LATVIJAS UNIVERSITĀTE
DATORIKAS FAKULTĀTE

JAVA INTEGRĀCIJU OPTIMIZĀCIJAS IESPĒJAS

BAKALAURA DARBS

Autors: **Arvis Taurenis**
Studenta apliecības Nr.: at13055
Darba vadītājs: Dr. dat. Uldis Straujums

RĪGA 2017

ANOTĀCIJA

Mūsdienās integrāciju izstrāde, augot sistēmu skaitam, starp kurām nepieciešama konstanta datu apmaiņa, kļūst arvien nozīmīgāka. Reāllaika datu integrācijās un augstas noslodzes integrācijās ļoti svarīga ir sistēmu veiktspēja, lai nodrošinātu maksimāli ātru datu apmaiņu, tādēļ ir svarīgi apzināties integrāciju un Java programmu optimizācijas iespējas.

Bakalaura darba mērķis ir izpētīt pieejamo teoriju par Java programmu, integrāciju un Java virtuālās mašīnas optimizācijas iespējām, kā arī praktiski lietot iegūtās zināšanas, testējot gan atsevišķu Java klašu, gan integrācijas komponentu, gan bibliotēku veiktspēju, noskaidrojot, ar kādam metodēm iespējams nodrošināt efektīvu Java integrāciju darbību. Autors darba ietvaros izstrādā integrāciju, lietojot teorijas daļā iegūtās zināšanas, un veic tās optimizāciju, demonstrējot optimizēšanas procesu un potenciālos veiktspējas ieguvumus.

Atslēgvārdi: optimizācija, integrācija, Java, HotSpot.

ABSTRACT

OPTIMIZATION POSSIBILITIES FOR JAVA BASED INTEGRATIONS

Nowadays software integrations are becoming more and more important for enterprises due to increasing amount of systems which needs constant data exchange. For integrations that deals with real-time data and has high workload it is necessity to process data as quickly as possible, therefore it is important to be aware of possibilities of optimizations that can be carried out in integrations and Java programs.

The goal of this Bachelor's thesis is to research available theory about Java program, Java virtual machine and Java integration optimization possibilities. Author practically tests performance of different Java language classes, integration components and different libraries to determine most effective methods to develop high performance Java integrations. Author in scope of this work practically develops and optimizes Java integration based on knowledge that's acquired in previously carried out research. Author shows the optimization process and potential performance gains from integration optimization.

Keywords: optimization, integration, Java, HotSpot.

SATURA RĀDĪTĀJS

APZĪMĒJUMU SARAKSTS

.NET Remoting - .NET programmēšanas valodas attālināto metožu izsaukšanas protokols

API – lietojumprogrammu programmēšanas interfeiss – koda un rīku kopums

C – zema līmeņa plaši lietota programmēšanas valoda, kura izstrādāta 1972. gadā

CORBA – protokols, kurš paredzēts attālinātu sistēmu savstarpējai komunikācijai

CPU – centrālais datora procesors, ierīce, ar kuru tiek apstrādātas visas programmatūras padotās instrukcijas

FTP/SFTP – datu pārraidei paredzēts tīkla protokols

HTTP – lietojumprogrammu līmeņa, hiperteksta pārsūtīšanas protokols.

IMAP/IMAPS - tīmekļa standarta protokols, lietots e-pastu piekļuvei un uzglabāšanai uz e-pastu serveriem

IRC – tīmekļa protokols, kurš tiek lietots komunikācijai ar tekstuālu informāciju

Java – programmēšanas valoda, kuru oriģināli izstrādāja uzņēmums Sun Microsystems, taču mūsdienās visas tiesības uz to pieder uzņēmumam Oracle

Java RMI – Java attālināto metožu izsaukšanas protokols

Java SE – Java sistēmas standarta versija, kurā nav iekļauti Java programmēšanai veidoti rīki

JDK – Java sistēmas versija, kura sastāv no Java SE un Java programmēšanai veidotiem rīkiem

JMS – Java ziņojumapmaiņas serviss, kurš darbojas Java MOM API

JPA – Java datu glabāšanas API Java programmu komunikācijai ar datubāzēm

JRE - Java darbības vide, kura iekļauj Java virtuālo mašīnu, uz kuras tiek darbinātas Java programmas

JSON – JavaScript objektu notācija, datu pieraksta formāts

LDAP – direktoriju vieglpiekļuves protokols, industrijas standarts sistēmu piekļuvei

POP3/POP3S – lietojumprogrammu līmeņa, tīmekļa standarta protokols komunikācijai ar e-pastu serveriem

RAM – datora operatīvā atmiņa

RMI – attālināts metožu izsaukuma standarts

RSS - tīmekļa plūsmas standartizēts formāts

SMTP/ SMTPS – lietojumprogrammu līmeņa tīmekļa protokols, kuru lieto kā e-pasta pārsūtīšanas protokolu

SOAP – tīmekļa protokols informācijas apmaiņai, plaši lietots tīmekļa servisos

SQL – standarta valoda vaicājumu rakstīšanai datubāzēs

SSH – tīmekļa protokols, kurš tiek lietots, lai nodrošinātu drošu apmaiņu starp divām tīkla ierīcēm

TCP – tīmekļa sistēmu komunikācijai un datu pārraidei veidots protokols, kurš nodrošina secīgu datu pārraidi

UDP – tīmekļa sistēmu komunikācijai un datu pārraidei veidots protokols, kurš nenodrošina secīgu datu pārraidi

URI – unikāls resursu identifikators

XML – datu attēlošanas valoda, industrijas standarts

IEVADS

Mūsdienās servera puses programmas reti ir izolētas no citām sistēmām, nereti ir nepieciešama konstanta datu apmaiņa starp sistēmām, lai tās varētu nodrošināt visu nepieciešamo funkcionalitāti, kuras vajadzīgas programmai. Šīs sistēmas var būt rakstītas dažādās programmēšanas valodās, strādāt ar dažādiem datu formātiem, atrasties dažādos reģionos, tikt lietotas dažādiem mērķiem. Šīs problēmas tiek risinātas ar integrāciju izstrādes palīdzību, tāpēc mūsdienās integrācijas kļūst arvien nozīmīgākas, īpaši liela mēroga uzņēmumos.

Tā iemesla dēļ, ka daudzos gadījums integrācijām ir jāstrādā ar reāllaika datiem, dati ir jāapstrādā ar minimālu aizturi, tāpēc izstrādātājiem ir svarīgi veidot integrācijas ar maksimāli augstu veiktspēju, atbilstoši pieejamajiem resursiem. Šo iemeslu dēļ autors par darba mērķi ir uzstādījis: atrast un praktiski pārbaudīt iespējamās optimizācijas iespējas Java valodā veidotām integrācijām, kas ietver gan integrāciju arhitektūras pareizo izvēli, gan Java koda un Java virtuālās mašīnas optimizācijas, kā arī praktiski demonstrēt iegūtās zināšanas, optimizējot izstrādātu integrāciju.

Darbā ar terminu optimizācija saprotama integrācijas veicamā darba, neatkarīgi no tā tipa, ātrāka izpilde, nepatērējot papildus resursus – operatīvo atmiņu un procesora noslogotību. Darbā autors apskata optimizācijas teoriju par standarta Java kodu, Oracle HotSpot Java virtuālās mašīnu un integrāciju veidošanas arhitektūru. Bez teorijas darbā tiek praktiski testētas dažādas integrācijās bieži lietotas tehnoloģijas un komponenti, noskaidrojot efektīvāko to izmantojumu, integrāciju darbības ātruma uzlabošanai.

Darbs ir sadalīts nodaļās un apakšnodaļās. Pirmajā nodaļā tiek sniegts teorētisks ieskats par integrācijas šabloniem un integrāciju izstrādi. Otrajā nodaļā tiek apskatīti populārākie integrāciju ietvari Java valodā. Trešajā nodaļā tiek apskatītas Java virtuālās mašīnas optimizācijas iespējas. Ceturtajā nodaļā tiek apskatītas Java koda optimizācijas iespējas un autora veiktie praktiskie eksperimenti integrācijas komponentu veiktspējai. Piektajā nodaļā tiek apskatīti profilēšanas un veiktspējas mērījumu rīki priekš Java valodā izstrādātām programmām. Sestajā nodaļā ir aprakstīta autora praktiski izstrādāta integrācija un tās optimizēšana.

1. INTEGRĀCIJAS ŠABLONI

Servera puses programmas reti ir pilnībā pašpietiekamas un spēj korekti funkcionēt izolētas no citām sistēmām. Vienas kompānijas ietvaros nereti ir neskaitāmas pašu veidotas, mantotas vai iegūtas no trešās puses, šīs sistēmas var būt programmētas, lietojot dažādas programmēšanas valodas, darboties uz dažādām operētājsistēmām, strādāt dažādos operētājsistēmas līmeņos un darboties uz serveriem dažādās vietās, šāda situācija var rasties vairāku iemeslu dēļ:

- dažādas sistēmas daļas tiek veidotas dažādos laika posmos, kas noved pie dažādu, konkrētajā laika posmā atbilstošāko, tehnoloģiju izvēles;
- ārēju sistēmu vai produktu iepirkšana, laika taupīšanas vai finansiālu apsvērumu nolūkos;
- sistēmas, īpaši liela mēroga uzņēmumos, programmē dažādi speciālisti, izvēloties tehnoloģijas, kurās tiem ir lielākā pieredze;
- programmatūras izstrādes un palaišanas produkcijā laiks ir nozīmīgāks, par ērtu integrācijas nodrošināšanu potenciālām nākotnes vajadzībām [1].

Integrācijas šabloni ir industrijas standarta pieejas datu integrācijai un bieži sastopamajām problēmām integrāciju izstrādē.

1.1. Programmu datu integrācija

Pastāv vairāki veidi, kā vairākām sistēmām savstarpēji veikt datu apmaiņu, katram no šiem veidiem ir savas priekšrocības un trūkumi noteiktos gadījumos, šī iemesla dēļ vienas integrācijas ietvaros var tikt lietoti vairāki datu integrācijas veidi. Galvenokārt tiek izšķirti četri datu apmaiņas šabloni – failu apmaiņa, koplietošanas datubāze, ziņojumapmaiņa un attālinātie metožu izsaukumi.

1.1.1. Failu apmaiņa

Failu lasīšana un rakstīšana ir ļoti plaši lietota gandrīz jebkurā programmēšanas valodā, tāpēc datu integrācija, lietojot failus, tiek uzskatīta par vienkāršāko integrācijas veidu. Failu apmaiņa, lai gan salīdzinoši vienkārša, var prasīt salīdzinoši daudz resursus, it īpaši, ja apmaiņa starp datiem ir nepieciešama bieži, piemēram, simtiem reižu minūtē, tad programma, kura raksta datus patērēs ievērojami daudz resursus, jo būs nepieciešams nepārtraukti veidot jaunus failus un nodrošināt unikālu failu vārdu veidošanu. Programmai, kura lasa datus, būs nepārtraukti

jāpārbauda, vai nav saņemts jauns datu fails apstrādei, kā arī jābūt loģikai, lai pārbaudītu, ka visi faili ir saņemti, lai izvairītos no datu nekonsekvences. Šāds datu apmaiņas veids parasti tiek lietots ar novecojušām, mantotām sistēmām, kuras netiek aktīvi izstrādātas. Šādām sistēmām parasti jau ir noteikts datu formāts, kurā dati tiek rakstīti, un sistēmām, kurām ir nepieciešams integrēt šos datus ir jāpielāgojas šim formātam, kas nereti nozīmē sarežģītu datu apstrādi [3].

Failu apmaiņa ir ērta tādos gadījumos, kad datu apmaiņa starp sistēmām ir jāveic reti, piemēram, reizi diennaktī. Šādos gadījumos viena no programmām visus pa dienu saņemtos jaunos datus ieraksta vienā failā un otra programma tos, lietojot datu sadalīšanas šablonu, sadala un apstrādā katru individuālo datu objektu, parasti ārpus darba laika, lai neatstātu iespaidu uz sistēmas veiktspēju diennakts stundās, kad sistēma tiek aktīvi lietota.

1.1.2. Koplietošanas datubāze

Koplietošanas datubāze, līdzīgi kā failu apmaiņa, arī ir plaši lietota, plašās relāciju datu bāzu izplatības dēļ. Gandrīz jebkurā no modernajām programmēšanas valodām atbalsts darbam ar datu bāzēm ir jau iebūvēts valodā vai pieejams trešo pušu bibliotēku veidā.

Ja datu bāzu tabulu veidošanas stadijā tiek pievērsta nepieciešamā uzmanība datiem, kuri tiks rakstīti tabulās, tad, veidojot dažādus datu lauku ierobežojumus, iespējams izvairīties no nesaderīgu datu apmaiņas starp sistēmām. Vēl koplietošanas datubāzes integrācijas veida priekšrocība ir tā, ka sistēmas, kuras lieto datubāzi datu ieguvei, var lasīt tikai tos datus, kuri ir nepieciešami, tādējādi samazinot informācijas daudzumu, kurš jāpārsūta pār tīklu [2].

Viens no koplietošanas datubāzes trūkumiem ir tabulu dizaina veidošana, lai apmierinātu visas integrējamās programmas, ņemot vērā programmas, kuras nākotnē varētu lietot kādu no tabulām. Koplietojamai datubāzei var būt negatīva ietekme uz sistēmu veiktspēju, kuras lieto šo datubāzi, it īpaši, ja ar datubāzi savstarpēji tiek integrētas vairākas programmas, kuras var vienlaicīgi rakstīt vai mainīt tabulu datus. Trešais šī datu integrācijas veida trūkums ir tabulu dizaina maiņa. Ja ir nepieciešams mainīt tabulas dizainu, piemēram, lai pievienotu papildus kolonas vai mainītu datu tipus vai to ierobežojumus, tad visās programmās, kuras lieto šo tabulu, var būt nepieciešams veikt izmaiņas kodā, lai sistēmas būtu spējīgas funkcionēt.

1.1.3. Attālinātie metožu izsaukumi

Ar attālinātajiem metožu izsaukumiem iespējams izsaukt funkciju no citas programmas, kura var atrasties uz cita servera, padodot visus nepieciešamos datus. Attālināto metožu

izsaukumu tehnoloģija ir pieejama vairākām plaši lietotām programmēšanas valodām. Izplatītākās attālināto metožu izsaukumu tehnoloģijas ir Java RMI, CORBA, .NET Remoting u.c., šīm tehnoloģijām nereti ir pieejamas tādas funkcijas kā transakciju veidošana [4].

1.1.4. Ziņojumapmaiņa

Ziņojumapmaiņas integrācijas mehānisms ir visplašāk lietotais datu integrācijas veids mūsdienās. Šī datu apmaiņa tiek veikta, sūtot un saņemot, parasti maza lieluma, datu paketes no ziņojumapmaiņas sistēmas.

Lietojot šo pieeju, tiek nodrošināts tas, ka savstarpēji integrējamās programmas nav atkarīgas viena no otras, jo visa komunikācija norit starp trešās puses sistēmu, tādējādi komunikācija starp sistēmām ir asinhrona. Parasti ziņas tiek sūtītas vienā no diviem izplatītākajiem formātiem: XML vai JSON. Ziņojumapmaiņas sistēmās ir parasti pieejami divi datu apmaiņas veidi: tēmas un rindas.

Rinda nodrošina, ka katru rindā ierakstīto ziņu saņems tieši viens abonents - sistēma, kura ir reģistrēta kā rindas datu apstrādātājs. Rindas var tikt lietotas kā slodzes stabilizators, ja slodzes uzlabošanas nolūkos darbojas vairākas vienas sistēmas instances uz dažādiem serveriem, tad tiek nodrošināts, ka visas ziņas tiks apstrādātas pareizā secībā un neviena no ziņām netiks apstrādāta vairāk kā vienu reizi. Tēmas nodrošina, ka katru rindā ierakstīto ziņu saņems visi tēmas abonenti, kas ir svarīgi, ja savā starpā ir jāintegrē vairāk kā divas sistēmas. Rindas un tēmas tiek sauktas par ziņojuma kanāliem [5].

Visi mūsdienu integrācijas ietvari ir primāri veidoti integrācijām, lietojot ziņojumapmaiņas sistēmas.

1.2. Ziņojumapmaiņas kanāli

Ziņojumapmaiņas kanāli tiek lietoti tad, kad ir nepieciešama komunikācija starp divām dažādām sistēmām, pār tīklu. Ziņojumu apmaiņa integrācijās var noritēt vairākos veidos, atkarībā no integrācijas vajadzības.

1.2.1. Divu punktu kanāls

Divu punktu kanāla šablons paredzēts, lai nodrošinātu to, ka tieši viena cita sistēma saņems izsūtīto ziņu. Šablons var tikt lietots gan lai sūtītu ziņas uz ziņojumapmaiņas sistēmām, gan lai ievietotu ierakstus datubāzē, gan lai veiktu attālinātos metožu izsaukumus. Ja kanālā ziņas

saņēmēja pusē ziņas gaida vairāk par vienu sistēmu, tad šis šablons nodrošina to, ka ziņu apstrādātu tikai viena sistēma. Lietojot divu punktu kanālu ar vairāk kā vienu sistēmu ziņas saņemšanas pusē, iespējams vienkārši implementēt slodzes stabilizatoru, tā kā ziņu apstrādās tikai viena no sistēmām, visas sistēmas, kuras gaida ziņu uz šī kanāla, tiek sauktas par konkurējošiem ziņu patērētājiem [6].

1.2.2. Publicēšanas-abonēšanas kanāls

Publicēšanas-abonēšanas kanāls ļauj vienam ziņas publicētājam nosūtīt vienu un to pašu ziņu visiem abonentiem – visām sistēmām, kuras gaida ziņas no publicētāja. Šis šablons parasti tiek lietots ar ziņojumapmaiņas sistēmām - veidojot tēmas, vai ar vairākām rindām virtuālajā atmiņā, kuras ir paredzētas integrācijas plūsmu iekšējai darbībai [7]. Katra ziņa tiek uzskatīta par patērētu jeb tādu, ko sistēma var dzēst no iekšējās atmiņas, tikai tad, kad visi abonenti ir saņēmuši ziņu, kas ļauj katram abonentam apstrādāt ziņu sev izdevīgā laika brīdī.

1.2.3. Nederīgu ziņu kanāls

Nederīgu ziņu kanāls domāts tam, lai gadījumā, ja ziņas saņēmējs nevar, dažādu iemeslu dēļ, apstrādāt ziņu, jo uzskata to par nederīgu, būtu kur to novadīt vēlākai novērtēšanai. Ja nederīgā ziņa tiktu vienkārši ignorēta, nebūtu iespējams vēlāk noskaidrot, kas tā par ziņu un kāpēc bijusi nederīga, ja nederīgo ziņu saņēmējs sūtītu atpakaļ pa kanālu, kur ziņa tika saņemta, tad šis vai kāds no citiem potenciālajiem saņēmējiem atkal mēģinātu apstrādāt ziņu, tādējādi aizkavējot citu ziņu apstrādi, tāpēc ir nepieciešams nederīgo ziņu kanāls [8].

1.2.4. Mirušu ziņu kanāls

Mirušu ziņu kanāls paredzēts ziņām, kuras publicētājs, dažādu iemeslu dēļ, piemēram, ziņojumapmaiņas sistēmas rinda nav pienācīgi konfigurēta, nevar piegādāt. Ziņas no kanāla var tikt vēlāk pārbaudītas un izlabotas vai, ja problēma ir bijusi ārējā sistēmā, var uzreiz atkārtot ziņas piegādes mēģinājumu.

