TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Tarkvarateaduse instituut

Kristjan Peterson 155378IAPB

BATCH PROTSESSIDE PARALLELISEERIMINE ÜLE MITME JAVA VIRTUAALMASINA RABBITMQ ABIL

Bakalaureusetöö

Juhendaja: Evelin Halling
MSc

Tallinn 2018

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

Autor: Kristjan Peterson

18.05.2018

Annotatsioon

Antud lõputöö eesmärgiks on Eesti kapitalil põhineva panga öise päevavahetusprotsessi kõige aeglasemat osa kiirendada, pannes see tööle paralleelselt mitmes lõimes ning mitmes Java virtuaalmasinas.

Töös võrreldakse erinevaid meetodeid, mille abil saaks protsessi kiirendada ning võrreldakse erinevaid raamistikke, mille abil on võimalik protsesse paralleliseerida.

Töö käigus realiseeritakse paralleliseeritud protsess, mis ühes Java virtuaalmasinas on 2,5 korda kiirem kui vana protsess, kuid mis töötab ka mitmes Java virtuaalmasinas paralleelselt. Lahendus kasutab RabbitMQ raamistikku ning tehtud tarkvaraarendused on kaetud ka mõistlikus koguses automaattestitega.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 18 leheküljel, 5 peatükki, 6 joonist, 0 tabelit.

Abstract

Batch process parallelisation over multiple Java virtual machines using RabbitMQ

The purpose of this thesis is to optimize the slowest part of a bank's batch process, by making it run in multiple threads in multiple Java virtual machines.

In the thesis the author has described several ways which the process could be made faster, which include buying better hardware, optimising the process internally, making the process run in multiple threads and making it run in multiple threads in multiple Java virtual machines.

The author has also compared several frameworks which could be used for the parallelisation, bringing out the advantages and disadvantages of each one in the given use case. The frameworks that are compared are Spring TaskExecutor, Quartz, RabbitMQ and Obsidian.

Then the author has shown how the parallelisation was done using RabbitMQ and Spring boot. The solution is also covered in automated tests in a resonable amount.

Lastly the author has analyzed the new solution, comparing it to the old process, which ran in one thread in one Java virtual machine. In one Java virtual machine the new process was 2.5 times faster than the old one.

The thesis is in estonian and contains 18 pages of text, 5 chapters, 6 figures, 0 tables.

Lühendite ja mõistete sõnastik

SQL	Andmebaasi päringukeel
Git	Versioonihaldus süsteem
REST	*Representational State Transfer*
API	*application programming interface*

Sisukord

Jooniste loetelu

1 Sissejuhatus

Hetkel Eesti kapitalil põhineva panga olemasolev öine automaatne batch protsess pangapäeva vahetuse toimingute teostamiseks (nt. Laenulepingute igapäevane intressiarvutus, maksegraafikute muudatused jne.), jookseb ühes lõimes ning ühes java virtuaalmasinas, mis teeb protsessi aeglaseks. Hetkel võtab protsess kõige rohkem aega umbes kaks tundi, kuid andmemahtude kasvades ning süsteemi laienedes uutesse riikidesse ei pruugi protsess enam ööga valmis saada, mis tähendab panga töö paljude toimingute seiskumist. Kõige aeglasemaks protsessiks on laenude intressiarvutus, mis võtab keskmiselt umbes poole kogu protsessi ajast.

Antud lõputöö eesmärgiks on kiirendada öise päevavahetusprotsessi kõige aeglasemat osa. Optimiseeritud protsess peab olema vähemalt kaks korda kiirem, kui eelnev protsess.

Lõputöös arutlen erinevate võimaluste üle, kuidas protsessi kiirendada, võrdlen erinevaid raamistikke, mille abil panna protsess paralleelselt tööle ning kirjeldan, kuidas RabbitMQ abil protsess paralleelselt jooksma panna mitme Java virtuaalmasina peal. Samuti räägin automaattestide koostamisest, et tagada lahenduse töökindlus ka tulevikus. Töös mainin ka ära võimalikud tulevikuarendused.

Töö käigus loodav tarkvara arendus peab olema töökindel ning kergesti edasiarendatav. Lahendus ei tohi projekti juurde tekitada tehnilist võlga, selle vältimiseks peab võimalikult suure osa funktsionaalsusest katma automaattestidega.

