podczas korzystania z aplikacji. Oprócz tego wspiera także sytuacje, w których parametry są ustawiane poza zapytaniem. Dodatkowo jest w stanie obsługiwać żądania pochodzące od wielu klientów w tym samym czasie.

5. EKSPERYMENTY I PORÓWNIANIA AGENTÓW

W rozdziale piątym zostały przeprowadzone różne eksperymenty mające porównać, bazując na liczbach, wszystkie omówione narzędzia, czyli OpenTelemetry, OneAgent, Datadog oraz autorski agent przedstawiony w czwartym rozdziale. Cel eksperymentów to sprawdzenie jak bardzo każde z narzędzi spowalnia uruchomienie i funkcjonowanie aplikacji. Inaczej mówiąc, jaki jest narzut czasu (ang. *overhead*) dla aplikacji, do której zostają dołączone wyżej wymienione agenty. Jest to bardzo ważna informacja, dlatego że użytkownikowi zależy na czasie i na pewno nie chciałby, aby jego aplikacja została znacząco spowolniona tylko ze względu na dołączenie do niej agenta. W związku z tym warto sprawdzić czy chociażby ten narzut czasu jest akceptowalny. Pomysł na eksperymenty został zaczerpnięty z artykułu o tytule "A Trace Agent with Code No-invasion Based on Byte Code Enhancement Technology" [15].

5.1. Plan testów

Plan testów został zaprojektowany za pomocą narzędzia Apache JMeter [16], którego celem jest przeprowadzanie testów wydajnościowych. Zakłada on odpytanie punktu końcowego `/createNewAction` w przypadku przelewu walutowego 100 razy w ciągu jednej sekundy. Metryki, które będą monitorowane, to:

- czas uruchomienia aplikacji testowej,
- średni czas obsługi żądania,
- odchylenie standardowe czasu obsługi żądania,
- maksymalna liczba wątków potrzebnych do działania aplikacji testowej.

Pierwsze trzy metryki można odczytać przy użyciu odpowiednich komponentów udostępnianych przez narzędzie Apache JMeter. Natomiast do śledzenia ostatniej z nich wykorzystany został program Process Explorer, czyli darmowy menedżer zadań i monitor systemu dla Microsoft Windows. Plan zakłada uruchomienie testów trzykrotnie i wyliczenie średniej arytmetycznej z zebranych danych, albo maksymalnej wartości w przypadku ostatniej metryki. Parametry komputera, na którym przeprowadzono testy, zostały przedstawione w tabelce 5.1.

Tabela 5.1: Parametry komputera do przeprowadzenia eksperymentów.

parametr	wartość
Procesor	Intel(R) Core(TM) i7-10510U
Pamięć RAM	16GB
System operacyjny	Windows 10

5.2. Wyniki i kierunki rozwoju

W tabelce 5.2 przedstawiono wyniki wszystkich czterech metryk przedstawionych w poprzednim podrozdziale dla aplikacji testowej zarówno bez jak i z dołączonymi agentami.

Tabela 5.2: Wyniki z przeprowadzonych testów dla wszystkich narzędzi.

	bez agenta	autorski agent	OpenTelemetry	OneAgent	Datadog
czas uruchomienia [s]	3.11	3.80	3.74	5.91	3.68
średni czas [ms]	784	1885	1123	1044	1112
odchylenie standardowe [ms]	367.44	972.38	476.97	485.31	497.99
maksymalna liczba wątków	42	45	51	55	58

Bez zaskoczenia aplikacja testowa bez agenta poradziła sobie najlepiej we wszystkich metrykach. Natomiast inaczej jest w sytuacji, w której został do niej dołączony agent. W takim przypadku występuje wcześniej już wspomniany narzut czasu, a nawet zasobów. Aplikacja testowa z uruchomionym autorskim agentem potrzebowała więcej wątków do jej działania, ale najmniej w porównaniu do innych agentów. Dzieje się tak, dlatego że inne narzędzia mają po prostu znacznie szersze zastosowanie. Czas uruchomienia, w przypadku dołączenia autorskiego agenta, był również konkurencyjny porównując do innych narzędzi takich jak OpenTelemetry czy Datadog. Natomiast pozostałe metryki, czyli średnia arytmetyczna i odchylenie standardowe z czasów obsługi żądania, jednoznacznie wskazują na to, że autorski agent bardziej spowalnia aplikację testową. W przypadku pozostałych narzędzi ten narzut czasu wynosi mniej więcej 300 ms, więc tak naprawdę jest niezauważalny. Natomiast dołączenie autorskiego agenta powoduje spowolnienie czasu obsługi żądania związanego z punktem końcowym `/createNewAction` średnio o ponad 1 s.