1.2.5. Garantētas piegādes kanāls

Garantētās piegādes kanāls domāts kritiska svarīguma ziņu piegādei. Ar šo kanālu tiek garantēta jebkuras ziņas piegāde pat, ja ziņojumapmaiņas sistēma pārtrauc darbību. Garantēta piegāde tiek panākta ar ziņu rakstīšanu diskā gan ziņas sūtītāja, gan saņēmēja galā. Ziņa tiek uzskatīta par veiksmīgi nosūtītu tikai tad, kad ziņas saņēmējs ir ierakstījis ziņu diskā, parasti datu

bāzē. Integrācijās, kuras apstrādā lielu daudzumu ar ziņām, var tikt aizņemta ievērojama diska vieta, ziņu glabāšanai, tāpēc šis šablons tiek lietots tikai kritiskos gadījumos.

1.3. Ziņu maršrutēšana

Integrācijām nereti ir jāspēj apstrādāt dažāda veida ziņas, kur visas ziņas ir loģiski saistītas, vai atkarībā no konkrētās ziņas satura jāspēj apstrādāt ziņas dažādos veidos, piemēram, ja ziņā trūkst kāds integrācijai nepieciešams datu lauks, tad var būt nepieciešamība izsaukt kādu servisu, lai uzzinātu trūkstošo informāciju. Ziņu maršrutēšanas šablonu grupa atbild par ziņu maršrutēšanu uz tām paredzēto apstrādes plūsmu. Izšķir divas lielas maršrutētāju grupas – vienkāršie un kompozīta maršrutētāji, kur vienkāršie maršrutētāji novirza ziņu pa nepieciešamo ziņas apstrādes kanālu un kur kompozīta maršrutētāji ir vairāku vienkāršo maršrutētāju kombinācija sarežģītu ziņu apstrādes plūsmu izstrādei [9].

1.3.1. Vienkārši maršrutētāji

Satura bāzēti maršrutētāji novirza ziņu pa kādu no iepriekš definētiem apstrādes kanāliem, balstoties uz ziņas saturu. Satura bāzēti maršrutētāji katru saņemto ziņu maršrutē uz atbilstošo galamērķi. Satura bāzēti maršrutētāji atvieglo ziņas producenta neatkarību no ziņas galamērķiem, producentam nav jāzina precīzs katras ziņas vajadzīgais galamērķis, tam ir tikai jānogādā ziņa maršrutētājam, kurš veic tālāko ziņu maršrutēšanu.

Ziņu filtri ir satura bāzētu maršrutētāju speciāls gadījums. Filtri, pārbaudot ziņas saturu, maršrutē ziņu tālākai apstrādei tikai, ja ziņas saturs atbilst noteiktiem kritērijiem. Ja ziņa neatbilst noteiktajiem kritērijiem, tad tā netiek tālāk apstrādāta. Filtri domāti, lai izfiltrētu nevajadzīgās vai nepilnīgās ziņas no konkrētās ziņu apstrādes plūsmas. Ziņu filtri lielākajā daļā gadījumu neuzglabā nekādu stāvokli, tie pārbauda konkrēto ziņu un vai nu nosūta uz galamērķi, vai nē. Filtrus, kuros netiek glabāts nekāds stāvoklis, iespējams veidot vairākās instancēs, lai apstrādātu ziņas paralēli. Taču ir iespējami arī filtri, kuriem ir jāuzglabā stāvoklis, piemēram, iepriekš filtrēto ziņu vēsture, lai filtrētu dublikātu ziņas.

Adresātu saraksts arī ir satura bāzēta maršrutētāja speciāls gadījums, tas, līdzīgi kā iepriekš apskatītie maršrutētāji, pārbauda ziņas saturu un maršrutē ziņu tālākai apstrādei atkarībā no ziņas satura, taču ar adresātu saraksta šablonu šo ziņu iespējams nosūtīt uz nevienu vai vairākiem galamērķiem. Saraksts, kuriem sūtīt ziņu, var būt gan statisks, iepriekš definēts, gan dinamisks,

kurš tiek izveidots katrai ziņai atsevišķi, piemēram, sūtot e-pastu, iespējams norādīt vairākus e-pasta saņēmējus dinamiski.

Ziņu sadalītājs ir speciāls maršrutēšanas šablons, ar kura palīdzību saraksts ar objektiem tiek sadalīts atsevišķos objektos, lai apstrādātu katru no atsevišķajiem objektiem kā individuālu ziņu. Pēc sadalīšanas katra individuālā ziņa var tikt maršrutēta un apstrādāta neatkarīgi no citām.

Ziņu apkopotājs ir pretējs šablons ziņu sadalītājam, tas saņem vairākas ziņas, gaidot, kad tiks izpildīti apkopošanas kritēriji, piemēram, saņemtas desmit viena veida ziņas, tad apkopo ziņas sarakstā un nodod šo sarakstu tālākai apstrādei. Apkopotājs bieži tiek lietots pie paralēlu ziņu daļu apstrādes, lai pēc paralēli apstrādātajām ziņām, tās varētu atgriezt sākotnējā saraksta izskatā.

Secības sakārtotāja šablons paredzēts, apkopojot ziņas, lai tās sakārtotu pareizā secībā – saraksta sākotnējā ziņu secībā. Šāds šablons ir nepieciešams, jo, apstrādājot individuālās ziņas paralēli, integrācija nevar garantēt, ka ziņas tiks apstrādātas pareizā secībā, jo ziņas, atkarībā no to satura, var tikt maršrutētas pa dažādām ziņu apstrādes plūsmām [9].

Ziņu apkopotājs un secības sakārtotājs ir vienīgie divi maršrutētāju veidi, kuriem ir jāuztur iekšējs stāvoklis, jo tiem ir jāgaida, līdz tiks izpildīti kritēriji ziņu apstrādei, kas nozīmē, ka tiem ir nepieciešama iekšēja atmiņa, kur glabāt šīs ziņas, līdz būs iespējama to apstrāde.

1.1. tabula

Vienkāršie ziņu maršrutētāji [10]

Šablons	**Patērēto ziņu skaits**	**Publicēto ziņu skaits**	**Stāvoklis**
Satura bāzēts maršrutētājs	1	1	Nē
Filtrs	1	0 vai 1	Jā/Nē
Adresātu saraksts	1	0 vai vairākas	Nē
Ziņu sadalītājs	1	vairākas	Nē
Ziņu apkopotājs	vairākas	1	Jā
Ziņu sakārtotājs	vairākas	vairākas (tikpat, cik patērētas)	Jā

1.3.2. Kompozīta maršrutētāji

Sarežģītākās integrācijas plūsmās vienkāršie maršrutētāju veidi ir jāapvieno, lai būtu iespējams panākt vēlamo maršrutēšanas rezultātu.

Kompozīt-ziņu apstrādātāja šablons apvieno zinu sadalītāja un ziņu apkopotāja šablonus, nereti bez šiem šabloniem tiek lietots arī ziņu sakārtotājs. Šis šablons domāts, lai sadalītu ziņu sarakstu, apstrādātu katru individuālo saraksta objektu atsevišķi un tad apvienotu apstrādātās ziņas, ar iespējamu ziņu kārtošanu, atpakaļ vienā atbildes ziņā. Šis šablons ir tiek ļoti bieži lietots pie sarakstu apstrādes.

Izkliedētāja-savācēja šablons apvieno adresāta saraksta un ziņu apkopotāja šablonu. Ar šo pieeju paredzēta vienas un tās pašas ziņas izsūtīšana vairākiem ziņas saņēmējiem, lai pēc tam apvienotu ziņas vienā kopīgā atbildē, balstoties uz iepriekš norādītiem biznesa likumiem. Šo šablonu bieži lieto, ja vienai ziņai iespējamas vairākas atbildes, piemēram, vienam preces pasūtījumam var būt dažādas cenas un pieejamība pie dažādiem preču piegādātājiem.

Maršrutēšanas veidlapas šablons nodrošina ziņas maršrutēšanu uz vairākiem ziņas apstrādes soļiem atkarībā no saņemtās ziņas, ziņas apstrādes laikā tiek veikta dinamiska maršrutēšana. Piemēram divām identiskām ziņām integrācijā var tikt veikta dažāda apstrāde, atkarība no kādas galvenes vērtības, kurā var tikt norādīts, kurus apstrādes soļus pildīt, kurus izlaist konkrētajai ziņai.

Bez apskatītajiem kompozīta maršrutētājiem iespējamas arī citas vienkāršo maršrutētāju kombinācijas, piemēram, vairāki secīgi filtri, kur katrs filtrs atbilst noteiktai biznesa prasībai, vai citas kombinācijas.

1.4. Ziņu transformācija

Savstarpēji integrējamās programmās ļoti retos gadījumos tiek lietots viens un tas pats datu tips, parasti dati no ārējām sistēmām ir jāpārveido uz iekšēju formātu pirms datu apstrādes. Tā iemesla dēļ, ka integrācijas parasti tiek veidotas starp jau esošām programmām, iespēja mainīt programmās lietotos datu modeļus nav.

1.4.1. Datu bagātināšana

Saņemot datus no vienas sistēmas, lai tos nosūtītu citai sistēmai, nereti dati nav pilnīgi un ir nepieciešams tos papildināt. Datu bagātinātājam, lietojot informāciju no saņemtās ziņas,

piemēram, kādu identifikācijas lauku, ir jāiegūst dati no kādas ārējas sistēmas, lai papildinātu saņemto datu objektu, pirms tālākas ziņas apstrādes. Visbiežāk papildus nepieciešamie dati tiek iegūti no kāda no trijiem avotiem:

- veicot aprēķinus no pieejamajiem datiem, piemēram, ja ziņa jāpārveido uz formātu, kuram jānorāda ziņas izmērs, tad to ir iespējams aprēķināt un pievienot sūtāmajai ziņai;
- integrācijas darbības vides, piemēram, bieži datos nepieciešams norādīt laika zīmogu, kas norādītu precīzu laiku, kad ziņa saņemta vai apstrādāta, lai būtu iespējama apstrāde trasēšana kļūdu gadījumā;
- citas sistēmas, šāda datu bagātināšana ir visizplatītākā, tā iekļauj datu ieguvi no dažādiem ārējiem datu avotiem, piemēram, datubāzes, faila, tīmekļa servisa vai lietotāja, kurš ievada datus manuāli.

Bieži integrācijās sastopamas situācijas, kad dati jāpapildina no vairākiem dažādiem avotiem, lai iegūtu visu nepieciešamo informāciju, datu bagātināšana, protams, palēnina integrācijas darbību, jo parasti ir jāveic komunikācija ar citām sistēmām pār tīklu.

1.4.2. Datu filtrēšana

Datu filtrēšanas šablons ir pretējs datu bagātināšanas šablonam, tas paredzēts gadījumiem, kad saņemtajā ziņā ir dati, kuri integrācijai nav nepieciešami. Datu filtrēšana galvenokārt tiek veikta divu iemeslu dēļ. Pirmais iemesls - datu filtrēšana tiek veikta drošības nodrošināšanas nolūkos, piemēram, iespējami gadījumi, kad datus, kuri tiek saņemti no kāda avota, nedrīkst sūtīt tālāk uz citām sistēmām vai glabāt, tāpēc pirms jebkādas datu apstrādes ir jāveic datu filtrēšana. Otrs iemesls datu filtrēšanas šablona lietošanai ir integrācijas ātrdarbības uzlabošanas nolūkos. Apstrādājot un pārsūtot pār tīklu tikai tos datus, kurus nepieciešams, netiek patērēti lieki sistēmas resursi, kā rezultātā integrācijas ātrdarbība netiek nevajadzīgi bremzēta [11].

Datu filtrēšanas šablons bez lieku datu atmešanas paredzēts arī datu modeļa vienkāršošanai, piemēram, dati lielās sistēmās bieži satur daudzus līmeņus ar pakārtotiem datiem, kurus nepieciešams pārveidot plakanā datu struktūrā, piemēram, rakstīšanai datu bāzē.

1.4.3. Īslaicīgā filtrēšana

Īslaicīgā filtrēšana ir šablons, kura funkcionalitāte ir datu bagātināšanas un datu filtrēšanas apvienojums. Datus, kuri nav nepieciešami ziņas apstrādes plūsmā, iespējams noglabāt uz laiku,

lai padarītu apstrādājamo datu objektu izmēros mazāku, tādējādi uzlabojot integrācijas veiktspēju. Apstrādes plūsmā, darbojoties tikai ar nepieciešamo datu komplektu, iespējams arī atvieglot integrācijas atkļūdošanu izstrādātājiem. Ar īslaicīgās filtrēšanas šablonu iespējams paslēpt sensitīvus datus, kuri, iespējams, nepieciešami kādai no ārējām sistēmām, bet kurus nedrīkst izpaust citām sistēmām, ar kurām komunicē integrācija ziņas apstrādes brīdī.

1.4.4. Datu normalizēšana

Datu normalizēšana ir process, kurā dati no dažādiem formātiem, bet ar vienu un to pašu informāciju tiek pārveidoti vienotā formātā, lai tos būtu iespējams apstrādāt. Dati integrācijām var tikt sūtīti no dažādām partneru sistēmām, un, ja konkrētais uzņēmums nav tādā pozīcijā, lai uzspiestu savu datu formātu visiem partneriem, tad integrācijai ir nepieciešams spēt apstrādāt visus datu formātus. Datu normalizēšana bieži tiek lietota kopā ar ziņu maršrutēšanu, kur atkarībā no saņemtā datu formāta katra ziņa tiek maršrutēta uz nepieciešamo datu konvertācijas plūsmu, lai pēc tam varētu apstrādāt datus no visiem datu avotiem vienā plūsmā.

1.5. Ziņojumapmaiņas galapunkti

Ar ziņojumapmaiņas galapunktiem tiek savienota ārēja sistēma ar integrāciju, gan ienākošām, gan izejošām ziņām. Pastāv vairāki klientu jeb ziņojumapmaiņas galapunktu veidi, kuri aprakstīti turpmākajā nodaļā.

1.5.1. Transackionāls klients

Ziņojumapmaiņas sistēmās iekšēji, katrai ziņas sūtīšanai vai saņemšanai, lieto transakcijas, lai izolētu katru ziņu, taču nereti ir nepieciešams kontrolēt transakcijas integrācijas pusē. Integrācijas pusē nepieciešamība pēc transakciju kontroles var rasties:

- ziņas sūtīšanas-saņemšanas pāriem, piemēram, pieprasījuma-atbildes protokola implementēšanai;
- ziņu grupas sūtīšanai – uzskata transakciju par veiksmīgi izpildītu tikai tad, ja visa grupa ar ziņām veiksmīgi nosūtīta;
- ziņas–datubāzes ieraksta koordinēšanai, piemēram, kad nepieciešams veikt datubāzes ierakstu pēc katras veiksmīgi nosūtītās vai saņemtās ziņas[12].

Šādi, kombinējot ziņojumapmaiņas transakcijas ar ārējām lielāka mēroga transakcijām, iespējams ērti izpildīt sarežģītas darbību virknes.

1.5.2. Sinhronais ziņu patērētājs

Sinhronais klients pats nosaka, kad lasīt ziņas, kuras gaida apstrādi atbilstošajā kanālā, tas periodiski pārbauda kanālu un, atrodot gaidošu ziņu, ielasa to un nodod tālākai apstrādei. Šāda veida klientam var konfigurēt kanāla pārbaudīšanas periodu atbilstoši vajadzībām. Šādi klienti kontrolē tieši vienas ziņas vienlaicīgu apstrādi, ja kanālā gaida vairākas ziņas, tās tiek ielasītas secīgi, katra nākamā ziņa tiek ielasīta tikai tad, kad klients ir gatavs jaunas ziņas apstrādei. Vairāku vienlaicīgu ziņu apstrādei parasti iespējams konfigurēt pavedienu skaitu klientam [12].

Šo iemeslu dēļ, ka klients pats pārbauda kanālu periodiski un tas bloķē jaunu ziņu ielasīšanu, līdz ielasītā ziņa ir apstrādāta, šāda klienta lietošana var būt neefektīva un samazināt integrācijas veiktspēju.

1.5.3. Notikumu bāzēts ziņu patērētājs

Notikumu bāzēts klients darbojas pretēji sinhronajam klientam, tas nevis pats pārbauda ziņu saņemšanas kanālu, bet gaida, kad to izsauks kanāls. Ar šādu pieeju visa atbildība par ziņu piegādi tiek uzlikta ziņu kanālam, tādējādi tiek ietaupīti resursi integrācijas pusē. Tipiski šāda veida ziņu patērētājs pēc vienas saņemtas ziņas gaida, kad šī ziņa tiks apstrādāta un tikai pēc tam ir atvērts jaunu ziņu saņemšanai, tādā gadījumā kanāls gaida līdz ziņu patērētājs ir gatavs saņemt jaunas ziņas. Taču, ja ziņu apstrāde plūsma ir veidota vairāku paralēlu ziņu apstrādei, ziņu patērētāju var konfigurēt, lai tas ļautu saņemt vairāk par vienu ziņu vienlaicīgi.

1.5.4. Konkurējoši ziņu patērētāji

Konkurējoši ziņu patērētāji ir vairāki patērētāji, kuri lasa ziņas no viena un tā paša kanāla. Kad pa kanālu tiek piegādāta ziņa, jebkurš no izveidotajiem patērētājiem var nolasīt ziņu un nodot tālāk apstrādei. Šādi patērētāji darbojas tikai uz divu punktu kanāla, jo, ja šo šablonu mēģinātu ieviest uz publicēšanas-abonēšanas kanāla, tad ziņas saņemtu visi patērētāji.

Katrs no patērētājiem strādā uz atsevišķa pavediena, lai būtu iespējama paralēla ziņu ielasīšana. Kanāls nodrošina to, ka katrs no patērētājiem saņem citu ziņu, pašiem patērētājiem savstarpēji nav nepieciešams veikt sinhronizāciju.

1.5.5. Ziņu dispečeri

Ziņu dispečeri ir domāti gadījumiem, kad pa vienu kanālu var tikt saņemtas vairāku veidu ziņas. Ziņu dispečers saņem visas ienākošās ziņas un izlemj, kur tālāk katru no ziņām sūtīt tālākai apstrādei. Dispečers strādā kā starpposms starp ziņu patērētājiem un ziņu kanālu.

1.5.6. Selektīvs ziņu patērētājs

Selektīvs ziņu patērētājs ir patērētājs, kurš neielasa visas saņemtās ziņas, bet tikai tās, kuras atbilst noteiktiem kritērijiem, kurus var konfigurēt pēc vajadzības. Šāda veida patērētājs pēc būtības izturas kā filtrs, kurš laiž cauri tikai tās ziņas, kuras ir nepieciešamas integrācijai. Šāda veida patērētāji var tikt lietoti kā konkurējoši ziņu patērētāji, lai nodrošinātu to, ka katrs no patērētājiem patērē tikai vajadzīgās ziņas [12].

Selektīvs ziņu patērētājs var tikt aizvietots ar ziņu filtru, kad patērētājs ielasa visas saņemtās ziņas, taču uzreiz pēc ielasīšanas ziņas tiek filtrētas un nevajadzīgās tiek atmestas.