2 Ülevaade

Panga rakendus põhineb mikroteenuste arhitektuuril, mis kujutab endast mitmetest eraldiseisvatest Springi rakendustest koosnevat tervikut, mis suhtlevad omavahel REST (*Representational State Transfer*) teenuste abil. Enamus rakendustest jooksevad kahes java virtuaalmasinas, et tagada suurem töökindlus juhul kui üks masin peaks töö lõpetama ning et tagada süsteemi lakkamatu töötamine süsteemi uuenduste puhul.

Süsteem põhineb mikroteenustel mitmetel põhjustel, nagu näiteks kergem hallatavus, sest erinevad tiimid saavad sama tarkvara kallal töötada mugavamalt, arendades funktsionaalsusi ainult endale vajaminevates moodulites ning teisi mooduleid mitte muutes. Samuti võimaldab antud lähenemine süsteemi erinevaid osasid erinevatel aegadel klientidele kättesaadavaks teha, lihtsustades kogu uuendamisprotsessi.

Öist batch protsessi alustab automaatselt üks teenus, mis paneb käima Camunda protsessi, mis kutsub omakorda välja teistes rakendustes Camunda protsesse. Camunda BPM on töövoogude ja otsustuste automatiseerimise raamistik [1]. Camunda võimaldab automatiseeritud protsesse arusaadavas vormis esitada ning jooksutada, protsessid kasutavad REST APIt (*application programming interface*) programmikoodiga suhtlemiseks [1].

2.1 Võimalused protsessi kiirendamiseks

Protsessi kiirendamiseks on võimalik kasutada erinevaid strateegiaid. Igal lähenemisel on omad plussid ja miinused, mille põhjal otsustada, milline lahendus realiseerida. Mõni lahendus on näiteks üsna ajakulukas, kuid kiirendab protsessi suuresti, mõni ei võta väga kaua aega, kuid on kallis.

Üks kõige primitiimsemaid lähenemisi probleemile oleks osta võimsamad arvutid, mille peal programm jookseb. Antud lahendusel on aga ilmselged negatiivsed küljed, milleks on suur maksumus. Kiiremate arvutite ostmine on aga asi mida saab teha võrdlemisi kiirelt, võrreldes keerulise tarkvaraarendusega. Praegune protsess, aga ei kasuta serverite jõudlust ära eriti effektiivselt, kasutades ainult umbes 10 protsenti protsessori jõudlusest.

Teine märksa keerulisem lähenemine oleks protsess sisemiselt optimiseerida, jooksutades seda endiselt ühes lõimes ning ühes Java virtuaalmasinas. Selline lähenemine on tarkvara arenduse aspektist üsnagi keeruline. Antud lahendusega võrreldes kergem variant oleks panna protsess jooksma mitmes lõimes, kuid ühes java virtuaalmasinas, selline lahendus oleks ilmselt arenduse aspektist üks kiiremaid, kuid ei tooks kõige suuremat ajalist võitu protsessi jooksutamisel.

Ilmselt kõige mõistlikum lahendus oleks panna protsess jooksma mitmes Java virtuaalmasinas paljudes lõimedes. Selline lahendus kasutaks maksimaalselt ära riistvara jõudlust ning kiirendaks protsessi ilma uut riistvara ostmata päris palju. Tarkvaraarenduse aspektist on lahendus siiski üsna keeruline. Antud töö raames otsustasin protsessi antud meetodil optimiseerida, sest see võimaldab maksimaalselt kasutada ära masinate riistvara. Võrreldes eeltoodud variantidega on seda küll realiseerida raskem, kuid võimaldab ilma riistvaraliste uuendustega kiirendada protsessi mitmekordselt.

2.2 Eesmärk

Minu ülesandeks on kõige aeganõudvam protsess panna jooksma mitmes lõimes ja mitmes java virtuaalmasinas. Lahendus peab suutma teha vajalikud sammud mingi protsessi alustamiseks ära ühe RESTi päringu raames, ehk antud juhul 60 sekundi jooksul, see tagaks kiirema tagasiside, juhul kui protsessi alustamiseks vajaminevate protsesside täitmisel tekib probleeme. Kõige olulisem asi, mille võiks jõuda RESTi päringu raames ära teha oleks andmete lugemine andmebaasist ning mingil kujul töötlemine, mis võimaldaks nende põhjal edasist tööd.

Lõputöö eesmärgiks on kiirendada protsessi keskmist töökiirust vähemalt kahekordselt, see peaks tagama protsessi piisavalt kiire töötamise ka süsteemi kasutusele võtmisel teistes riikides. Protsessi kiirendamine mugavdab ka igapäevast tarkvara arendust suuresti, lubades arendajatel ja testijadel testimiseks mõeldud keskkondades päevavahetusprotsessi kiiresti jooksutada, et testida vajaminevaid funktsionaalsusi kiiremini.