Jakob Nielsen w książce pt. "Usability Engineering" z 1993 roku [17] zdefiniował trzy przedziały czasowe na podstawie zdolności percepcyjnych człowieka określające odczucia użytkownika związane z czasem obsługi jego żądania. Zostały one wymienione poniżej:

- od 0 do 100 ms - użytkownik czuje, że system reaguje natychmiastowo i ma poczucie bezpośredniej interakcji z interfejsem użytkownika,
- od 100 ms do 1 s - użytkownik zauważa opóźnienie w obsłudze żądań, ale przepływ jego myśli pozostaje nienaruszony,
- od 1 s do 10 s - użytkownik oczekuje informacji w postaci jakiegoś znaku np. zmiany kształtu kursora myszki czy paska postępu, opóźnienia dłuższe niż 10 s dopuszczalne są jedynie podczas naturalnych przerw w pracy użytkownika.

Z tą wiedzą można stwierdzić, że narzut czasu wprowadzony przez narzędzia już istniejące na rynku jest z pewnością akceptowalny. Natomiast w przypadku agenta zaprezentowanego w poprzednim rozdziale można mieć wątpliwości. Przyczyn tak dużej różnicy jeśli chodzi o spowolnienie aplikacji między autorskim narzędziem a pozostałymi może być wiele.

Jednym z powodów takich wyników może być biblioteka Javassist, która została wykorzystana w autorskim agencie do modyfikacji kodu bajtowego klas. Jak było już wspomniane w podrozdziale 2.4, Javassist działa na poziomie samego języka programowania Java, więc nie wymaga znajomości struktury pliku `.class` czy mnemoników. Dlatego należy do bibliotek wysokiego rzędu. Natomiast istnieją też narzędzia niższego rzędu takie jak ASM czy BCEL, które wymagają takiej wiedzy. Jednak to sprawia, że działają dużo szybciej, gdyż nie ma wtedy potrzeby translacji kodu napisanego w języku programowania Java na mnemoniki. Można również nie korzystać z żadnej biblioteki i proces instrumentacji kodu bajtowego klas przeprowadzać na tablicy bajtów. Jest to najszybszy sposób, ale też i najmniej wygodny. W każdym razie OpenTelemetry, OneAgent i Datadog są dużo szybsze od autorskiego agenta, jeśli chodzi o średni czas obsługi żądania, być może ze względu na zastosowanie bibliotek niższego

rzędu albo niekorzystanie z nich wcale.

Kolejnym powodem może być sposób przesyłania danych do serwera. Jak już było wspomniane do tego celu została wykorzystana biblioteka Jersey, która za pomocą protokołu HTTP i metody POST wysyła mapę z transakcjami na serwer. Tabela 5.3 przedstawia te same wyniki co tabela 5.2, ale z zaktualizowanymi wartościami w sytuacji, w której aplikacja testowa działa wraz z autorskim narzędziem. W trakcie uruchamiania przekazano jednak agentowi parametr o wartości *console* świadczący o tym, że mapa z transakcjami ma być wyświetlana w konsoli bez wysyłania jej na serwer.

Tabela 5.3: Wyniki z przeprowadzonych testów dla autorskiego agenta uruchomionego z parametrem o wartości *console*.

	bez agenta	autorski agent	OpenTelemetry	OneAgent	Datadog
czas uruchomienia [s]	3.11	3.61	3.74	5.91	3.68
średni czas [ms]	784	961	1123	1044	1112
odchylenie standardowe [ms]	367.44	507.67	476.97	485.31	497.99
maksymalna liczba wątków	42	45	51	55	58

Okazało się, że wyniki znacznie się poprawiły. Średni czas obsługi żądania zmniejszył się prawie o 1 s. To tylko potwierdza, że głównym czynnikiem spowalniającym aplikację testową jest sposób przesyłania danych do serwera, a nie algorytmy np. zapisu danych w mapie dostarczone przez autorskie narzędzie. Problem z tym związany może polegać na tym, że agent wysyła mapę ze wszystkim transakcjami do serwera za każdym razem. Rozwiązaniem mogłoby być dostarczenie do serwera informacji związanych tylko z nową aktualnie dodawaną do mapy transakcją. Natomiast problem może również dotyczyć użytej biblioteki, czyli Jersey, realizującej protokół HTTP. Istnieje wiele innych narzędzi zapewniających taką komunikację w języku programowania Java jak np. Apache HttpClient czy OkHttpClient, które są być może bardziej zoptymalizowane. Inną przyczyną tego problemu może leżeć w samym protokole HTTP. Istnieją inne sposoby komunikacji takie jak AMQP (ang. *Advanced Message Queuing Protocol*) lub gniazda (ang. *sockets*), które do tego zadania mogłyby okazać się lepsze, a przede wszystkim szybsze. Wychodzi więc na to, że OpenTelemetry, OneAgent i Datadog wykorzystują szybszy sposób wysyłania danych do serwera, być może dzięki skorzystaniu z bardziej zoptymalizowanej biblioteki lub innego protokołu komunikacyjnego.