1.5.7. Izturīgs abonents

Izturīgs abonents ir tāds ziņu patērētājs, kuram tiek garantēts tas, ka tam tiks piegādātas visas paredzētās ziņas, pat tās, kuras tiek saņemtas laikā, kad abonents nedarbojas, piemēram, kad programma, kura patērē ziņas, ir izslēgta. Ziņojumapmaiņas sistēma, zinot to, ka konkrētais abonents ziņas nav saņēmis, glabā ziņas, līdz ir iespējams tās piegādāt. Ziņas netiks piegādātas abonentam tikai tad, ja tiek padota speciāla komanda, kura atreģistrē abonentu no kanāla.

Šāds šablons ir bieži lietots, jo tas nodrošina visu ziņu piegādi, ja integrācija ir uz laiku izslēgta, kas ir ļoti kritiski daudzās nozarēs, piemēram, finanšu nozarē.

1.5.8. Unikālu ziņu saņēmējs

Parasti ziņojumapmaiņas sistēmas sūta ziņas, lietojot HTTP protokolu, un ziņa tiek uzskatīta kā saņemta tikai tad, kad tiek saņemta apstiprinājuma atbilde. Taču apstiprinājuma atbilde, dažādu iemeslu dēļ, var tikt izsūtīta, bet nesaņemta, kas var novest pie atkārtotas ziņas izsūtīšanas. Unikālu ziņu saņēmējs nodrošina to, ka, ja tiks saņemtas vairākas vienādas ziņas, tad tas nodos katru unikālo ziņu apstrādei tikai vienu reizi. Ziņu saņēmējs iekšēji uztur sarakstu ar visām saņemtajām ziņām un pirms ziņas nodošanas apstrādei pārbauda, vai ziņa ir unikāla.

2. JAVA INTEGRĀCIJAS IETVARI

Integrācijas starp dažādām sistēmām parasti tiek realizētas, lietojot kādu no integrācijas ietvariem vai programmatūras platformu, kura veidota priekš integrāciju izstrādes. Abas minētās pieejas ir domātas, lai izstrādātājiem atvieglotu integrāciju izstrādi, parasti šajos ietvaros un platformās ir implementēti integrācijas šabloni, kurus izstrādātāji var pielietot. Bez ietvariem integrāciju kods lielā daļā gadījumu būtu ļoti apjomīgs un sarežģīts tā iemesla dēļ, ka būtu nepieciešams liels koda apjoms priekš savienošanās ar katru ārējo sistēmu. Mūsdienu ietvari bez papildus nepieciešama koda parasti atbalsta populārākos sistēmu veidus, ar kuriem jāveic integrācija, piemēram, relāciju un bez-relāciju datubāzes, rindas, failu sistēmas, tīkla servisus.

Industrijā lielākā daļa integrāciju ir veidotas tieši uz Java virtuālās mašīnas bāzētām valodām vai platformām. Populārākie Java integrāciju ietvari ir Apache Camel un Spring Integration, kuri tiks sīkāk apskatīti tālāk nodaļā, taču bez ietvariem ir arī vairākas integrāciju platformas.

Oracle SOA Suite ir rīku komplekts, kurš balstās uz integrāciju izstrādi, tīmekļa servisu formā, lietojot SOAP protokola servisus. Rīku komplekts ir maksas produkts, kuram ir nepieciešama licenču iegāde. Kā pozitīvais šajā rīku komplektā ir *vilkt un nomest* principa lietošanas iespējamība servisu izstrādē, kas atvieglo vienkāršu integrāciju izstrādi. Oracle SOA Suite ietilpst vairāki komponenti:

- Oracle BPEL Process Manager – biznesa procesu motors, priekš BPEL servisu izstrādes un izvietošanas uz speciāla Oracle SOA servera;
- Oracle Service Bus (OSB) – SOAP protokola web servisu izstrāde un izvietošana uz speciāla Oracle Service Bus servera;
- JDeveloper – izstrādes vide, kurā ir iespējams izstrādāt BPEL procesus, lietojot *vilkt un nomest* principu;
- OEPE – paplašināta Eclipse izstrādes vide, ar iespējām veidot OSB servisus, lietojot *vilkt un nomest* principu;
- Oracle B2B – platforma starp uzņēmumu sadarbībai [13].

JBoss Enterprise SOA ir rīku komplekts integrāciju izstrādei, kuru izstrādā RedHat uzņēmums. Rīku komplekts ir atvērtā pirmkoda uz Java programmēšanas valodas bāzēts

komplekts. Arī JBoss Enterprise SOA komplekts sastāv no vairākiem komponentiem, biežāk lietotie no tiem:

- JBoss ESB – servisu serveris JBoss Java integrācijām;
- Business Rules Engine – programmatūra biznesa likumu atvieglotai izstrādei un darbināšanai;
- JBoss Developer Studio – izstrādes vide JBoss produktiem [14].

Mule ESB ir servisu serveris un izstrādes vide, kura bāzēta izstrādāta uz Java virtuālās mašīnas. Arī ar MuleESB ir iespējams veidot integrācijas servisus ar *vilkt un nomest* principu. Integrācijas iespējams veidot balstītas uz dažādiem ziņu formātiem, piemēram, SOAP un bināras ziņas. Arī Mule integrācijas platformā, līdzīgi kā iepriekš apskatītajām platformām, pieejami gatavi savienojumi ar populārākajām integrējamajām sistēmām, datubāzēm, ziņojumapmaiņas serveriem u.c. [15].

2.1. Spring Integration ietvars

Java izstrādātāju vidū Spring, precīzāk Spring-Framework, ietvars pazīstams jau no 2002. gada. Spring-Framework jeb kodols tiek lietots ļoti plaši ietvara *atkarību injekcijas* (no angļu val. dependency injection) dēļ, ietvaru izstrādā ASV bāzēts uzņēmums Pivotal. Atkarību injekcija izpaužas kā klašu instanču savstarpēju saikņu veidošana vietās, kur tas nepieciešams, programmas darbības laikā, piemēram, ja eksistē klases A instance un klases B instance, kurai kā viens no laukiem ir ar tipu klase A, tad ar atkarību injekcijas palīdzību ietvars nodrošina to, ka klases B instancei ir pieejama klases A instance, bez vajadzības speciāli B klasē veidot jaunu objektu, šīs savstarpējās saiknes tiek veidotas automātiski. Spring ietvarā atkarību injekcija ir veicama ar vienu no diviem veidiem, atkarībā, kā konkrētais projekts ir konfigurēts, XML konfigurācija vai Java anotācijas, kuras ir ieviestas salīdzinoši nesen kā daudz kompaktāks un ērtāks konfigurācijas līdzeklis, pēc ietvara kritikas par lielo XML failu daudzumu nepieciešamību projekta konfigurēšanai, kuri lielos projektos kļūst grūti pārvaldāmi un lasāmi [17].

Spring kodols tiek plaši lietots gan trešās puses bibliotēkās, piemēram, Apache Camel ietvarā, gan citās Spring bibliotēkās. Uz Spring kodola bāzes laika gaitā ir izveidoti daudzas Spring bibliotēkas, kuras visas ir ērti savā starpā integrējamas. Populārākās Spring bibliotēkas ir:

- Spring MVC – ietvars mājaslapu, REST servisu izstrādei;

- Spring Batch – ietvars pakešu procesu (no angļu val. batch process) izstrādei;
- Spring Security – tīmekļa lapu drošībai paredzēta bibliotēka, tai skaitā autentifikāciju un autorizāciju atvieglotai izstrādei;
- Spring Web Services – SOAP protokola tīmekļu servisu izstrādei paredzēta bibliotēka;
- Spring Cloud – bibliotēka ar rīkiem dalītu sistēmu izstrādei (no angļu val. distributed systems).

Spring Integration, līdzīgi kā pārējās Spring bibliotēkas, izstrādātas uz Spring kodola bāzes, pirmo reizi bibliotēka tika izlaista 2008. gadā. Tradicionāli visas Spring bibliotēkas ir atvērtā koda, bezmaksas lietošanai. Ar ietvaru iespējams veidot ziņu apstrādes plūsmas, lietojot vai nu XML failus vai Java valodā rakstītas plūsmas, kuras tika ieviestas tikai 2015. gadā, līdz tam vienīgais veids bija XML faili [17]. Spring Integration bibliotēkā implementēta lielākā daļa integrācijas šablonu, kuri plūsmu veidošanā atbilst šablonu nosaukumiem, kas atvieglo integrāciju izstrādi šablonu pārzinātājiem. Ir pieejams plašs atbalsts integrāciju veidošanai, kur nepieciešama integrācija ar JMS ziņojumapmaiņas sistēmām, FTP protokolu failu pārraidei, sistēmas failu lasīšanu un rakstīšanu, TCP un UDP protokoliem datu pārraidei, SOAP protokola tīmekļa servisiem un HTTP protokolu tīmekļa servisiem.

Kā Spring Integration sliktās puses tiek uzskatīta sarežģītā, grūti lasāmā, integrāciju plūsmu veidošana, salīdzinot ar Camel ietvaru, dažādu komponentu atbalsta trūkums, kāds ir pieejams Camel vai Mule ESB un XML nelasāmā konfigurācija, kura tiek lietota lielākajā daļā gadījumu, jo Java valodas plūsmu veidošana ir salīdzinoši jauna, vēl nepilnīga un tai trūkst labas dokumentācijas.

Spring Integration ietvaru galvenokārt iesaka lietot jau esošos uz Spring ietvaru bāzētos projektos, kuros nepieciešama salīdzinoši vienkārša integrācija ar kādu no minētajām tehnoloģijām, kuru atbalsta ietvars, lai nebūtu vienkāršos gadījumos nepieciešamība dažādu ietvaru kopā jaukšanai [18].

2.2. Apache Camel ietvars

Apache Camel ir populārākais, biežāk lietotais integrācijas ietvars Java programmēšanas valodai, tas ir bezmaksas, atvērtā pirmkoda projekts, kuru izstrādā ASV bāzēts uzņēmums

Apache. Ietvars pirmo reizi tika izlaists 2007. Gadā, un tas tiek nepārtraukti attīstīts un papildināts ar jaunām iespējām [19].

Apache Camel integrācijas ietvars, kā lielākā daļa Java projekti mūsdienās, parasti tiek lietots kopā ar Spring ietvaru, Spring atkarību injekcijas funkcionalitātes dēļ. Lietojot Camel ietvaru, integrācijas iespējams veidot kādā no vairākiem pieejamajiem veidiem:

- Java DSL – lietojot Java programmēšanas valodu;
- Annotation DSL – lietojot Java programmēšanas valodas anotācijas;
- Spring XML – definējot integrācijas Spring XML failos;
- Blueprint XML – lietojot OSGi Blueprint XML failus (neliela atšķirība, salīdzinot ar Spring XML);
- Groovy DSL – lietojot Groovy programmēšanas valodu;
- Scala DSL – lietojot Scala programmēšanas valodu;
- Kotlin DSL – lietojot Kotlin programmēšanas valodu (darba rakstīšanas brīdī Kotlin DSL ir izstrādes stadijā) [20].

Biežāk lietotie integrāciju rakstīšanas veidi ir Java DSL un Spring XML, taču Groovy un Scala programmēšanas valodu popularitāte starp valodām, kuras darbojas uz Java virtuālās mašīnas, nepārtraukti pieaug, kas, domājams, novedīs pie Groovy DSL un Scala DSL plašākas lietošanas.

Arī Apache Camel ietvars veidots, balstoties uz integrācijas šabloniem, no kuriem lielākā daļa ir tieši implementēta ietvarā, ber ir arī tādi šabloni, kuri nav tieši implementēti, bet ietvara izstrādātāji ir ar vienkāršiem koda piemēriem parādījuši, kā panākt ekvivalentu funkcionalitāti ar Camel pieejamajiem komponentiem, piemēram, selektīva ziņu patērētāja (skatīt 1.5.6. Selektīvs ziņu patērētājs) funkcionalitāte ietvarā ir izveidojama ar ziņu filtrēšanu uzreiz pēc saņemšanas [21].

Camel ietvars balstās uz URI lietošanu, ar ko iespējams novirzīt jebkuru ziņu uz tai nepieciešamo apstrādes kanālu vai ārēju sistēmu, tā kā Camel ir ļoti plašs atbalstīto sistēmu skaits, šāda URI lietošana ir ļoti ērta, jo vajadzības gadījumā var aizstāt ziņas sūtīšanu uz datubāzi vai ziņojumapmaiņas sistēmu, tikai samainot vienu koda rindiņu. Apache Camel atbalsta tādus protokolus un ārējas sistēmas kā HTTP protokols, JMS dažādas implementācijas, tai skaitā ActiveMQ, ZeroMQ, IBM MQ, Rabbit MQ u.c., SSH protokols, SFTP un FTP protokoli, RMI protokols, JPA datubāzu API, Quartz plānošana, IRC un XMPP protokoli

tērzēšanas sistēmām, RSS tīmekļa plūsma, LDAP autentifikācijas sistēma, IMAP, IMAPS, POP3, POP3S, SMTP, SMPTS e-pasta protokoli un daudzas citas sistēmas un protokoli.

Tieši lielais protokolu un ārējo sistēmu skaits, kurš tiek atbalstīts Camel ietvarā, ir galvenie ietvara plusi. Otra lielā pozitīvā lieta gan pret Spring Integration ietvaru, gan pret citām integrāciju izstrādes sistēmām ir vienkāršā ziņu apstrādes plūsmu veidošana gan XML formātā, gan Java formātā.

3. JAVA VIRTUĀLĀS MAŠĪNAS OPTIMIZĀCIJAS IESPĒJAS

Java virtuālā mašīna jeb JVM ir programma, kura interpretē kompilētu Java bināru kodu jeb baitkodu par datora procesoram saprotamām instrukcijām un darbina to. Tā iemesla dēļ, ka Java programmu kods netiek kompilēts tieši uz mašīnkodu, bet gan uz java baitkodu, kompilētas Java programmas ir iespējams darbināt uz dažādiem datoriem neatkarīgi no operētājsistēmas, ja uz konkrētā datora ir uzstādīta atbilstošā Java virtuālā mašīna.

Lai nodrošinātu to, ka Java baitkods strādā identiski neatkarīgi no konkrētā datora vai operētājsistēmas, katrai Java virtuālajai mašīnas implementācijai ir jāatbilst konkrētās versijas Java virtuālās mašīnas specifikācijai, kuru izdod Java preču zīmes turētājs – ASV bāzēta kompānija *Oracle Cooperation* [22].

Pašreiz aktuālajai Java 8 versijai tirgū ir pieejamas vairākas Java virtuālās mašīnas implementācijas (skatīt tabulu 3.1.).

3.1. tabula

Java virtuālās mašīnas

Nosaukums	**Ražotājs**	**Komentāri**
HotSpot	Oracle	Java SE sastāvdaļa. JIT kompilācija. Pieejams plašs klāsts optimizācijas karodziņu. Atbalsta Windows, Unix, Mac OS X. Sastāda > 99% no tirgus daļas [23].
IBM J9	IBM	Atbalsta UNIX, Windows, AIX, z/OS, IBM i operētājsistēmas, bāzēts uz Oracle HotSpot JVM. Galvenokārt JVM tiek lietota IBM produktos. [24]
Azul Zing JVM	Azul Systems	Maksas programmatūra. Atbalsta UNIX sistēmas, paredzēta lielu uzņēmumu sistēmu darbināšanai ar augstā veiktspējas prasībām. Bāzēts uz Oracle HotSpot JVM [25].

3.1. HotSpot virtuālās mašīnas optimizācijas parametri

HotSpot virtuālajā mašīnā kopš Java versijas 1.3 ir JIT (*Just In* Time) kompilators, kurš monitorē programmas izpildi, analizējot tās programmas daļas, kuras tiek izpildītas visbiežāk vai kuras patērē visvairāk resursus. Ja JIT kompilators automātiski secina, ka kāda metode tiek izpildīta pastiprināti, tad tā virtuālās mašīnas fona pavedienā tā tiek kompilēta uz mašīnkodu,

izmantojot iekšējās kompilatora optimizācijas, un ierakstīta virtuālās mašīnas kešatmiņā, lai pie nākamās metodes izsaukšanas reizes būtu jau pieejams vajadzīgais mašīnkods.[26]

Tā iemesla dēļ, ka Oracle HotSpot Java virtuālā mašīna sastāda vairāk kā 99% no kopējās tirgus daļas un tā ir iekļauta standarta Java versijā, tālāk tiks apskatītas tieši HotSpot virtuālās mašīnas optimizācijas iespējas.

HotSpot virtuālo mašīnu iespējams optimizēt Java programmas palaišanas brīdī, komandrindā norādot vajadzīgos parametrus. Ja parametri netiek norādīti virtuālā mašīna lieto noklusētās vērtības. Norādot specifiskus parametrus ir iespējams iespējot vai atspējot kādu no virtuālās mašīnas funkcijām vai norādīt parametrus funkciju darbināšanai. Turpmākajā nodaļas daļā aprakstīti HotSpot virtuālās mašīnas biežāk lietotie parametri, ar kuru palīdzību iespējams optimizēt virtuālās mašīnas darbu priekš konkrētas Java programmas.

-client un -server

64 bitu Java sistēmām noklusētā un vienīgā iespējamā vērtība ir -server, 32 bitu operētājsistēmām noklusētā vērtība ir -client, taču ir iespējams uzstādīt arī -server. Vienlaicīgi var būt iespējots tikai viens parametrs -client vai -server [27].

Atkarībā no parametra tiek darbināta viena no divām HotSpot virtuālajām mašīnām (klienta vai servera). Klienta virtuālā mašīna ir konfigurēta, lai tās startēšana būtu pēc iespējas ātrāka, taču servera virtuālā mašīna ir konfigurēta ilgstošai darbībai, lai būtu pēc iespējas augstāka veiktspēja.

-Xcomp un -XX:CompileThreshold

Pēc noklusējuma HotSpot virtuālā mašīna metodes izpilda, interpretējot baitkodu par mašīnkodu programmas darbošanās brīdī, ja virtuālā mašīna konstatē, ka kāda no metodēm tiek pastiprināti lietota (1000 reižu, ja virtuālā mašīna darbojas kā -client, 10000 reižu, ja kā -server), tad metode tiek kompilēta uz mašīnkodu, un šis kods ierakstīts koda kešatmiņā, tādējādi pie nākamās metodes izsaukšanas reizes kompilēšana uz mašīnkodu vairs nav nepieciešama.

Ar -Xcomp parametru virtuālā mašīna tiek piespiesta kompilēt katru metodi uz mašīnkodu, tādējādi var paātrināt programmas darbību metožu atkārtotu izsaukumu gadījumā.

Ar -XX:compileThreshold var norādīt patvaļīgu metodes izsaukumu skaitu, pēc kura kompilēt to uz mašīnkodu, piemēram, -XX:compileThreshold=1500.

-Xmn, -Xms un -Xmx

Ar šiem parametriem iespējams uzstādīt sākuma un maksimālo jaunās paaudzes kaudzes izmēru baitos (-Xmnsize), sākotnējo (-Xmssize) un maksimālo (-Xmxsize) kaudzes izmēru baitos. Norādot atbilstošu izmēru kaudzei, var izvairīties no dinamiskas kaudzes izmēra mainīšanas, tiek atvieglots virtuālās mašīnas darbs, jo tai nav periodiski dinamiski jārēķina nepieciešamais izmērs un jāmaina kaudzes izmērs. -Xmnsize vietā iespējams lietot arī -XX:NewSize un -XX:MaxNewSize, lai uzstādītu dažādo sākotnējo un maksimālo jaunās paaudzes objektu izmēru kaudzē.