2.3 Võimalikud raamistikud

Antud ülesande täitmiseks on palju erinevaid võimalusi, näiteks ise implementeerida vajalik lõimede funktsionaalsus, kasutada mingit raamistikku või kasutada pilveteenuseid.

Ise kõigi vajalike funktsionaalsuste implementeerimine oleks väga ajakulukas ja keeruline, kuid tagaks selle, et kõik on tehtud vastavalt enda spetsiifilistele vajadustele, ilma et projekti oleks lisatud juurde uusi kolmanda osapoole raamistikke.

Raamistiku kasutamine lihtsustaks oluliselt erinevate lõimede loomist, haldamist ning protsessi jooksutamist mitmes java virtuaalmasinas. Raamistikega kaasneb aga nende tundma õppimine, projekti lisamine ja konfigureerimine. Arvestama peab ka sellega, kui suured on nende litsentsikulud.

On olemas ka erinevaid pilveteenuseid, mis võimaldavad jooksutada kolmanda osapoole serverites praktiliselt ükskõik mis koodi [2], [3]. Pilveteenused skaleerivad koodi automaatselt vastavalt vajadustele ning maksma peab ainult kasutatud serveri aja eest [2], [3]. Pank aga ei taha enda koodi jooksutada kolmanda osapoole serverites, eriti kui kood tegutseb delikaatsete isikuandmetega, ehk pilveteenused on antud probleemi lahendamiseks välistatud.

Ülesande täitmiseks vajaminevat funktsionaalsust sisaldavad paljud raamistikud, mis erinevad üksteisest paljude aspektide poolest, näiteks on mõned raamistikud tasulised: Flux Scheduler [4] ja Obsidian [5] , osad on aga tasuta kasutatavad: Quartz [6] ja RabbitMQ [7].

Sobilik raamistik peab sisaldama vaja minevat funktsionaalsust batch protsesside paralleliseerimiseks. Samuti peab raamistiku valimisega arvestama ajakuluga mis kaasneb selle tundma õppimisega, projekti lisamise ja konfigureerimisega.

2.3.1 Spring TaskExecutor

Spring sisaldab versioonist 2.0 endas funktsionaalsust, mis lubab kergesti tegeleda *thread poolidega* TaskExecutor liidese abil. See liides võimaldab järjekorda lisada vastavalt vajadusele *task*'e, ning TaskExecutor tegeleb nendega vastavalt sellele, kuidas see on seadistatud [8].

TaskExecutor ei ole väga konfigureeritav, aga see võimaldab siiski määrata ära selle, mitmes lõimes vaikimisi *taske* jooksutatakse ning mitmes lõimes neid maksimaalselt jooksutatakse. Samuti saab seadistada seda, kui palju *taske* saab järjekorda panna.

Kui üritada vajaminevaid batch protsesse TaskExecutori abil paralleliseerida oleks sellel mitmeid eeliseid teiste raamistike ees. Nimelt ei peaks projekti lisama ühtegi uut raamistikku. Samuti on TaskExecutorit lihtne kasutada, mis tähendaks, et vähe aega kuluks selle tundma õppimise peale, kuid keeruline realiseerida mahukamat funktsionaalsust.

Taskexecutoril on ka probleemsemaid külgi. Näiteks on see väga vähe konfigureeritav. Samuti oleks selle abil väga keeruline paralleliseerida batch protsesse mitme java virtuaalmasina vahel.

Suurim probleem Springi TaskExecutori kasutamise juures on aga see, et antud hetkel ei ole projektis veel kasutusel Springi versioon 2.0, ehk et selle funktsionaalsuse kasutamiseks oleks vaja uuendada projektis Springi versiooni, mis on aga üsnagi keerukas ning ajakulukas.

2.3.2 Quartz

Quartz on avatud lähtekoodiga laialdaselt konfigureeritav *job scheduling* raamistik [6]. See võimaldab valitud ajal või mingi ajendi peale ülesandeid käivitada. Quartz ei ole mõeldud *job queue*'de loomiseks, kuid selle jaoks on seda võimalik kasutada.

Quartzi abil saab ka *taske* hallata ning käivitada neid erinevates lõimedes. Selle konfiguratsiooni seadetes saab valida, mitmes erinevas lõimes koodi jooksutatakse.