Podsumowując, w tym rozdziale zostały przedstawione wyniki i analiza z przeprowadzonych eksperymentów, które miały za zadanie określić narzut czasu i zasobów dla aplikacji testowej wprowadzony przez wszystkie już omówione narzędzia. Okazało się, że autorski agent zapewnia najmniejszy narzut jeśli chodzi o wykorzystanie zasobów, ale za to największy jeśli chodzi o czas wykonania. Średnia czasu obsługi żądania związanego z punktem końcowym `/createNewAction` była większa o prawie 1 s w prównaniu do innych agentów. Przyczyn takiego wyniku może być kilka m.in. biblioteka Javassist do instrumentacji kodu bajtowego klas czy protokół HTTP. W związku z tym dużo jest kierunków rozwoju czy potencjalnych optymalizacji autorskiego agenta. Biorąc to wszystko pod uwagę, te narzędzie z pewnością może konkurować z już istniejącymi narzędziami na rynku.

6. PODSUMOWANIE

Niniejsza praca pokazała, że możliwe jest efektywne automatyczne instrumentowanie kodu bajtowego aplikacji działającej na maszynie wirtualnej Java (JVM). W tym celu zostało zaprezentowane autorskie narzędzie, które za pomocą biblioteki Javassist umożliwiającej modyfikację kodu bajtowego, jest w stanie automatycznie śledzić, grupować i mierzyć czas transakcji bazodanowych, które wykonuje aplikacja do swojej bazy danych. Z sukcesem, dzięki fladze autoCommit, został znaleziony również sposób na rozróżnienie tego, czy zapytania są realizowane indywidualnie czy jednak tworzą transakcję w porównaniu do innych agentów istniejących już na rynku. Funkcjonalnością, która odróżnia jeszcze autorskie narzędzie od innych, jest możliwość podejrzenia oryginalnych zapytań i transakcji wraz z parametrami. Dzięki zapisaniu odpowiednich obiektów w strukturach danych stała się możliwa również obsługa transakcji pochodzących od wielu klientów w tym samym czasie. Oprócz tego została także podjęta próba stworzenia rozwiązania działającego dla dowolnej bazy danych. Niestety zakończona niepowodzeniem, ale dzięki tej próbie autorskie narzędzie wspiera zarówno bazę danych PostgreSQL jak i MySQL. Stworzenie aplikacji testowej umożliwiło łatwe i przejrzyste porównanie wszystkich narzędzi. Natomiast dzięki eksperymentom została przeprowadzona wiarygodna analiza wydajności aplikacji, do której zostały dołączone omówione agenty. Zrealizowane zadania świadczą o wykonaniu założonych na początku celów.

Z eksperymentów można dowiedzieć się, że rozwiązanie zaproponowane w tej pracy magisterskiej wprowadza najmniejsze dodatkowe zapotrzebowanie na zasoby systemowe, ale największy dodatkowy narzut czasowy. Przyczyną jest przede wszystkim sposób przesyłania danych do serwera. Aktualnie wysyłana jest mapa ze wszystkimi transakcjami za każdym razem. Rozwiązaniem mogłoby być dostarczenie informacji związanych tylko z nową aktualnie dodawaną do mapy transakcją. Natomiast problem może również leżeć w samym protokole HTTP lub w bibliotece Jersey realizującej ten sposób komunikacji. Innym czynnikiem spowalniającym aplikację może być również biblioteka Javassist wykorzystana w autorskim narzędziu, która należy do bibliotek wysokiego rzędu. W związku z tym jest kilka kierunków rozwoju, które pomogłyby temu rozwiązaniu stać się znaczącym narzędziem na rynku. Natomiast już na ten moment autorski agent z pewnością może konkurować z dotychczas istniejącymi narzędziami.