Norādot šos lielumus, virtuālajai mašīnai tiek nodota informācija, cik lielu atmiņas daudzumu var atvēlēt dinamisku Java objektu uzglabāšanai, praksē bieži maksimālie un sākotnējie lielumi tiek norādīti vienādi, lai teorētiski novērstu atmiņas datu reorganizāciju, kaudzes izmēram mainoties, taču šāda pieeja var nebūt optimāla visos gadījumos [28].

-XX:MaxMetaspaceSize

Līdzīgi kā iepriekš apskatītie parametri dinamisku objektu atmiņas vietas uzstādīšanai ar -XX:MaxMetaspaceSize var norādīt, cik liela atmiņa jāatvēlē klašu metadatiem. Pēc noklusējuma atmiņas izmērs ir neierobežots, tāpēc sistēmas stabilitātes dēļ ir ieteicams norādīt šo lielumu.

- Xnoclassgc

Atspējo *atkritumu* savākšanu klašu metadatiem, ar šo parametru iespējams ietaupīt *atkritumu* savākšanai patērēto laiku, taču rezultātā programma var patērēt lielāku atmiņas daudzumu.

-Xsssize

Ar parametru iespējams uzstādīt atmiņas daudzumu baitos, kāds tiks atvēlēts katram Java pavedienam. Noklusētā vērtība ir atkarīga no sistēmas RAM lieluma.

-XX:+AggressiveOpts

Iespējo agresīvās veiktspējas optimizācijas funkcijas. Tiek sagaidīts, ka šis parametrs būs iespējots pēc noklusējuma, sākot ar Java 9 HotSpot virtuālās mašīnas izlaišanu, Java 8 versijā pēc noklusējuma ir atspējots.

-XX:+AggressiveHeap

Iespējo Java dinamisko objektu kaudzes optimizāciju. Norādot šo parametru, startējot Java virtuālo mašīnu, tiks iespējoti vai uzstādīti vajadzīgie parametri priekš optimālas ilgu Java

procesu izpildes. Vajadzīgie parametru lielumi tiks automātiski aprēķināti pēc sistēmas RAM un CPU informācijas.

3.2. Java 9 izpildlaika vides optimizācija

Java platformai attīstoties, Java sistēmas koda daudzums un izmērs ar katru jauno versiju ir pieaudzis, taču pirms Java 8 versijas iespējas instalēt tikai daļu no JRE nebija. Java 8 versijā tika ieviesti kompaktie profili [29]. Kompaktie profili ir pieejami Java SE Embedded versijā, kas ir paredzēta ierīcēm ar ierobežotiem resursiem. Tie dod iespēju instalēt pilnu JRE vai kādu no trijiem kompaktajiem profiliem, lai gan šāda pieeja, bez šaubām, ir uzlabojums, taču šī pieeja tikai uzlaboja esošo situāciju, nevis pilnībā atrisināja, jo kompaktie profili ir iepriekš definēti un tos nav iespējams veidot pēc nepieciešamības [30]. Kompakto profilu iekļautās pakotnes un izmērus skatīt tabulā 3.2.. Standarta Java 8 klientu sistēma aizņem ~163MB, pilna Java SE Embedded verija aizņem ~100MB.

3.2. tabula

JRE kompaktie profili [29]

Compact1 (~11mb)	**Compact2 (~15mb)**	**Compact3 (~21mb)**
java.lang.* (bez .instrument un .management)	Compact1 profils	Compact2 profils
java.io(.)*	java.rmi(.)*	java.lang.instrument
java.math	java.sql	java.lang.management
java.net	javax.sql (bez .rowset.*)	java.security.acl
java.nio(.)*	java.x(.)*	java.util.prefs
java.security(.)*	java.xml(.)* (bez .crypto)	javax.lang.*
java.text(.)*	org.w3c.dom(.)*	javax.management.*
java.time(.)*	org.xml.sax(.)*	javax.naming(.)*
java.util(.)* (bez .prefs)		javax.auth.kerberos
javax.crypto(.)*		javax.sasl
javax.net(.)*		javax.rowset(.)*
javax.script		javax.tools
javax.security.* (bez sasl .auth.kerberos, .sasl)		javax.xml.crypto(.)*,org.ietf.jgss

Ja programmai nav nepieciešamas klases no noteiktām Java platformas klasēm, tad iespējams lietot atbilstošo kompakto profilu, kā rezultāta iespējams uzlabot programmas ātrdarbību un startēšanas laiku, jo nepieciešams izmantot mazāk atmiņas.

Patvaļīga JRE sistēmas veidošana sākotnēji bija paredzēta kā Java 8 uzlabojums, taču tas tika pārcelts uz Java 9 versiju, kam par iemeslu bija Jigsaw projekta izstrādes aizkavēšanās. Jigsaw projekts pārveido Java platformas arhitektūru no monolītiskas uz modulāru, un, pateicoties skaidri definētajiem moduļiem, no kuriem sastāvēs Java platforma, izstrādātāji varēs veidot JRE ar tikai tiem moduļiem, kuri nepieciešami konkrētās programmas darbināšanai, kas gan paātrinās Java programmu darbību, gan Java platforma būs ērti instalējama arī uz ierīcēm ar ierobežotiem resursiem, piemēram, rūteriem, televizoriem.

4. JAVA PROGRAMMU OPTIMIZĀCIJA

Lai programma nepatērētu nevajadzīgus gan atmiņas, gan procesora resursus, ir svarīgi, lai rakstītais kods neveidotu liekas datu struktūras, neveiktu liekas darbības, bez kurām programma var iztikt, lai sasniegtu nepieciešamo rezultātu – šo rezultātu iespējams sasniegt, rakstot kvalitatīvu kodu, izvēloties pareizākās datu struktūras, kā reprezentēt programmai nepieciešamos datus, un lietojot katram gadījumam atbilstošākās programmēšanas valodas struktūras.

Nodaļā tiek apskatītas Java programmu veidošanas tipiskākās kļūdas no koda efektivitātes viedokļa, autors demonstrē sekas, kādas neefektīvs kods, nepareizu klašu lietošana, dažādu bibliotēku lietošana un nepareizu arhitektūras lēmumu pieņemšana var atstāt uz programmas darbības ātrumu, gan veicot praktiskus eksperimentus, gan atsaucoties uz citu autoru veiktiem pētījumiem.

4.1. Java koda optimizācija

Java programmēšanas valodā nav jāuztraucas par vairs nevajadzīgu objektu atmiņas atbrīvošanu pēc tam, kad objekts vairs nav vajadzīgs, kā tas ir C vai zemāka līmeņa programmēšanas valodās. Kad objekts vairs nav vajadzīgs atkritumu savācējs atbrīvo operatīvo atmiņu automātiski, taču tas nenozīmē, ka izstrādātājiem nav jādomā par objektu veidošanas, metožu izsaukšanas, konstrukciju lietošanas ietekmi uz atmiņu un procesora slodzi.

4.1.1. Objektu veidošana

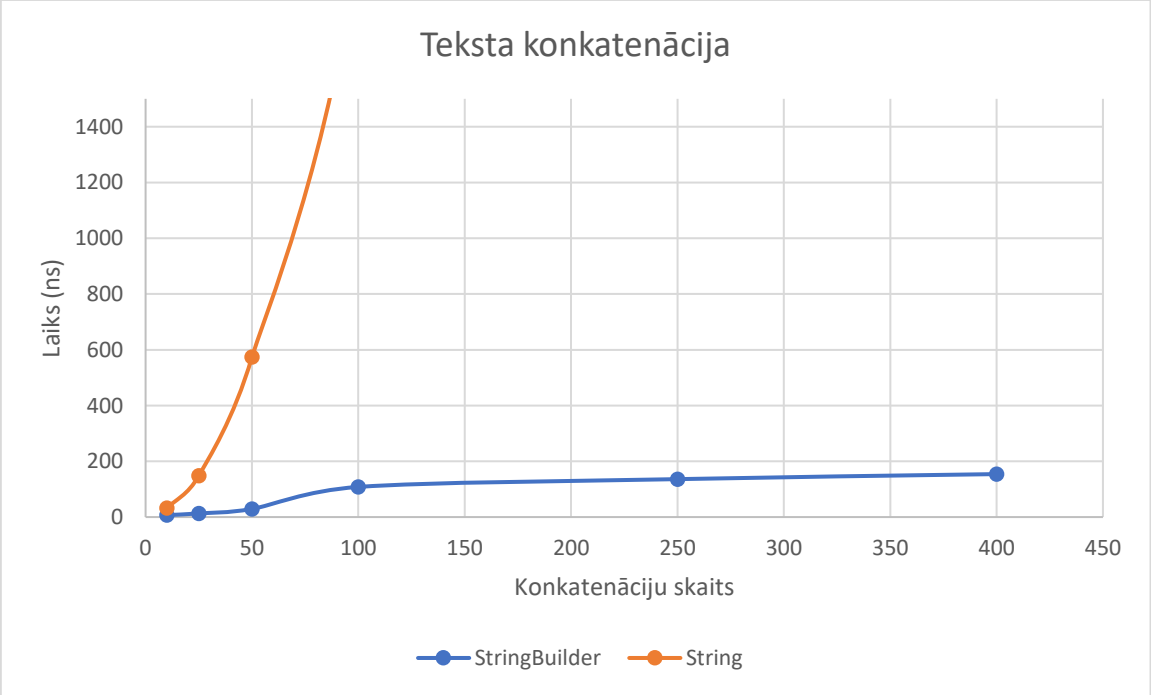
Tā kā Java programmēšanas valoda ir objekt-orientēta valoda, tad jebkura datu reprezentācija atmiņā ir objekts. Ja kodā tiek veidoti nevajadzīgi objekti, tad tas nevajadzīgi patērē procesora resursus: vispirms objektu veidojot, pēc tam, darbinot atkritumu savācēju, kurš veic virtuālās mašīnas steka reorganizāciju, ļaujot bijušā objekta atmiņu, lietot jaunu objektu izveidei.

Lai minimizētu objektu veidošanas iespaidu uz programmas veiktspēju, būtu jāizvairās no jaunu objektu veidošanas bieži lietotās metodēs. Tā vietā, lai veidotu jaunu objektu, vairumā gadījumu iespējams izmantot objektu, kurš padots metodei kā parametrs, vai citos gadījumos metodē iespējams atkārtoti izmantot vienu un to pašu izveidoto objektu jauna veidošanas vietā.

Teksts, viens no biežāk lietotajiem objektu tipiem, Java valodā tiek glabāts String klases objektos, ar kuriem darbojoties nereti tiek ievērojami pasliktināta programmas veiktspēja. Tā kā

String klasē datu reprezentācijā tiek lietots burtu masīvs, String objekti ir nemainīgi, t.i., lai pieliktu kaut vienu burtu klāt esošam objektam, viss burtu masīvs tiek kopēts uz jaunu masīvu, kura garums ir par vienu burta vietu lielāks un kuram tiek pievienots jauns burts. Ja cikliski tiek veikta teksta konkatenācija, tad katrā cikla izpildes reizē tiek veidots jauns masīvs un tiek veikta masīva kopēšana, kas prasa salīdzinoši daudz resursu, tāpēc šādām darbībām ar tekstu veiktspējas apsvērumu dēļ jāizmanto StringBuilder klase, kura, lai gan arī izmanto masīvu, tās veiktspēja ir daudz augstāka. StringBuilder instancēs masīvu kopēšana notiek daudz retākos gadījumos – masīvs tiek uzturēts lielāks kā nepieciešams, un kopēšana tiek veikta tikai tad, kad StringBuilder instances limits ir sasniegts, tad jaunais masīvs tiek veidots divas reizes lielāks nekā esošais.

String un StringBuilder klases veiktspēju darba autors ar autora rakstītu testa Java kodu, kurš atrodams GitHub repozitorijā [48], testēja eksperimentāli, izmērot laiku mikrosekundēs, kāds tiek patērēts objektu konkatenācijām ar katru no klasēm. Kā redzams attēlā 4.1., jau pie 25 tekstu konkatenācijām ātrdarbības starpība starp String un StringBuilder klasēm ir vairākas reizes, pie 400 tekstu konkatenācijām patērētā laika starpība pārsniedz 238 reizes.



Attēls 4.1. String un StringBuilder klašu veiktspēja

Līdzīgi kā teksta objekti, arī kolekcijas Java valodā tiek glabātas masīvos, tāpēc, lai gan kolekcijas tiek uztvertas kā dinamiskas, t.i., tās izmērs pats tiek mainīts automātiski pēc

vajadzības, tomēr arī kolekcijās tiek veikta masīvu kopēšana, lai panāktu šo dinamiskumu, kas prasa papildus resursus. Tāpēc, ja iespējams, programmās kolekcijas būtu jāveido, iepriekš mēģinot paredzēt to izmēru. Nedaudz lielākas kolekcijas izveidei nekā nepieciešams parasti, tomēr ir izdevīgāka kā dinamiskas kolekcijas izveide, protams, ne vienmēr ir iespējams paredzēt kolekcijas izmēru.

Bieži lietotas kolekcijas Java valodā ir saraksti – klases, kuras implementē interfeisu *java.util.List*. Populārākās šī interfeisa implementācijas ir LinkedList, ArrayList, Stack, Vector un CopyOnWriteArrayList. Kā redzams tabulā 4.1., katrai saraksta implementācijai patērētās milisekundes dažādu metožu izpildei atšķiras (laiks mērīts 100 reizes,veicot 10000 operācijas uz kolekcijas objektu)[31]. Java dokumentācija rekomendē lielākajā daļā gadījumu lietot ArrayList implementāciju, kas arī redzams tabulas ātrdarbības datos, taču, ja programmā nepieciešama bieža elementu dzēšana un pievienošana saraksta sākumam, tad lietot LinkedList implementāciju, ja nepieciešama sinhronizācija starp vairākiem Java pavedieniem, tad lietot Vector implementāciju[32].

4.1. tabula

Sarakstu ātrdarbība [31]

Saraksts / Metode (ms)	**add()**	**get()**	**iterate()**	**size()**
Vector	12.691	0.143	0.286	0.047
Vector (inicializēts ar izmēru)	10.134	0.045	0.042	0.009
ArrayList	9.873	0.051	0.037	0.013
ArrayList (inicializēts ar izmēru)	9.845	0.036	0.003	0.005
LinkedList	9.913	172.824	0.538	0.030
Stack	9.843	0.105	0.129	0.060
CopyOnWriteArrayList	36.909	0.092	0.099	0.051

Otrs biežāk lietotais kolekciju tips Java valodā ir mape – klases, kuras implementē interfeisu *java.util.Map*. Mape ir līdzīga sarakstam, tikai katrs mapes objekts ir atslēgas-vērtības pāris. Gan atslēga, gan vērtība var būt patvaļīgs objekts (vienā mapē atslēgām jābūt ar vienādu tipu, tas pats attiecas arī uz vērtībām). Vispārējos gadījumos Oracle rekomendē lietot kādu no lietotākajām implementācijām – HashMap, TreeMap vai LinkedHashMap. Ja nepieciešams, lai mape tiktu uzturēta sakārtotā secībā, tad jālieto TreeMap, ja par secību svarīgāka ir ātrdarbība,

tad jālieto HashMap, ja nepieciešama, lai objekti ar laiku tiktu kārtoti pēc to lietošanas biežuma, kas var uzlabot veiktspēju, tad jālieto LinkedHashMap [33]. Ja tiek plānots, kā atslēgu glabāt enumerācijas, tad viseffektīvāk lietot EnumMap, taču ar WeakHashMap iespējams veidot reģistriem līdzīgas struktūras, atslēgu vietā tiek glabātas vājās references uz pašu atslēgas objektu. Brīdī, kad neviens no programmas pavedieniem nevar piekļūt atslēgas objektam, atkritumu savācējs atbrīvos gan atslēgas objekta, gan mapes ieraksta atmiņu.

Paātrināt programmas darbību iespējams arī, izvērtējot , kurā brīdī visizdevīgāk veidot jaunos objektus. Ja programmai ir nepieciešams maksimāli ātrs startēšanas laiks, tad iespējams lietot aizkavēto objektu inicializāciju, t.i., inicializēt objektu tikai tad, kad tas ir nepieciešams. Piemēram, ja objektam no klases A ir mainīgais cits objekts no klases B, tad A klases konstruktorā neinicializēt objektu B – atstāt to ar vērtību *null*, un veikt inicializēšanu tikai pie pirmās nepieciešamības, veicot pārbaudi ar objekta vienādību pret *null*. Šo konceptu var gan ļoti viegli implementēt, gan to automātiski nodrošina dažādi Java ietvari, piemēram, Spring ietvarā iespējams norādīt anotāciju *@Lazy*.

Pretējais koncepts aizkavētajai objektu inicializācijai ir ātrā objektu inicializācija – visi zināmie programmā nepieciešamie objekti tiek inicializēti programmas startēšanas brīdī. Ar šo konceptu programmas startēšana var kļūt nedaudz lēnāka, taču tiek ietaupīts laiks programmas darbināšanas brīdī. Šis koncepts Spring ietvarā tiek lietots kā noklusētais variants.

4.1.2. Izņēmumu gadījumu apstrāde

Ja programmā notiek izņēmum-notikums, tas patērē salīdzinoši daudz resursus, jo tiek kopēts virtuālās mašīnas steka saturs, lai varētu izdrukāt precīzu steka trasējumu. Tāpēc izņēmum-notikumu veidošanai metodēs nevajadzētu būt daļai no normālas programmas plūsmas, kas arī atbilst labai programmēšanas praksei, jo tas var atstāt iespaidu uz programmas veiktspēju. Izņēmum-notikumiem būtu jātiek veidotiem tikai tad, kad programmā tiešām noticis izņēmums, kuram nevajadzētu notikt, ja programma darbojas korekti.

Izņēmum-notikuma veidošana ar virtuālās mašīnas steka kopēšanu patērē aptuveni 2000 reižu vairāk laika nekā izņēmum-notikuma veidošana nekopējot steka saturu [34]. Lai gan steka satura kopēšanai izņēmumos ir liela nozīme, lai identificētu to, kurā vietā un kāda iemesla dēļ izņēmums ir noticis, ja ir iespējams, no steka kopēšanas ir jāizvairās, to var panākt divējādi: pārrakstot izņēmuma klases *fillInStackTrace* metodi vai veidojot atkārtoti, izmantojot izņēmuma objektu, kurš izveidots, lietojot konstruktoru, kurš neizsauc *fillInStackTrace* metodi.

4.1.3. Cikli un zarošanās izteiksmes

Cikli ir ļoti plaši lietota konstrukcija jebkurā programmēšanas valodā, kur šāda konstrukcija ir pieejama. Tā iemesla dēļ, ka cikla ķermenis tiek izpildīts vairākas reizes, ir jāpievērš uzmanība tam, kādas darbības tiek veiktas cikla ķermeņa kodā. Visam kodam – vērtību piešķiršanai, datu piekļuvei, citu metožu izsaukšanai, kurš nav jāizpilda katrā iterācijas reizē, noteikti jābūt iznestam pirms cikla.

Ja ciklā tiek apstrādāti masīva dati, tad vienmēr izdevīgāk ir masīva elementa datus ierakstīt pagaidu mainīgajā, nekā vairākas reizes piekļūt viena un tā paša indeksa elementam, jo virtuālā mašīna vienmēr pārbauda, vai netiek pārkāptas masīva robežas, pirms atgriež nepieciešamo vērtību.

Ciklu definīcijā jāizvairās no metodes izsaukumu - cikla definīcijā, kā tas bieži tiek darīts iterācijās pār kolekcijām, jo metode tiek izsaukta pirms katras iterācijas un metodes izsaukums patērē vairāk resursus nekā testa veikšana ar lokālu mainīgo.

Cikla indeksam, ja tāds tiek lietots, jālieto primitīvs *int* tips, jebkura cita cipara tipa (*double, float, char, Integer, Float, Double utt.)* vietā, ja tas ir iespējams.

Ja programmā jāveic masīva kopēšana uz citu masīvu, tad jāizmanto *System* klases *arraycopy* metode, kura ir vismaz 2x ātrāka par kopēšanu ciklā gan lielu, gan mazu masīvu gadījumos [35].

Ja iespējams izdevīgāk programmā lietot *switch* konstrukciju vairāku *if - else if* konstrukciju vietā. Pie maza skaita pārbaudēm veiktspēja ir līdzīga, taču, pārbaužu skaitam palielinoties, *switch* konstrukcija tiek izpildīta ātrāk. Ātrdarbības atšķirības novērojamas tā iemesla dēļ, ka Java baitkodā ir speciālas konstrukcijas *switch* izpildei. Switch salīdzinājuma kreisā puse atmiņā tiek ielasīta tikai vienu reizi, taču *if-else* konstrukcijā katrā pārbaudes reizē visas vērtības tiek ielasītas atmiņā par jaunu.

4.2. Integrācijas komponentu eksperimentāla veiktspējas noteikšana

Nodaļā tiek apskatītas konkrētas biežāk lietoto integrācijas komponentu, plaši lietotu bibliotēku un ar integrāciju arhitektūru saistītu lēmumu sekas uz integrāciju veiktspēju, autoram veicot praktiskus eksperimentus vai atsaucoties uz citu autoru veiktiem eksperimentiem. Autora veikto eksperimentu kods atrodams GitHub repozitorijā [48].

4.2.1. Teksta datu konvertācija

Integrācijās, kurās tiek integrētas vairākas sistēmas, dati starp šīm sistēmām, parasti, tiek pārsūtīti, lietojot vienu no diviem formātiem: XML vai JSON. Zinot sagaidāmo datu saturu, teksta datus var pārveidot par Java klašu instancēm un otrādi, šo procesu sauc par serealizāciju, lai šo procesu veiktu ir pieejamas vairākas Java trešo pušu bibliotēkas un Java standarta versijā iekļautais API - JAXB. Populārākie veidi, kā pārveidot XML datus par Java objektiem, ir Java sistēmā pieejamais API – JAXB un XStream bibliotēka, savukārt populārākie veidi, kā pārveidot JSON datus, ir atvērtā koda bibliotēka Jackson un Google bibliotēka GSon.

Java valodā nav pieejams standarta API, kā pārveidot JSON datus, šāda funkcionalitāte ir viens no potenciālajiem Java 10 versijas jauninājumiem, sākotnēji tika paredzēts, ka šis jauninājums būs pieejams Java 9 versijā, taču šī jauninājuma ieviešana tika atlikta. [36]

Autors, eksperimentāli izpildot autora izstrādātu Java kodu, kurš 100 reize sarealizē liela apjoma XML un JSON datnes (~ 100 MB lielu XML datni un ~ 58 MB lielu JSON datni ar ekvivalentiem datiem), mērot sistēmas laiku milisekundēs pirms un pēc katras veiktās sarealizācijas, nonāca pie secinājuma, ka JSON dati vidēji tiek pārveidoti 2-3 reizes ātrāk kā XML dati. Precīzi autora iegūtie sarealizācijas veiktspējas dati apskatāmi tabulā 4.2.. Autora izstrādātais Java testa kods ir atrodams GitHub repozitorijā [48].

4.2. tabula

Datu serealizēšanai patērētais laiks

Bibliotēka / Sarealizēšanas rezultāts	**XML (ms)**	**JSON (ms)**
JAXB	2255.32	X
XStream	14799.45	X
Jackson	X	769.47
Gson	X	972.36

Galvenais iemesls, kāpēc XML dati tiek pārveidoti ievērojami lēnāk kā JSON dati, ir tas, ka XML formātā aprakstīti dati aizņem vairāk vietas uz diska kā JSON dati. Jackson bibliotēka ir ātrākā JSON datu pārveidošanā, savukārt ātrākais veids, kā pārveidot XML datus, ir, lietojot Java sistēmas standarta API - JAXB.

4.2.2. Ziņu maršrutēšana

Svarīga integrācijas ietvaru funkcionalitāte ir ziņu maršrutēšana atkarībā no ziņas satura. Maršrutēšanu iespējams veikt, lietojot vienu no vairākām iespējām – XPath, XQuery valodām (XML formāta ziņām), galvenes saturu kādā no atslēgas – vērtības pāriem, objekta lauka vērtību. Parasti maršrutēšanas veids tiek izvēlēts atkarībā no tā, kas maršrutēšanas brīdī ir ziņas ķermeņa objekts – teksts vai jau konvertēts objekts.

Autors, eksperimentāli izpildot autora izstrādātu Java kodu, kurš maršrutē 100 secīgas ziņas, lietojot kādu no iespējamajām maršrutēšanas tehnoloģijām, uzņemot sistēmas laiku milisekundēs pirms un pēc ziņu maršrutēšanas, noskaidroja, ka, lietojot Apache Camel ietvaru, visātrāko maršrutēšanu iespējams veikt, pārbaudot objekta lauka vērtības, taču šī maršrutēšana iespējama tikai tad, ja teksta ziņa jau ir konvertēta uz objektu. Precīzi autora mērījumos iegūtie dati apskatāmi tabulā 4.3.. Autora izstrādātais Java testa kods ir atrodams GitHub repozitorijā [48].

4.3. tabula

Ziņu maršrutēšanas eksperimentāli rezultāti

Maršrutēšanas veids	**Vidējais laiks (ms)**
Galvenes vērtība	0.67
XPath valoda	3.99
XQuery valoda	3.85
Objekta lauka vērtība	0.38

Otrs ātrākais veids, kā veikt maršrutēšanu, ir, lietojot galvenes vērtību, šāda maršrutēšana ir aptuveni divas reizes lēnāka par objekta lauka vērtību maršrutēšanu, taču arī šo maršrutēšanas veidu iespējams veikt tikai tad, kad galvenes lauka vērtība jau ir uzstādīta, šādu maršrutēšanu var ļoti efektīvi veikt, lietojot standarta galvenes vērtības, kuras tiek saņemtas no ziņojumapmaiņas serveriem. XPath un XQuery maršrutēšana ir aptuveni 10 reizes lēnāka par objekta lauka vērtību maršrutēšanu, taču XPath un XQuery maršrutēšana veicama pret XML teksta datiem, un tie, parasti, nepieprasa nekādas papildus konvertācijas.

4.2.3. Lielu ziņu sadalīšana

Nereti integrācijām jābūt spējīgam sadalīt XML, JSON, retāk CSV formāta objektu sarakstu neatkarīgos objektos, lai apstrādātu katru objektu individuāli. XML failu sadalīšanai pieejamas vairākas tehnoloģijas, jo XML formāts vēsturiski lietots biežāk tieši kā patstāvīgi objekti, turpretī JSON formāts vairāk tieši datu pārsūtīšanai, pirms tas tiek konvertēts uz programmēšanas valodu objektiem.

XML failu sadalīšanai pieejamas vairākas iespējas: XPath, VTD-XML valodas un XML marķieri, JSON failiem pieejama JSONPath valoda, kā arī iespējams vispirms konvertēt teksta datus uz programmēšanas valodas objektiem un tad sadalīt objektu sarakstu atsevišķos objektos (skatīt nodaļu 4.2.1.).

Autors eksperimentāli, darbinot autora izstrādātu Java kodu, kurš testē katru no populārākajiem sarakstu sadalīšanas veidiem 1000 reizes un nosakot vidējo laiku, nonāca pie secinājuma, ka visātrāk Java integrācijās iespējams sadalīt Java sarakstu (objekts, kurš implementē *java.util.List* klasi) ar objektiem, taču šo sadalīšanas metodi iespējams lietot tikai tad, ja sadalīšanas brīdī ir pieejams saraksts atmiņā ar objektiem, papildu pārveidošana no teksta datiem uz Java sarakstu netika ņemta vērā. Precīzi autora veiktā eksperimenta iegūtie rezultāti redzami tabulā 4.4..Autora izstrādātais Java testa kods ir atrodams GitHub repozitorijā [48].

4.4. tabula

Sarakstu sadalīšanas eksperimentāli rezultāti

Sadalīšanas veids	**Vidējais laiks (ms)**
XPath valoda	1.159
Saraksta elementi	0.285
XML marķieri	0.609
VTD-XML valoda	0.665
JSONPath valoda	0.431

XML failu sadalīšanu visātrāk iespējams veikt, lietojot VTD-XML un XML marķierus – abām tehnoloģijām uzrādot līdzīgus rezultātus, gandrīz 2x lēnāk darbojas XPath valoda. JSON failu sadalīšana ar ekvivalentiem datiem kā XML failā, lietojot JSONPath valodu, ir gandrīz 1.5 reizes ātrāka par ātrākajām XML sadalīšanas iespējām.

4.2.4. Datu bāzu lietošana integrācijās

Integrācijās datu bāzes tiek lietotas ļoti plaši gan datu rakstīšanai, gan lasīšanai. Nereti viena integrācijas ceļa laikā tiek veiktas vairākas darbības ar datu bāzi vai pat dažādām datubāzēm.

Ja integrācijā jāveic datu rakstīšana vai atjaunošana datubāzē ātrdarbību iespējams paātrināt, veicot rakstīšanu vai atjaunošanu grupās, t.i., neveikt, piemēram, rakstīšanu, pēc katras ziņas, bet sakrāt, piemēram, 100 ziņas, un tad vienā reizē ierakstīt vai atjaunot visus objektus vienā reizē. Ar šādu pieeju iespējams samazināt ārējās sistēmas – datubāzes izsaukumus. Protams, ne vienmēr šo pieeju ir iespējams īstenot, ja kādai no nākamajām ziņām var būt nepieciešami dati no ziņām, kuras tikai gaida kārtu, kad tiks ierakstītas datu bāzē, tad grupēšanu nevar lietot, citādāk integrācija nestrādās korekti.

Pētījumā, kurš tika veikts ar visām populārākajām SQL datubāzēm, tika noskaidrots, ka ievietošanas un atjaunošanas darbības, lietojot grupēšanu, ir efektīvākas, izņemot MySQL datubāzē. Citām datubāzēm, to skaitā PostgreSQL, Oracle Database, Apache Derby, Microsoft SQL Server, H2 u.c., grupēšana sniedz ievērojamus uzlabojumus ātrdarbībā, sākot no 7% uzlabojuma Apache Derby datbāzei, līdz pat 503% un 325% uzlabojums attiecīgi Oracle Database un PostgreSQL datbāzem [37].

4.2.5. Ziņu filtrēšana

Integrācijās, īpaši tādās, kuras darbojas ar lielu apjomu reāllaika datiem, tiek lietots daudz filtra šablonu, lai filtrētu visus nederīgos datus, gan tādus, kuriem trūkst informācijas, gan tādus, kuri neatbilst kādiem biznesa likumiem. Šādās sistēmās svarīgi ir apzināties to, cik lielu daļu no ziņām katrs filtrs atzīst par nederīgu. Filtri jāsakārto tādā secībā, lai tie filtri, kuri atzīst vislielāko daļu no ziņām par nederīgām, tiktu pārbaudīti pirmie, tādējādi izvairoties no lieku darbību veikšanas tālākajā ziņu apstrādes plūsmā. Atstātais iespaids uz filtru secību katrā gadījumā būs citādāks, atkarībā no lieko darbību ietekmes uz veiktspēju, dažos gadījumos tas var neatstāt manāmu iespaidu, taču citos gadījumos filtru secība var atstāt salīdzinoši lielu iespaidu uz integrācijas veiktspēju.

5. PROFILĒŠANAS UN VEIKTSPĒJAS MĒRĪŠANAS RĪKI

Profilēšanas un veiktspējas mērīšanas rīki nepieciešami, lai programmas darbības laikā varētu analizēt programmas darbību – cik daudz tiek patērēta atmiņa, cik ilgs laiks tiek patērēts, izsaucot metodes, procesora noslogotību, pavedienu darbību utt.. Lietojot profilēšanas rīkus, iespējams ērti atrast kritiskās programmas koda daļas vai trešās puses bibliotēkas, kuras var optimizēt vai aizstāt, lai iegūtu vēlamo programmas veiktspēju. Java profilēšanas rīkus var iedalīt trijās dažādās kategorijās:

- standarta profilēšanas rīki, kuri trasē katru programmas un virtuālās mašīnas darbību;
- uz instrumentācijas API bāzes veidoti rīki, kuri atstāj minimālu iespaidu uz programmas veiktspēju, taču ir ierobežoti arī lietās, ko var analizēt;
- reāllaika veiktspējas pārvaldības rīki produkcijas videi.

5.1. Standarta profilēšanas rīki

2016. gada aptaujā, kurā piedalījās 2040 Java izstrādātāji, testētāji un arhitekti, tika atklāts, ka gandrīz puse jeb 38% no aptaujātajiem lieto Java profilēšanas rīku – VisualVM, 16% JProfiler rīku un 15% Java Mission Control, un 35% aptaujāto atzina, ka nelieto nekādus profilēšanas rīkus, kas parāda to, ka programmas ātrdarbības testēšana nereti tiek ignorēta, iespējams, īso termiņu vai projekta plānošanas dēļ. [38]. Visi minētie rīki pieskaitāmi pie standarta profilēšanas rīkiem, ar kuru palīdzību var iegūt ļoti plašu informācijas klāstu par programmas un virtuālās mašīnas darbību.

5.1.1. VisualVM rīks

VisualVM rīks sākotnēji bijis NetBeans izstrādes vides iekšējais profilēšanas rīks, taču tā iemesla dēļ, ka NetBeans vide starp Java izstrādātājiem nav pārāk populāra (~10% no visiem izstrādātājiem), profilēšanas rīks izveidots tika kā neatkarīga programma[38]. VisualVM ir atvērtā pirmkoda projekts, un tā pēdējai versijai (darba rakstīšanas brīdī jaunākā versija ir 1.3.9) jau ir atbalsts Java 9, neskatoties uz to, ka Java 9 versija plānota izlaišanai tikai 2017. gada beigās.

Rīku iespējams lietot programmām, kuras tiek darbinātas gan lokāli, gan attālināti. Ar Visual VM rīku iespējams:

- attēlot gan lokālas, gan attālināti darbojošās Java programmas;
- pārvaldīt un analizēt programmas atmiņas patēriņu;
- pārvaldīt programmas pavedienus;
- pārvaldīt programmas veiktspēju;
- veidot un attēlot pavedienu dublējumus programmas darbošanās brīdī;
- veidot un attēlot atmiņas kaudzes dublējumus programmas darbošanās brīdī.

Viena no VisualVM rīka labākajām īpašībām ir tā, ka rīks veidots ar modulāru arhitektūru, kas atvieglo dažādu spraudņu izstrādi, ar kuriem papildināt rīku. Darba rakstīšanas brīdī rīkam ir pieejami 26 oficiāli spraudņi, to skaitā programmas startēšanas analizēšanas spraudnis, GlassFish servera padziļinātas analizēšanas spraudnis, JavaFX lietotņu analizēšanas spraudnis un daudzi citi [39].

5.1.2. JProfiler rīks

Līdzīgi kā VisualVM, arī ar JProfiler rīku var analizēt gan lokālas programmas, gan attālināti pieslēdzoties serverim, uz kura konkrētā programma strādā. JProfiler rīks ir pieejams gan kā neatkarīga programma, gan kā spraudnis visām populārākajām Java izstrādes vidēm – InteliJ IDEA, Eclipse, NetBeans, Oracle JDeveloper. Ar JProfiler rīku iespējams veikt visus tos pašus mērījumus, kurus iespējams veikt ar VisualVM, kā arī JProfiler ir pieejamas datubāžu vaicājumu profilēšanas funkcijas, ar kurām var sekot līdzi katra datubāzes pieprasījuma izpildei, visiem atvērtajiem datubāzes savienojumiem un tiek automātiski atrasti pieprasījumi, kuri izpildās lēnāk nekā tiem vajadzētu. JProfiler rīks ir komerciāls rīks, kuru ražo uzņēmums EJ-Technologies un par tā lietošanai ir jāpērk licence [40].

5.1.3. Java Mission Control rīks

Java Mission Control ir rīku komplekts, kurš ir iekļauts ar Java izstrādātāju versijā (JDK). Java Mission Control sastāv no diviem galvenajiem rīkiem – JXM Console un Java Flight Recorder, ir pieejami arī papildus spraudņi funkcionalitātes paplašināšanai, kurus var uzstādīt no rīka lietotāja interfeisa. Java Mission Control ir pieejama kā neatkarīga programma un arī kā Eclipse izstrādes vides spraudnis.

Java Mission Control lieto JMX, lai komunicētu ar strādājošu Java programmu, lietojot RMI transporta protokolu. Ar rīku, līdzīgi kā ar iepriekš aprakstītajiem rīkiem, ir iespējams analizēt programmas slodzi uz procesoru, izmantoto atmiņu, pārvaldīt Java atmiņas kaudzi, kā

arī ierakstīt programmas izpildi, lai vēlāk to analizētu. Java Mission Control rīki ir pieejami bezmaksas lietošanai uz izstrādes vidēm, taču, ja vajadzīga programmas novērošana produkcijas vidē, ir nepieciešams iegādāties licenci [41].

5.2. Instrumentācijas programmēšanas profilēšanas rīki

Šie profilēšanas rīki izmanto *java.lang.instrument* API, kurš iekļauts Java standarta versijā, lai programmas baitkodā ievietotu koda profilēšanai nepieciešamo kodu. Tā iemesla dēļ, ka tiek izmainīts Java baitkods, ar šiem rīkiem iespējams testēt jebkuru kodu – gan paša rakstītu, gan trešās puses bibliotēku.

5.2.1. XRebel rīks

XRebel rīks ir komerciāls rīks, kuru izstrādā uzņēmums ZeroTurnaround. Rīks pieejams gan kā neatkarīga programma, gan Eclipse izstrādes vides spraudnis. Rīks automātiski atbalsta populārākos Java serverus (Tomcat, JBoss, Jetty u.c.). Visi profilēšanas dati no rīka pieejami reāllaikā, kamēr programma darbojas. Ar rīku iespējams:

- mērīt patērēto laiku katram tīmekļa pieprasījumam;
- apskatīt katru veikto tīmekļa pieprasījumu un atbildi;
- iegūt statistiku par veiktajiem datubāzes vaicājumiem;
- iegūt statistiku par metožu izsaukumiem;
- iegūt detalizētu informāciju par izņēmum-notikumiem [42].