WYKAZ LITERATURY

- [1] J. Gage, "Database locking and database isolation levels," 2020. [Online]. Available: <https://retool.com/blog/isolation-levels-and-locking-in-relational-databases>.
- [2] T. Lindholm, F. Yellin, G. Bracha, A. Buckley, and D. Smith, *The Java® Virtual Machine Specification*. Oracle, 2024. [Online]. Available: <https://docs.oracle.com/javase/specs/jvms/se22/html>.
- [3] J. Aarniala, "Instrumenting Java bytecode," 2005. [Online]. Available: <https://www.cs.helsinki.fi/u/pohjalai/k05/okk/seminar/Aarniala-instrumenting.pdf>.
- [4] (2024), [Online]. Available: <https://docs.oracle.com/en/java/javase/22/docs/api/java.instrument/java/lang/instrument/package-summary.html>.
- [5] A. Precub, "Guide to Java Instrumentation," 2024. [Online]. Available: <https://www.baeldung.com/java-instrumentation>.
- [6] "OpenTelemetry Instrumentation for Java," [Online]. Available: <https://github.com/open-telemetry/opentelemetry-java-instrumentation>.
- [7] "Agent Configuration," [Online]. Available: <https://opentelemetry.io/docs/languages/java/automatic-configuration>.
- [8] "OneAgent Docs," [Online]. Available: <https://docs.dynatrace.com/docs/platform/oneagent/how-one-agent-works>.
- [9] "OneAgent," [Online]. Available: <https://www.dynatrace.com/platform/oneagent>.
- [10] "Datadog Docs," [Online]. Available: <https://docs.datadoghq.com/agent>.
- [11] "Datadog Agent," [Online]. Available: <https://github.com/DataDog/datadog-agent>.
- [12] "JSQLParser," [Online]. Available: <https://github.com/JSQLParser/JSqlParser>.
- [13] "Jersey," [Online]. Available: <https://eclipse-ee4j.github.io/jersey/>.
- [14] "ConcurrentHashMap," [Online]. Available: <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html>.
- [15] H. Wang and W. Fang, "A Trace Agent with Code No-invasion Based on Byte Code Enhancement Technology," 2019. [Online]. Available: <https://ieeexplore.ieee.org/document/9040725>.
- [16] "Apache JMeter," [Online]. Available: <https://jmeter.apache.org/>.
- [17] J. Nielsen, *Usability Engineering*. Nielsen Norman Group, 1993. [Online]. Available: <https://www.nngroup.com/books/usability-engineering/>.

WYKAZ RYSUNKÓW

2.1	Proces uruchamiania programów napisanych w języku programowania Java. Źródło własne.	9
2.2	Kod bajtowy odpowiadający programowi 1. Źródło własne.	12
2.3	Kod bajtowy odpowiadający programowi 2. Źródło własne.	13
2.4	Kod bajtowy odpowiadający zmodyfikowanemu programowi 1 przez agenta 4. Źródło własne.	19
2.5	Kod bajtowy odpowiadający zmodyfikowanemu programowi 1 przez agenta 6. Źródło własne.	19
3.1	Diagram bazy danych aplikacji testowej. Źródło własne.	21
3.2	Interfejs użytkownika aplikacji testowej. Źródło własne.	22
3.3	Serwis Zipkin prezentujący zgromadzone dane przez OpenTelemetry w czasie interakcji z interfejsem użytkownika aplikacji testowej. Źródło własne.	24
3.4	Operacje wykonane podczas realizacji punktu końcowego /createNewAction. Źródło własne.	25
3.5	Grupy, do których OneAgent przyporządkowuje zarejestrowane zapytania. Źródło własne.	26
3.6	Zapytania wchodzące w skład pierwszej grupy. Źródło własne.	26
3.7	Zapytania wchodzące w skład drugiej grupy. Źródło własne.	27
3.8	Zapytania wchodzące w skład trzeciej grupy. Źródło własne.	27
3.9	Zarejestrowane zapytania bazodanowe przez narzędzie Datadog. Źródło własne.	28
3.10	Wykres przedstawiający powiązanie zapytań z punktem końcowym na osi czasu. Źródło własne.	28
4.1	Diagram architektury przedstawiający proponowane narzędzie. Źródło własne.	29
4.2	Schemat blokowy przedstawiający zasadę działania funkcji <i>executeTransaction</i> . Źródło własne.	33
4.3	Schemat blokowy przedstawiający zasadę działania funkcji <i>processData</i> . Źródło własne.	35
4.4	Strona internetowa udostępniona przez serwer wyświetlająca zebrane dane przez agenta. Źródło własne.	37
4.5	Strona internetowa udostępniona przez serwer umożliwiająca podejrzenie transakcji wraz z parametrami. Źródło własne.	38
4.6	Strona internetowa udostępniona przez serwer wyświetlająca status transakcji złożonej z wielu zapytań. Źródło własne.	38

WYKAZ TABEL

2.1	Struktura pliku .class.	10
2.2	Przedrostki i przyroski mnemoników wskazujące na typ operandów.	10
2.3	Przykłady kodów operacji w systemie binarnym i szesnastkowym wraz z ich mnemonikami.	11
3.1	Przykładowe zapytania wykonywane w czasie obsługi żądania związanego z danym przyciskiem i jego punktem końcowym.	22
5.1	Parametry komputera do przeprowadzenia eksperymentów.	40
5.2	Wyniki z przeprowadzonych testów dla wszystkich narzędzi.	41
5.3	Wyniki z przeprowadzonych testów dla autorskiego agenta uruchomionego z parametrem o wartości <i>console</i>	42