5.2.2. Prefix rīks

Prefix rīks ir bezmaksas rīks, kuru izstrādā uzņēmums Stackify, tas ir pieejams gan Java, gan .NET programmēšanas valodām. Līdzīgi kā ar XRebel rīku, arī ar Prefix rīku iespējams pārvaldīt gan visus programmas veiktos datubāzes pieprasījumus, gan tīmekļa pieprasījumus. Arī Prefix rīks darbojas reāllaikā [47].

5.2.3. YourKit Java rīks

YourKit rīks pieejams gan uz Windows, gan Mac OS X, gan Linux, gan Solaris operētājsistēmām. Ar rīku iespējams veikt profilēšanu gan lokālām, gan attālināti strādājošām Java programmām. Rīks sniedz ļoti plašu informāciju par programmas patērētajiem sistēmas resursiem, izņēmuma gadījumiem, kā arī informāciju par koda kritiskajām vietām.

5.3. Produkcijas reāllaika pārvaldības rīki

Iepriekš apskatītie rīki ir paredzēti lietošanai programmatūras izstrādes laikā, lai atrastu kritiskas programmas vietas, uzlabotu veiktspēju un iegūtu informāciju par to, kā programma darbojas virtuālajā mašīnā. Bet ir nepieciešami arī rīki, kuri uzrauga produkcijā esošu sistēmu, pirmkārt, lai nodrošinātu stabilu sistēmas darbību, otrkārt, lai redzētu, kā programma darbojas ar produkcijas datiem, jo testēšanas vidēs nereti produkcijas dati dažādu apsvērumu dēļ nav pieejami. Tālāk nodaļā apskatīti populārākie rīki, kuri tiek lietoti, lai pārvaldītu produkcijā esošas sistēmas. Tā iemesla dēļ, ka apskatītie rīki darbojas ar produkcijā esošām sistēmām, ir svarīgi, lai šie rīki atstātu minimālu iespaidu uz sistēmas resursiem, kur darbojas pārvaldāmā programma.

5.3.1. NewRelic APM rīks

NewRelic APM ir komerciāls maksas rīks, kuru izstrādā kompānija NewRelic. Rīks ir lietojams ar vairāku populāru programmēšanas valodu programmām – Java, Ruby, Node.js, PHP, .NET, Python un Go. Rīks ir ļoti plašs savā funkcionalitātē, pieejami gan dažādi grafiki par pieprasījumu izpildes ātrumu, kļūdām tajos, gan par transakcijām, programmas pavedieniem utt. Programmām, kuras darbojas uz Java virtuālās mašīnas, pieejams arī speciāls Java veiktspējas analizēšanas modulis, kurš sniedz papildus detalizētu informāciju par programmu un virtuālās mašīnas stāvokli [43].

5.3.2. AppDynamics rīks

AppDynamics ir komerciāls maksas rīks, kurš pieder Cisco uzņēmumam. Arī AppDynamics rīks ir lietojams ar dažādu programmēšanas valodu programmām – Java, .NET, PHP, Node.js, C++, Python, un Go. Piedāvātā funkcionalitāte ir līdzīga kā NewRelic APM rīkam, papildus piedāvājot vizualizētus datus vienā vietā programmām, kuras sastāv no vairākiem neatkarīgiem komponentiem, piemēram, mikroservisu arhitektūras sistēmām [44].

5.3.3. Retrace rīks

Retrace rīks ir komerciāls maksas rīks, kuru izstrādā uzņēmums Stackify, tas ir pieejams gan Java, gan .NET programmēšanas valodām. Rīks ir veidots uz Stakify Prefix rīka bāzes (skatīt nodaļu 5.2.2.), pielāgojot to darbam ar produkcijas sistēmām [47].

6. JAVA INTEGRĀCIJAS OPTIMIZĀCIJA

Šaja nodaļā autors praktiski apskata un optimizē integrāciju, kura izstrādāta, lietojot Apache Camel ietvaru, aprakstot to lietojot Java DSL integrāciju aprakstīšanas veidu, Maven rīku bibliotēku atkarību pārvaldīšanai un Spring ietvaru Java objektu atkarību pārvaldībai. Visi mērījumi, ja nav norādīts citādāk, veikti ar Java versiju 1.8.0_112 ar Java HotSpot 64 bitu virtuālās mašīnas versiju 25.112-b15, uz sistēmas, kura strādā ar Intel Core i7-6500 2.50 GHz procesoru, 8.00 GB operatīvo atmiņu uz Windows 10 Home operētājsistēmas. Pilns integrācijas kods ar aprakstu, kā darbināt integrāciju, pieejams publiskā GitHub repozitorijā [46].

Integrācijas veiktspējas mērījumiem tiks lietotas trīs metodes:

- VisualVM rīks (skatīt nodaļu 5.1.1.);
- YourKit Java rīks (skatīt nodaļu 5.2.3.);
- mērot sistēmas laiku sekundes tūkstošdaļās pirms un pēc noteiktas operācijas izpildes.

Šīs metodes izvēlētas tā iemesla dēļ, ka tās plaši lietotas un tām ir pietiekams dokumentācijas apjoms, kā arī tās ir bez maksas, kas ļauj autora veiktos eksperimentus atkārtot citiem izstrādātājiem uz cita veida sistēmām.

Optimizācijas mērķis ir paātrināt integrācijas darbību: ziņu apstrādes ātrumu un startēšanas laiku, samazināt programmas atstāto iespaidu uz sistēmu, uz kuras tā darbojas, samazinot patērēto RAM daudzumu, samazinot CPU noslogotību. Kā galveno prioritāti autors izvirza ziņu apstrādes ātruma palielināšanu.

Izstrādātā integrācija tiks optimizēta divās secīgās grupās, vispirms tiks optimizēts autora rakstītais Java kods, lietotās bibliotēkas un integrācijas plūsma, tad tiks optimizēta HotSpot virtuālās mašīnas darbība, lietojot iepriekš apskatītos parametrus.

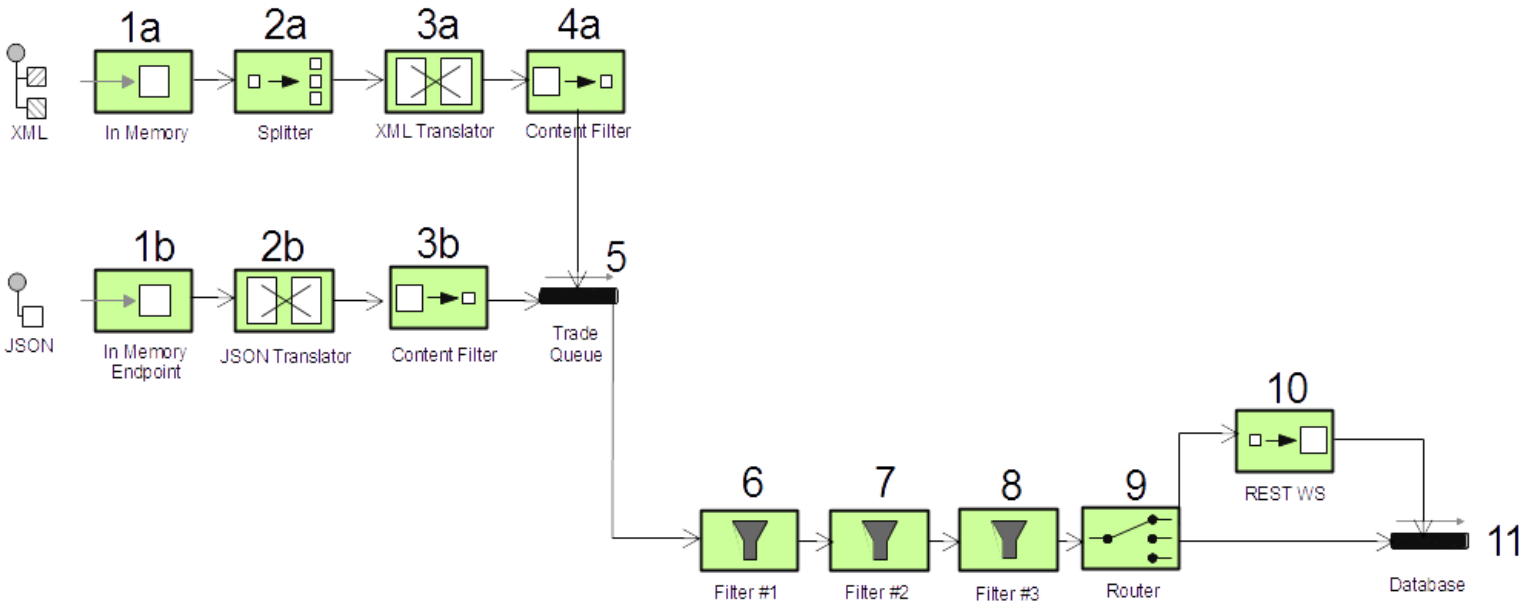
Integrācijas veiktspēja tiks mērīta, apstrādājot 2000 secīgas ziņas, kas simulē augstas datplūsmas integrācijas darbību, programma tiks darbināta 10 reizes pēc kārtas, kopā integrācijai apstrādājot 20000 ziņu, kas, pēc autora uzskatiem, ir pietiekami liels daudzums ziņu apstrādes, lai veiktspējas uzlabojumu vai pasliktinājumu gadījumā varētu uzskatīt, ka tieši autora veikto izmaiņu dēļ ir ietekmēta programmas ātrdarbība. Programmas startēšanās laiks tiks mērīts no Java virtuālās mašīnas startēšanas laikam līdz brīdim, kad programma ir pilnībā ielādēta Java

virtuālajā mašīnā un sākusi darbu. Programmas startēšanas laiks tiks mērīts, ņemtot vērā vidējo vērtība no jau minētajām 10 programmas darbināšanas reizēm.

Integrācijas modelī, kura veidota, lietojot integrācijas šablonu modeļus, redzama attēlā 6.1.. Integrācija veidota tā, lai tā iekļautu dažādus integrācijas šablonus, demonstrētu integrāciju izstrādi Java valodā, tajā pašā laikā iekļaujot tikai vienu ārēju programmatūras sistēmu – datubāzi, lai mērījumus būtu iespējams atkārtot uz citām sistēmām ar pēc iespējas mazāku darbu, kurš jāiegulda ārēju sistēmu uzstādīšanā, kā arī tā iemesla dēļ, ka tieši ārēju sistēmu ātrdarbību izstrādātāji parasti nevar konfigurēt vai optimizēt nekādā mērā, jo šīs sistēmas parasti ir trešās puses, pakalpojumu sniedzēju, īpašums. Integrācijā tiek lietota MySQL datubāze, kura ir atvērtā pirmkoda, bez maksas, taču tās vietā iespējams lietot jebkuru citu datubāzi, veicot nelielas konfigurācijas izmaiņas.

Izstrādātā integrācija iekļauj vienpadsmit integrācijas soļus:

- 1a un 1b ir divi galapunkti, pa kuriem integrācija saņem ziņas apstrādei. Abi galapunkti veidoti kā iekšēji atmiņas galapunkti, kuriem tiek padota ziņa, kuras ķermenis satur vai nu XML vai JSON ziņas datus;
- 2a – XML ziņu plūsmā katra ziņa var saturēt vairāk par vienu sagaidāmo objektu, tāpēc tiek veikta objekta sadalīšana;
- 2b un 3a – JSON un XML datu transformācija par Java objektiem, kuri satur visu teksta datos glabāto informāciju;
- 3b un 4a – dati tiek transformēti uz iekšēju integrācijas datu formātu, kurš satur tikai integrācijā nepieciešamos datus. Abas plūsmas transformē datus uz vienas klases objektiem;
- 5 – transformētie dati tiek ievietoti iekšējā atmiņas rindā tālākai kopīgai apstrādei;
- 6,7,8 – ziņas tiek filtrētas atbilstoši biznesa likumiem;
- 9 un 10 – nepieciešamības gadījumā, ja ziņā trūkst noteikti dati, tie tiek iegūti no tīmekļa servisa (integrācijā implementēts kā vienkārša metode, iepriekš aprakstīto iemeslu dēļ).
- 11 – dati tiek ierakstīti datu bāzē.



Attēls 6.1. Integrācijas plūsma

6.1. Integrācija ar standarta konfigurāciju

Nodaļā tiek apskatīta integrācijas sākotnējā veiktspēja pirms jebkāda veida optimizācijas. Iegūtie mērījumi tiks ņemti kā bāzlīnija tālākai integrācijas optimizācijai, lai noteiktu, vai un cik lielā mērā iespējams optimizēt konkrēto Java integrāciju, demonstrējot procesu, pēc kura var vadīties, optimizējot citas gan Java, gan citu programmēšanas valodu integrācijas un Java programmas. Sākotnējās integrācijas kods atrodams projekta repozitorija zarā *'initial'* kopā ar katru programmas darbināšanas reizes žurnālfailu. Pēc bāzlīnijas mērījumu veikšanas tiks identificētas integrācijas koda potenciālās optimizācijas iespējas, lietojot profilēšanas rīkus, kuras autors ievieš tālākajā nodaļā.

Veicot mērījumus sākotnējai integrācijas veiktspējai, tika noskaidrots, ka desmit reizes darbinot integrāciju Java virtuālā mašīna vidēji pilnībā tiek startēta 17.2489 sekundēs, ieskaitot programmas inicializēšanas laiku, kas vidēji ir 16.5734 sekundes. Divi tūkstoši ziņu desmit programmas darbināšanas reizēs vidēji tiek apstrādātas 108.6182 sekundēs. Vidēji tūkstotis ziņu, kuras tiek apstrādātas kā XML dati, patērē par 2.4 sekundēm ilgāku laiku nekā JSON datu apstrāde.

Attēlā 6.2. redzams 80 sekunžu fragments no virtuālās mašīnas kaudzes patērētās atmiņas, kurā var novērot, ka katrās desmit sekundēs, kurās programma apstrādā ziņas, atmiņa tiek tīrīta vienu līdz divas reizes. Gaiši zaļajā krāsā redzams, kā virtuālā mašīna automātiski pielāgo kaudzes atmiņu, kura atvēlēta programmai, katrā momentā, kad pieejamais atmiņas daudzums tiek mainīts, virtuālajai mašīnai ir jāveic aprēķini, lai noteiktu optimālo atmiņas daudzumu, aprēķinu veikšanu var likvidēt, norādot nepieciešamās atmiņas izmēru, kas, kā redzams attēlā 6.2., ir starp 800 MB un 900 MB. Kaudze sastāv no trijām atsevišķām daļām:

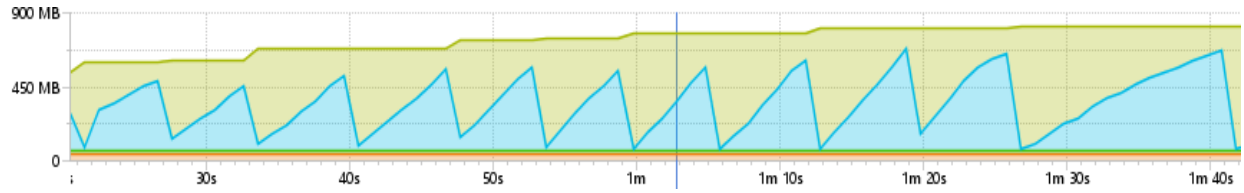
- zilajā krāsā tiek attēlots patērētais atmiņas daudzums īsa dzīves laika objektiem;
- tumši zaļajā krāsā tiek attēlots patērētais atmiņas daudzums vidēja dzīves laika objektiem;
- oranžajā krāsā tiek attēlots patērētais atmiņas daudzums ilga dzīves laika objektiem.

Jebkurš izveidotais objekts Java valodā sākotnēji tiek ievietots īsa dzīves laika objektu kaudzē, ja objekti no šīs kaudzes netiek likvidēti pēc noteikta skaita atkritumu savākšanas, tad objekts tiek pārvietots uz vidēja dzīves laika kaudzi. Līdzīgi notiek arī pāreja no vidēja dzīves laika kaudzes uz ilga dzīves laika objektu kaudzi. Pēc attēla 6.2. var secināt, ka ziņu apstrādes

laikā tiek veidots liels objektu daudzums, kuru dzīves laiks ir īss, un lielākās daļas šo objektu atmiņa tiek atbrīvota pie katras atkritumu savākšanas. Vidēju un ilgu dzīves laiku objektiem patērētā atmiņa praktiski nemainās visā ziņu apstrādes laikā, aptuveni patērējot 38MB atmiņas.

Samazinot objektu veidošanas daudzumu, teorētiski būtu iespējams samazināt darbu, kas veicams atkritumu savācējam, tādējādi uzlabojot veiktspēju, autors apzinās, ka daļa no īslaicīgajiem objektiem tiek veidota, ielasot ziņas no teksta faila pirms apstrādes, šo objektu veidošana parādās profilēšanas datos, taču tas netiek ņemts vērā ziņu apstrādes uzņemtajā laikā. Ziņu ielasīšanas veids netiks mainīts, veicot programmas optimizēšanu, lai šo objektu ietekme paliktu konstanta visās programmas darbināšanas reizēs.

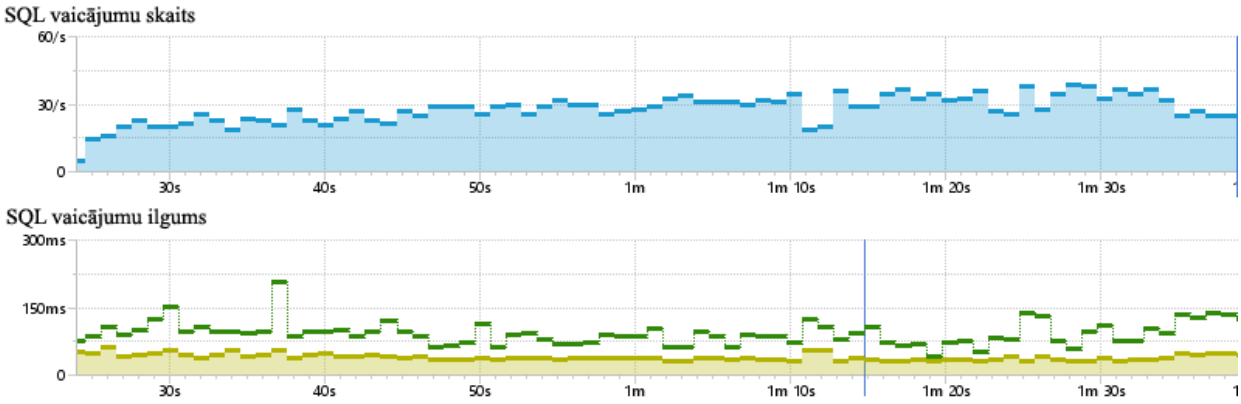
Programmas darbības laikā kopā Java virtuālajā mašīnā pēc profilēšanas rīka datiem tiek ielādētas 11734 klases, kuras lielākā daļa ir ietvaru un dažādu bibliotēku klases. Spring ietvars nodrošina automātisku konfigurāciju, kas samazina programmētajam veicamo darbu, izstrādājot projektu, taču šī automātiskā konfigurācija un klašu ielāde tiek veikta arī priekš programmā neesošas funkcionalitātes, ielādēto klašu skaitu, pēc autora domām, šo skaitu iespējams samazināt specifiski norādot ietvaram veicamo konfigurāciju, kā arī pārskatot projektā iekļautās bibliotēkas.



Attēls 6.2. Integrācijas patērētā virtuālās mašīnas kaudzes atmiņa

Attēlā 6.3. redzams SQL vaicājumu skaits sekundē, kas ziņu apstrādes laikā svārstās starp 15 un 40 vaicājumiem sekundē, kā arī vaicājumiem patērētais laiks milisekundēs, kas galvenokārt svārstās starp 15 un 75 milisekundēm. 4000 ziņu apstrādes laikā tiek izpildīti 12000 vaicājumi, vidējais vaicājuma laiks ir 26 milisekundes, maksimālais 200 milisekundes.

Vecot katru vaicājumu, autora Java kodā tiek veidoti trīs jauni objekti, kas nav liels skaits, taču šajā skaitā netiek ņemti vērā objekti, kuri tiek veidoti bibliotēku kodā, kuras veido savienojumu ar datubāzi, kā rezultātā šis skaits, pēc autora domām, ir ievērojami lielāks. Izpildīto vaicājumu atstāto ietekmi uz veiktspēju iespējams samazināt, lietojot vaicājumu grupēšanu vienā datubāzes savienojumā, sesijā un transakcijā.



Attēls 6.3. SQL vaicājumu ietekme uz veiktspēju

Veicot CPU profilēšanu integrācijai, tika noskaidrots, ka lielāko ietekmi uz procesora darbību (skatīt attēlu 6.4.) atstāj datu sagatavošana un ierakstīšana datu bāzē (attēlā atzīmēta ar burtu A), datu konvertācija uz integrācijas iekšējo datu formātu (B), XML datu pārveidošana uz Java objektu (C), XML failu sadalīšana, lietojot XPath valodu (D), žurnālfaila rakstīšana (E).

CPU samples | Thread CPU Time

Snapshot

Hot Spots - Method		Self Time [%]	Self Time
com.mchange.v2.async.ThreadPoolAsynchronousRunner$PoolThread.**run** ()	A		234,186 ms (56%)
com.mysql.jdbc.util.ReadAheadInputStream.**fill** ()	A		117,221 ms (28%)
org.apache.coyote.AbstractProtocol$AsyncTimeout.**run** ()			18,190 ms (4.3%)
lv.lu.df.java.integration.converter.CompanyDTOConverter.**convert** ()	B		12,647 ms (3%)
org.apache.camel.converter.jaxb.JaxbDataFormat.**unmarshal** ()	C		6,972 ms (1.7%)
org.apache.camel.builder.xml.XPathBuilder.**doInEvaluateAs** ()	D		3,484 ms (0.8%)
org.apache.camel.converter.jaxp.XmlConverter.**toDOMDocument** ()	C		2,363 ms (0.6%)
org.apache.camel.converter.jaxp.XmlConverter.**toResult** ()	C		1,482 ms (0.4%)
org.springframework.boot.loader.LaunchedURLClassLoader.**loadClass** ()			1,228 ms (0.3%)
ch.qos.logback.core.joran.spi.ConsoleTarget$1.**write** ()	E		1,195 ms (0.3%)
org.hibernate.jpa.boot.archive.internal.ArchiveHelper.**getBytesFromInputStream** ()	A		1,084 ms (0.3%)
org.apache.commons.io.FileUtils.**openInputStream** ()			1,081 ms (0.3%)
org.springframework.boot.loader.data.RandomAccessDataFile$DataInputStream.**<init>** ()			870 ms (0.2%)
com.mysql.jdbc.util.ReadAheadInputStream.**available** ()	A		781 ms (0.2%)
org.hibernate.engine.jdbc.spi.SqlStatementLogger.**logStatement** ()	A		768 ms (0.2%)
org.apache.commons.io.IOUtils.**copyLarge** ()			688 ms (0.2%)
com.mysql.jdbc.MysqlIO.**send** ()	A		627 ms (0.1%)
com.mysql.jdbc.SQLError.**convertShowWarningsToSQLWarnings** ()	A		520 ms (0.1%)
org.apache.camel.converter.jaxp.StaxConverter.**createXMLStreamReader** ()			493 ms (0.1%)
com.fasterxml.jackson.databind.deser.impl.FieldProperty.**deserializeAndSet** ()			402 ms (0.1%)
org.hibernate.action.internal.UnresolvedEntityInsertActions.**resolveDependentActions** ()	A		393 ms (0.1%)

Attēls 6.4. Integrācijā lietoto metožu ietekme uz procesoru

XML datu sadalīšanai bez XPath valodas iespējams lietot VTD-XML vai XML marķierus, kuri var samazināt datu sadalīšanai patērēto laiku.

Žurnālfaila rakstīšanu integrācijas sākotnējā versijā var veikt arī bibliotēkas, konfigurējot žurnālfaila rakstīšanas likumus, iespējams samazināt žurnālfaila rakstīšanas atstāto iespaidu uz integrācijas veiktspēju, integrācijas sākotnējā versija, apstrādājot 4000 ziņas, izveido 7156 KB – 7184 KB lielu žurnālfailu.Kā redzams attēlā 6.4., žurnālfailu rakstīšana tiek veikta ar automātiski nodrošināto bibliotēku Logback, kas, pēc Java žurnālfailu rakstīšanas pētījuma, ir ātrākā no Java žurnālfailu rakstīšanas bibliotēkām [45].

6.2. Integrācijas Java koda optimizācija

Šajā nodaļā autors apskata praktiski veikto integrācijas optimizāciju rakstītajam Java kodam, integrācijas plūsmai, kā arī Java baitkodam, mērķis ir paātrināt ziņu apstrādei patērēto laiku, neatstājot negatīvu ietekmi uz patērēto virtuālās atmiņas daudzumu vai to samazinot. Optimizētas integrācijas kods atrodams projekta repozitorija zarā *optimized* kopā ar katru programmas darbināšanas reizes žurnālfailu.

Nodrošinot to, ka tikai nepieciešamās bibliotēkas tiek iekļautas kompilētā Java arhīvā (izpildāms .jar faila formāts), arhīva izmērs tika samazināts no 49 MB uz 25 MB lielu arhīvu. Autors sagaida, ka no šīs izmaiņas Java virtuālās mašīnas ielādēto klašu skaits tiks ievērojami samazināts, kā arī tiks samazināts virtuālās mašīnas startēšanas laiks.

Nevajadzīgu bibliotēku iekļaušana Java projektos ir ļoti plaši novērojama, jo, laikam ejot, izstrādātāji iekļauj arvien jaunas bibliotēkas, un programmas darbināšanas vai kompilēšanas laikā nav iespējams noteikt, vai nav iekļauta kāda lieka bibliotēka. Tā kā lielākā daļa Java projektu tiek veidoti, lietojot vai nu Maven vai Gradle bibliotēku atkarības rīku, kur katra no iekļautajām bibliotēkām var iekļaut arī citas bibliotēkas, kuras nepieciešamas, var gadīties, ka projektā tiek iekļautas vairākas vienas bibliotēkas versijas vai tiek iekļautas nevajadzīgas bibliotēkas. Vienīgais veids, kā atbrīvoties no projektā nevajadzīgi iekļautajām bibliotēkām, ir apzināties katras iekļautās bibliotēkas atkarības, lai kāda bibliotēka netiktu iekļauta vairākas reizes, vai pārbaudīt programmas darbību, mēģinot bibliotēkas izņemt no projekta.

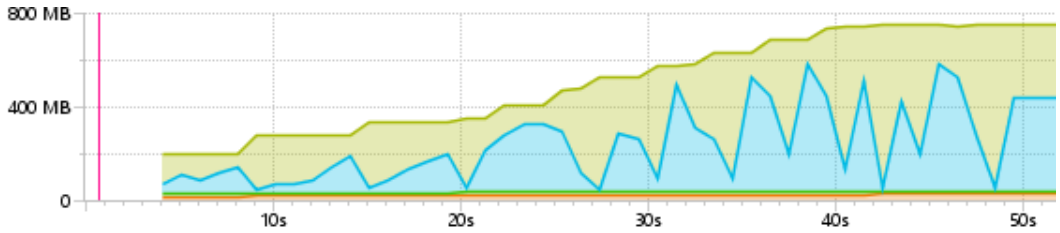
Ļaujot žurnālfailā rakstīt tikai autora izstrādātās integrācijas kodam un Spring ietvaram, kurš programmas startēšanas brīdī izvada integrācijas konfigurāciju, 2000 ziņu apstrādes laikā žurnālfailā tiek izvadīti teksta dati ar lielumu 3884 KB, kas ir 45% uzlabojums, saglabājot visus nepieciešamos datus žurnāla failā.

Autoram testējot integrāciju, tika noskaidrots, cik lielu daļu no ziņām katrs filtrs atzīst par nederīgām, tāpēc filtri integrācijas plūsmā tika pārkārtoti tādā secībā, lai tas filtrs, kurš filtrē vislielāko daļu no visām ziņām, tiktu pārbaudīts pirmais, tādējādi nodrošinot to, ka plūsmā netiek veiktas nevajadzīgas pārbaudes.

Optimizācijas nolūkos autors implementēja grupveida SQL datu rakstīšanu, integrācijas sākotnējā versijā, kā apskatīts iepriekš, tika veikti 12000 SQL vaicājumi, katrs savā savienojumā ar datubāzi, savā sesijā, savā transakcijā pēc optimizēšanas SQL datu rakstīšana notiek pie katriem 100 datu modeļa objektiem, kur katram objektam vidēji ir vēl 3 apakš-objekti, kuri tiek rakstīti citā tabulā, rezultātā katras 400 datubāzes sesijas un transakcijas ir aizvietotas ar vienu. SQL vaicājumi tiek izpildīti, kad ir sakrāti 100 objekti vai vienu reizi 60 sekunžu laikā, atkarībā no tā, kurš nosacījums izpildās pirmais. Datu ievietošana pēc 60 sekundēm implementēta, lai izvairītos no situācijas, ka datubāzē ir novecojuši dati ilgāk par nolikto laika periodu. Rezultātā vaicājumu skaits nemainās, taču savienojumu, sesiju un transakciju skaits tiek ievērojami samazināts.

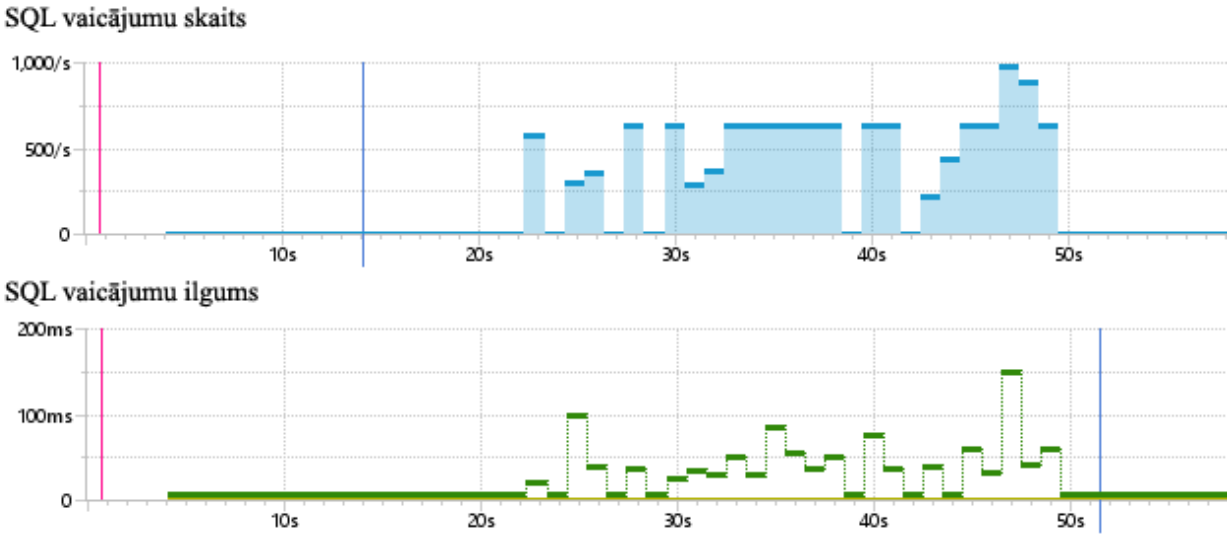
Veicot mērījumus integrācijai pēc aprakstītajām optimizācijām, tika noskaidrots, ka vidēji desmit reizes darbinot integrāciju Java virtuālā mašīna pilnībā tiek startēta 10.1177 sekundēs, ieskaitot programmas inicializēšanas laiku, kas vidēji ir 9.4499 sekundes. Virtuālās mašīnas startēšanas laiks ir uzlabots par 43%, salīdzinot ar integrācijas sākotnējo versiju, šāds uzlabojums ir, pateicoties bibliotēku daudzuma samazinājumam, kā rezultātā Java virtuālajai mašīnai, sākot darbību, jāielasa par 24 MB mazāk datu. Divi tūkstoši ziņu desmit programmas darbināšanas reizēs vidēji tiek apstrādātas 11.8122 sekundēs, kas ir uzlabojums par 89%, salīdzinot ar integrācijas sākotnējo versiju, šāds uzlabojums galvenokārt ir, pateicoties datu grupveida rakstīšanai datubāzē, kā arī ietekmi uz laiku, taču mazāku, atstāj pārējās aprakstītās optimizācijas, kuras tika veiktas.

Programmas darbības laikā kopā Java virtuālajā mašīnā pēc profilēšanas rīka datiem tiek ielādētas 9080 klases, kas ir par 23% mazāk nekā sākotnējā versijā. Attēlā 6.5. redzams, ka optimizēta integrācija patērē par apmēram 50-70 MB mazāk virtuālās atmiņas nekā integrācijas sākotnējā versija. Optimizētā integrācijā vēl uzskatāmāk redzams tas, kā virtuālā mašīna nepārtraukti palielina kaudzei atvēlēto atmiņu, līdz sasniedz 738 MB atzīmi, kas ir maksimālais atmiņas daudzums, kas nepieciešams integrācijai.



Attēls 6.5. Integrācijas patērētā virtuālās mašīnas kaudzes atmiņa

Attēlā 6.6. redzama grupveida SQL izpilde, redzams, ka tas pats daudzums datu tiek ievietoti daudz īsākā laika periodā, jo vairākas datu ievietošanas tiek veiktas vienā sesijā, vienā transakcijā.



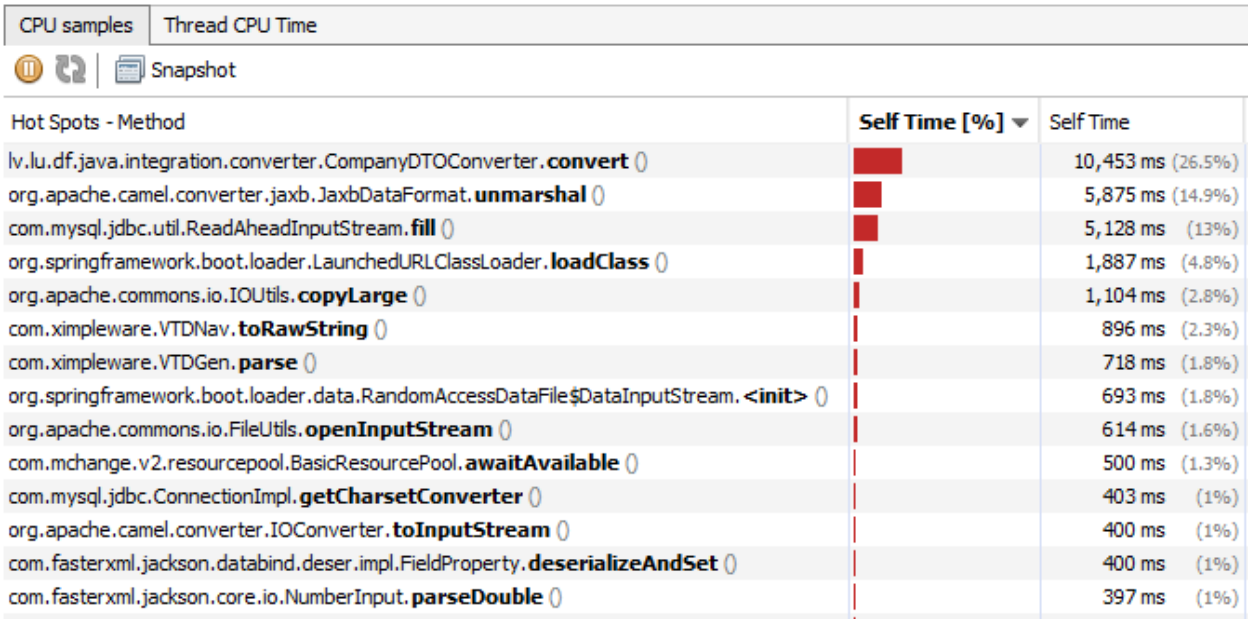
Attēls 6.6. SQL vaicājumu ietekme uz veiktspēju

6.3. Virtuālās mašīnas optimizācija

Šajā nodaļā tiks aprakstīti virtuālās mašīnas parametri, kuri lietojami, lai uzlabotu konkrētās integrācijas ātrdarbību. Par bāzi virtuālās mašīnas optimizācijai tiek ņemta optimizēta integrācija no nodaļas 6.2.. Nodaļā ir aprakstīti optimālie parametri un to lietošanas iemesli tieši priekš izstrādātās integrācijas. Katrā situācijā lietojamie parametri vai parametru vērtības var atšķirties, tāpēc ir nepieciešams labi izprast optimizējamo Java programmu, lai nav eksperimentāli jāmēģina visi virtuālās mašīnas parametri.

Integrācijas ātrdarbība tiek pārbaudīta 10 reizes ar katru aprakstīto parametru, tikai, ja visos mērījumos ir novērojami neapšaubāmi ātrdarbības uzlabojumi, autors uzskata, ka konkrētais parametrs ir uzlabojums, salīdzinot pret Java virtuālās mašīnas noklusētajām vērtībām.

Attēlā 6.7. redzamas integrācijas metodes, kuras pildot, tiek patērēts visvairāk laika pirms Java virtuālās mašīnas optimizācijas veikšanas. Redzams, ka vislielāko ietekmi uz procesoru un tādējādi arī uz programmas darbības ātrumu atstāj metode *convert*, kurā tiek veikta datu konvertācija no sākotnējā tipa uz integrācijas iekšējo datu modeli, kā arī metode *unmarshal*, kurā tiek veikta datu pārveidošana no XML teksta datiem uz Java objektu, un metode *fill*, kura atbild par datu sūtīšanu uz datubāzi.



Attēls 6.7. Java metožu ietekme uz procesoru pirms virtuālās mašīnas optimizācijas

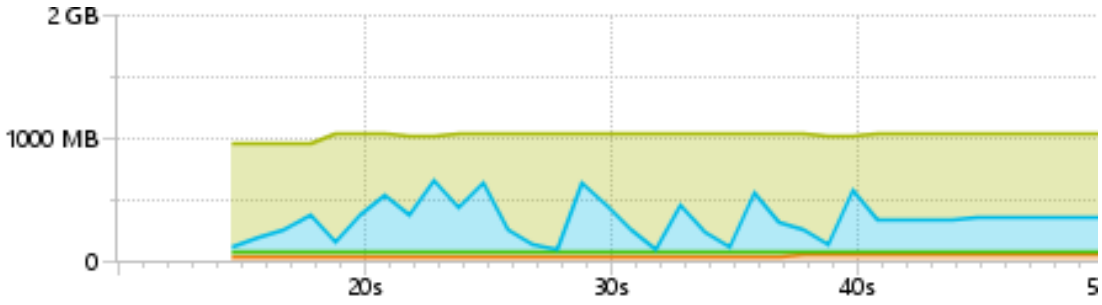
-XX:MaxMetaspaceSize=75m – atvēlētais atmiņas daudzums klašu metadatiem, lai izvairītos no virtuālās mašīnas dinamiskas atmiņas lieluma mainīšanas. Lielums 75 MB noteikts pēc vairākām darbināšanas reizēm, kurās tika noteikts, ka vidējais metadatu patērētās atmiņas daudzums, ja tiek dota neierobežota vieta datiem, ir 55-60 MB, kuri tika pareizināti ar koeficientu 1.25, lai gadījumā, ja kādā no darbināšanas reizēm tiek patērēti vairāk dati, neciestu integrācijas veiktspēja.

-Xnoclassgc – izslēdz atkritumu savākšanu klašu metadatiem, tā kā konkrētajā programmā nav liels klašu daudzums un lielākā daļa objektu ir ilga dzīves laika, klašu metadatu savākšana var potenciāli patērēt liekus resursus.

-XX:+AggressiveOpts – ieslēgta agresīvā veiktspējas automātiska optimizācija.

Kaudzes izmēru norādīšana ar parametriem -Xmn, -Xms, -Xmx pretēji gaidītajam palēnina programmas darbību. Lai gan ar parametriem iespējams panākt konstantu virtuālās mašīnas

kaudzes izmēru (skatīt attēlu 6.8.) un samazināt veikto atkritumu savākšanas procesu skaitu, pati procesa darbība tiek palēnināta par aptuveni 7-10%.



Attēls 6.8. Integrācijas patērētā virtuālās mašīnas kaudzes atmiņa

Kaudzes izmēru norādīšana, pēc autora domām, lielāku ieguvumu spēj dot ilglaicīgāku, vairāk atmiņu patērējošu procesu gadījumos, kā arī lai norādītu maksimālo atmiņu, ko var patērēt process.

CPU samples | Thread CPU Time

Snapshot

Hot Spots - Method	Self Time [%]	Self Time
lv.lu.df.java.integration.converter.CompanyDTOConverter.**convert** ()		9,636 ms (26.1%)
com.mysql.jdbc.util.ReadAheadInputStream.**fill** ()		5,698 ms (15.4%)
org.apache.camel.converter.jaxb.JaxbDataFormat.**unmarshal** ()		5,389 ms (14.6%)
com.ximpleware.VTDNav.**toRawString** ()		1,196 ms (3.2%)
org.springframework.boot.loader.LaunchedURLClassLoader.**loadClass** ()		1,173 ms (3.2%)
org.apache.commons.io.IOUtils.**copyLarge** ()		990 ms (2.7%)
com.ximpleware.VTDGen.**parse** ()		798 ms (2.2%)
org.springframework.boot.loader.data.RandomAccessDataFile$DataInputStream.**<init>** ()		775 ms (2.1%)
org.apache.commons.io.FileUtils.**openInputStream** ()		614 ms (1.7%)
ch.qos.logback.core.joran.spi.ConsoleTarget$1.**write** ()		602 ms (1.6%)
org.springframework.boot.loader.data.RandomAccessDataFile$DataInputStream.**close** ()		500 ms (1.4%)
com.fasterxml.jackson.core.io.NumberInput.**parseDouble** ()		398 ms (1.1%)

Attēls 6.9. Java metožu ietekme uz procesoru pēc virtuālās mašīnas optimizācijas

Attēlā 6.9. redzams, ka ietekme uz procesoru pēc integrācijas startēšanas ar apskatītajiem HotSpot virtuālās mašīnas parametriem ir mazinājusies, procentuāli redzams, ka metožu ietekme saglabājusies iepriekšējos apmēros, taču patērētais laiks milisekundēs ir samazinājies par 7-8%. Kopumā pēc optimizācijas Java virtuālās mašīnas laiks palielinājies vidēji desmit darbināšanas reizēs no 10.1177 sekundēm līdz 10.4177 sekundēm, kas ir par 2.8% lēnāk nekā pirms tam, taču 2000 ziņu apstrādei patērētais laiks, vidēji desmit darbināšanas reizēs, samazinājies no 11.812 sekundēm uz 10.872 sekundēm, kas ir 8 % uzlabojums.

REZULTĀTI

Autors ir iepazinies ar integrācijas šablonu un pareizas izstrādes teoriju, kā arī iepazinies ar Java programmēšanas valodas ietvariem un rīkiem, kuri paredzēti integrāciju izstrādei. Ir izpētītas un aprakstītas Java programmu optimizācijas iespējas, praktiski demonstrējot veiktspējas sekas, ja tiek lietotas noteiktas konstrukcijas, neiepazīstoties ar to atstāto iespaidu uz programmas veiktspēju. Ir apskatīti biežāk lietotie HotSpot virtuālās mašīnas parametri, ar kuriem var, esot labām zināšanām par optimizējamo programmu, pielāgot virtuālo mašīnu optimālai programmas darbībai.

Veiktās teorijas izpēte autoram sniedz zināšanas par korektu integrāciju izstrādi, Java virtuālās mašīnas darbību un efektīvu Java programmu izstrādi, ko autors darbā demonstrē veiktajos eksperimentos un darba praktiskajā daļā.

Autors praktiskajā daļā izstrādāja integrāciju, lietojot iepriekš apskatīto Apache Camel integrācijas ietvaru, ar dažādu mērījumu un rīku palīdzību noteica tās veiktspējas parametrus. Izstrādāto integrāciju autors optimizēja, balstoties uz iepriekš iegūtajām teorētiskajām zināšanām. Salīdzinot sākotnējo un gala versiju, apstrādā 2000 ziņu par 89% ātrāk.

Praktiskajā daļā autors ir demonstrējis, kā optimizēt integrācijas un citas Java programmas, balstoties uz teorētiskām zināšanām un izpratni par programmas darbību, autora veiktās optimizācijas nav absolūtas priekš jebkuras Java integrācijas, jo katrs gadījums ir citādāks, taču autora veiktais process ir praktisks piemērs, kā veikt optimizācijas citām integrācijām.

Darba izstrādē autora izvirzītie mērķi ir sasniegti, ir iegūtas nepieciešamās teorētiskās zināšanas par augstas veiktspējas integrāciju izstrādes Java valodā, ir iegūtas zināšanas un praktiska pieredze integrāciju optimizācijā un demonstrēts optimizācijas process, kāds veicams, lai uzlabotu integrāciju veiktspēju.

SECINĀJUMI

Izstrādājot bakalaura darbu, tika secināts tas, ka optimizāciju veikšana un veiktspējas uzlabošana ir iespējama, pēc autora domām, jebkurai Java programmai, taču katrā gadījumā var būt veicamas cita veida optimizācijas. Veicot Java koda optimizāciju, ir gadījumi, kad izstrādātājiem ir jāpieņem lēmumi vai nu par labu nelieliem veiktspējas uzlabojumiem, vai koda lasāmībai un vienkāršībai. Analizējot apskatīto teoriju, var secināt, ka optimizācijas lielākajā daļā gadījumu veicamas visos Java programmas līmeņos - gan izstrādātāju rakstītajā kodā, gan virtuālās mašīnas pielāgošanā, gan savstarpējā komunikācijā ar trešās puses sistēmām.

Pēc teorijas apskates par profilēšanas rīkiem autors var secināt, ka tirgū ir plašs rīku klāsts, kuru mērķis ir palīdzēt programmētājiem izstrādāt programmatūru ar augstu veiktspēju. Katrā no rīku kategorijām ir pieejami gan maksas komerciālie rīki, gan bezmaksas rīki ar plašu funkcionalitāti.

Izstrādājot praktisko daļu, autors secina, ka optimizāciju veikšana ir iteratīvs process, jo, pat lietojot veiktspējas mērīšanas rīkus un profilēšanas rīkus, pēc vienas vai dažām programmas darbināšanas reizēm nav iespējams viennozīmīgi pateikt, vai veiktā optimizācija ir kaut ko ietekmējusi pozitīvi, negatīvi vai nav atstājusi nekādu iespaidu uz programmas veiktspēju. Praktiskai optimizāciju veikšanai nepieciešams liels, dažāds, vēlams produkcijas vidi simulējošs datu apjoms.

IZMANTOTĀ LITERATŪRA UN AVOTI

1. *Messaging Patterns Overview, Gregor Hohpe, Bobby Wolf* [tiešsaiste] [atsauce 2017. gada 1. maijs] Pieejams: http://www.enterpriseintegrationpatterns.com/patterns/messaging/
2. *Pattern: Shared database, Chris Richardson* [tiešsaiste] [atsauce 2017. gada 1. maijs] Pieejams: http://microservices.io/patterns/data/shared-database.html
3. *File Transfer, Gregor Hohpe, Bobby Wolf* [tiešsaiste] [atsauce 2017. gada 2. maijs] Pieejams: http://www.enterpriseintegrationpatterns.com/patterns/messaging/FileTransferIntegration.html
4. *What is RPC, Microsoft TechNet* [tiešsaiste] [atsauce 2017. gada 1. maijs] Pieejams: https://technet.microsoft.com/en-us/library/cc787851(v=ws.10).aspx
5. *Messaging Systems, Oracle* [tiešsaiste] [atsauce 2017. gada 1. maijs] Pieejams: https://docs.oracle.com/cd/E26576_01/doc.312/e24949/messaging-systems-introduction.htm#GMTOV00025
6. *Point-to-Point Channel, Gregor Hohpe, Bobby Wolf* [tiešsaiste] [atsauce 2017. gada 2. maijs] Pieejams: http://www.enterpriseintegrationpatterns.com/patterns/messaging/PointToPointChannel.html
7. *Publish-Subscribe Channel, Gregor Hohpe, Bobby Wolf* [tiešsaiste] [atsauce 2017. gada 2. maijs] Pieejams: http://www.enterpriseintegrationpatterns.com/patterns/messaging/PublishSubscribeChannel.html
8. *Invalid Message Channel, Gregor Hohpe, Bobby Wolf* [tiešsaiste] [atsauce 2017. gada 2. maijs] Pieejams: http://www.enterpriseintegrationpatterns.com/patterns/messaging/InvalidMessageChannel.html
9. *Message Routing, Spring Integration* [tiešsaiste] [atsauce 2017. gada 2. maijs] Pieejams: http://docs.spring.io/spring-integration/reference/html/messaging-routing-chapter.html
10. *Message Routing Intro, Gregor Hohpe, Bobby Wolf* [tiešsaiste] [atsauce 2017. gada 2. maijs] Pieejams: http://www.enterpriseintegrationpatterns.com/patterns/messaging/MessageRoutingIntro.html
11. *Message Filter, Gregor Hohpe, Bobby Wolf* [tiešsaiste] [atsauce 2017. gada 2. maijs] Pieejams: http://www.enterpriseintegrationpatterns.com/patterns/messaging/Filter.html

12. *Messaging Endpoints, Spring Integration* [tiešsaiste] [atsauce 2017. gada 4. maijs] Pieejams: http://docs.spring.io/spring-integration/docs/2.0.0.RELEASE/reference/html/messaging-endpoints-chapter.html
13. *Oracle SOA Suite, Oracle* [tiešsaiste] [atsauce 2017. gada 11. maijs] Pieejams: http://www.oracle.com/us/products/middleware/soa/suite/overview/index.html
14. *JBoss SOA Platform, Red Hat* [tiešsaiste] [atsauce 2017. gada 11. maijs] Pieejams: https://access.redhat.com/products/red-hat-jboss-soa-platform
15. *What is Mule ESB?, Mule* [tiešsaiste] [atsauce 2017. gada 11. maijs] Pieejams: https://www.mulesoft.com/resources/esb/what-mule-esb
16. *Spring Framework History, Amit Sharma* [tiešsaiste] [atsauce 2017. gada 11. maijs] Pieejams: http://springtutorials.com/spring-framework-history/
17. *Spring Integration Java DSL, MvnRepository* [tiešsaiste] [atsauce 2017. gada 12. maijs] Pieejams: https://mvnrepository.com/artifact/org.springframework.integration/spring-integration-java-dsl
18. *Showdown: Integration Framework vs Enterprise Service Bus, Kai Wahner* [atsauce 2017. gada 12. maijs] Pieejams: https://www.slideshare.net/KaiWaehner/showdown-integration-framework-spring-integration-apache-camel-vs-enterprise-service-bus-esb
19. *Camel Core, MvnRepository* [tiešsaiste] [atsauce 2017. gada 13. maijs] Pieejams: https://mvnrepository.com/artifact/org.apache.camel/camel-core
20. *DSL, Apache Camel* [tiešsaiste] [atsauce 2017. gada 13. maijs] Pieejams: http://camel.apache.org/dsl.html
21. *Selective Consumer, Apache Camel* [tiešsaiste] [atsauce 2017. gada 15. aprīlis] Pieejams: http://camel.apache.org/selective-consumer.html
22. *The Java Virtual Machine Specification, Oracle* [tiešsaiste] [atsauce 2017. gada 17. aprīlis] Pieejams: https://docs.oracle.com/javase/specs/jvms/se8/html/index.html
23. *Java version and vendor data analyzed: 2016 edition, Plumbr* [tiešsaiste] [atsauce 2017. gada 15. aprīlis] Pieejams: https://plumbr.eu/blog/java/java-version-and-vendor-data-analyzed-2016-edition
24. *J9 Virtual Machine, IBM* [tiešsaiste] [atsauce 2017. gada 15. aprīlis] Pieejams: https://www.ibm.com/support/knowledgecenter/SSYKE2_7.0.0/com.ibm.java.lnx.70.doc/user/java_jvm.html

25. *Azul Zing JVM, Azul* [tiešsaiste] [atsauce 2017. gada 15. aprīlis] Pieejams: https://www.azul.com/products/zing/
26. *JVM JIT optimization techniques, AdvancedWeb* [tiešsaiste] [atsauce 2017. gada 21. aprīlis] Pieejams: https://advancedweb.hu/2016/05/27/jvm_jit_optimization_techniques/
27. *JVM Options, Java Code Geeks* [tiešsaiste] [atsauce 2017. gada 15. aprīlis] Pieejams: https://www.javacodegeeks.com/2011/07/jvm-options-client-vs-server.html
28. *Java Heap Start In Practice. Dont Panic* [tiešsaiste] [atsauce 2017. gada 15. aprīlis] Pieejams: http://dontpanic.42.nl/2013/08/java-heap-start-xms-in-practice.html
29. *Java SE Embedded 8 Compact Profiles Overview, Oracle* [tiešsaiste] [atsauce 2017. gada 17. aprīlis] Pieejams: http://www.oracle.com/technetwork/java/embedded/resources/tech/compact-profiles-overview-2157132.html
30. *Project Jigsaw, OpenJDK* [tiešsaiste] [atsauce 2017. gada 17. aprīlis] Pieejams: http://openjdk.java.net/projects/jigsaw/
31. *Java Creed – Comparing the Performance of some List implementations, Github* [tiešsaiste] [atsauce 2017. gada 16. aprīlis] Pieejams: https://github.com/javacreed/comparing-the-performance-of-some-list-implementations
32. *List Implementations, Oracle* [tiešsaiste] [atsauce 2017. gada 16. aprīlis] Pieejams: https://docs.oracle.com/javase/tutorial/collections/implementations/list.html
33. *Map Implementations, Oracle* [tiešsaiste] [atsauce 2017. gada 22. aprīlis] Pieejams: https://docs.oracle.com/javase/tutorial/collections/implementations/map.html
34. *The Performance of Exception, Aleksey Shipilëv* [tiešsaiste] [atsauce 2017. gada 17. aprīlis] Pieejams: https://shipilev.net/blog/2014/exceptional-performance/
35. *Is it better to use System.arraycopy than a for loop for copying arrays?, Stackoverflow* [tiešsaiste] [atsauce 2017. gada 17. aprīlis] Pieejams: http://stackoverflow.com/questions/18638743/is-it-better-to-use-system-arraycopy-than-a-for-loop-for-copying-arrays#answer-18639042
36. *JEP 198: Delayed, Coman Hamilton* [tiešsaiste] [atsauce 2017. gada 01. maijs] Pieejams: https://jaxenter.com/json-api-dropped-java-9-113028.html

37. *Java Persistence Performance, James Sutherland* [tiešsaiste] [atsauce 2017. gada 04. maijs] Pieejams: http://java-persistence-performance.blogspot.com/2013/05/batch-writing-and-dynamic-vs.html
38. *Java Tools and Technologies Landscape Report, ZeroTurnaround* 2016 [tiešsaiste] [atsauce 2017. gada 22. aprīlis] Pieejams: https://zeroturnaround.com/rebellabs/java-tools-and-technologies-landscape-2016/
39. *Introduction to VisualVM, VisualVM* [tiešsaiste] [atsauce 2017. gada 22. aprīlis] Pieejams: http://visualvm.java.net/intro.html
40. *JProfiler EJ Technologies* [tiešsaiste] [atsauce 2017. gada 22. aprīlis] Pieejams: https://www.ej-technologies.com/products/jprofiler/features.html
41. *Java Mission Control User's Guide, Oracle* [tiešsaiste] [atsauce 2017. gada 22. aprīlis] Pieejams: https://www.ej-technologies.com/products/jprofiler/features.html
42. *XRebel, ZeroTurnaround* [tiešsaiste] [atsauce 2017. gada 24. aprīlis] Pieejams: https://zeroturnaround.com/software/xrebel/
43. *New Relic APM Features, New Relic* [tiešsaiste] [atsauce 2017. gada 25. aprīlis] Pieejams: https://newrelic.com/application-monitoring/features
44. *Application Performance Management, App Dynamics* [tiešsaiste] [atsauce 2017. gada 25. aprīlis] Pieejams: https://www.appdynamics.com/product/application-performance-management/
45. *Benchmarking Java Logging Frameworks* [tiešsaiste] [atsauce 2017. gada 25. aprīlis] Pieejams: https://www.loggly.com/blog/benchmarking-java-logging-frameworks/
46. Arvis Taurenis. IntegrationPerformance Repository, GitHub [tiešsaiste] [atsauce 2017. gada 25. aprīlis] Pieejams: https://github.com/arvist/IntegrationPerformance
47. Stackify Prefix Stackify, [tiešsaiste] [atsauce 2017. gada 20. aprīlis] Pieejams: https://stackify.com/prefix/
48. Arvis Taurenis. IntegrationComponentsPerformance Repository, GitHub [tiešsaiste] [atsauce 2017. gada 22. maijs] Pieejams: https://github.com/arvist/IntegrationComponentsPerformance

Bakalaura darbs „Java integrāciju optimizācijas iespējas” izstrādāts LU Datorikas fakultātē.

Ar savu parakstu apliecinu, ka pētījums veikts patstāvīgi, izmantoti tikai tajā norādītie informācijas avoti un iesniegtā darba elektroniskā kopija atbilst izdrukai.

Autors Arvis Taurenis:_____

Rekomendēju darbu aizstāvēšanai

Vadītājs: profesors Dr. dat. Uldis Straujums _____ 29.05.2017.

Recenzents: asociētais profesors Dr. comp. sc. Edgars Celms

Darbs iesniegts Datorikas fakultātē 29.05.2017.

Dekāna pilnvarotā persona:

vecākā metodiķe Ārija Sproģe _____

Darbs aizstāvēts bakalaura gala pārbaudījuma komisijas sēdē

___.06.2017. prot. Nr. _____

Komisijas sekretārs: _____