Quartzi eelised seisnevad selles et see on väga laialdaselt konfigureeritav, mis võimaldab teha kõigile vajadustele vastavaid lahendusi. Quartz vastutab ka ise selle eest, et ükski *task* kaotsi ei läheks, isegi siis kui programm mingil põhjusel vahepeal sulgub, kasutades selleks andmebaasi, mis ei ole iseenesest iga protsessi puhul hädavajalik. Samuti võimaldab see kergelt aru saada millal kõik *taskid* on tehtud, mida on antud protsessi läbimiseks kindlasti vaja, sest kui intresside arvutamise protsess on läbi, hakkavad tööle järgmised protsessid. Quartz on ka avatud lähtekoodiga ning seda on ka võmalik ise oma vajadustele vastavalt muuta. Antud raamistiku abil on ka lihtne implementeerida selline lahendus, mis töötab mitmes java virtuaalmasinas.

Quartzi probleemid seisnevad selles, et projekti peab lisama uue raamistiku, mis lisab projekti keerukust. Samuti on antud raamistikku väga keeruline kasutada, kõigi võimaluste tõttu, mida see endas sisaldab. Seetõttu on üsna ebaotstarbekas lisada projekti juurde nii suur raamistik batch protsesside paralleliseerimiseks kuna antud projekti puhul kasutataks ainult väikest osa Quartz raamistiku funktsionaalsusest.

2.3.3 RabbitMQ

RabbitMQ on avatud lähtekoodiga *message broker*. Selle peamiseks funktsiooniks on sõnumite vahendamine erinevate rakenduste vahel, kuid see võimaldab aga ka tekitada *work* järjekordasid, ning kuulajaid, mis võtavad järjekorrast mitmes lõimes järjest andmeid. RabbitMQ sisaldab endas ka mugavat kasutajaliidest, mida saab brauseriga kasutada, mille abil saab jälgida erinevaid järjekordi ja kui palju on neis objekte ning kui kiirelt neid sinna pannakse ja võetakse. Kasutajaliidese kaudu saab ka mugavalt luua järjekordasid [7].

RabbitMQ eelised antud rakenduses Batch protsesside paralleliseerimiseks seisnevad selles, et see on rakenduses hetkel juba olemas, ehk uut raamistikku ei pea projekti hakkama lisama ja tundma õppima. RabbitMQ jaoks on panga projektis eraldi moodul, mis kontrollib selle tööd. Samuti on see protsesside paralleliseerimiseks hästi konfigureeritav, raamistik võimaldab hallata seda mitmes lõimes *taske* jooksutatakse ning kuidas neid järjekorda lisatakse, näiteks et kas salvestatakse kõvakettale või jäetakse ainult muutmällu. RabbitMQ abil on ka lihtne teha lahendus, mis töötaks mitmes java virtuaalmasinas.

RabbitMQ negatiivseks aspektiks on see, et objektide järjekorda panemine ei toimu väga kiirelt, eriti kui on soov need kõvakettale salvestada, kuid see pole niikuinii protsessi kõige ajakulukam osa. Ilmselt jõuab vajaliku hulga andmeid panna järjekorda siiski 60 sekundi jooksul.

2.3.4 Obsidan

Obsidian on java põhine *job-scheduler*, mis sarnaneb oma olemuselt suuresti Quartziga, kuid ei ole avatud lähtekoodiga ning on tasuline. Obsidian koosneb kahest osast, milleks on *scheduler service* ning veebipõhine kasutajaliides [9].

Scheduler service võimaldabki käivitada *taske* valitud aegadel või valitud ajendite peale. Obsidian võimaldab hästi paralleliseerida ülesandeid, luues mitu *scheduler*i ning neis paralleelselt koodi jooksutada, konfigureerida neid ühest tsentraalsest kasutajaliidesest, mis võimaldab konfigureerida ka Obsidiani REST APIt.

Obsidiani eelised batch protsesside paralleliseerimises seisnevad lihtsas konfigureerimises ja Springiga integratsioonis, lihtsas tehtud ülesannete salvestamises ning heas klienditoes. Klienditugi on tingitud sellest, et teenusel on litsentsitasu.

Obsidiani suurimaks probleemiks ongi aga see, et see on tasuline, sest batch protsesside paralleliseerimiseks on võimalik kasutada ka tasuta alternatiive, nagu Quartz. Obsidian ei võimalda ka luua enda vajadustele kohandatud kuulajaid, mis on antud ülesande jaoks ilmselt oluline.

2.4 Valitud raamistik

Antud ülesande täitmiseks valisin allolevaks tehnoloogiaks RabbitMQ. Selle raamistiku valisin ma seetõttu, et see on hetkel projektis juba olemas, mis tähendab et vähem kulub aega selle projekti lisamisega, seadistamisega ning tundma õppimisega. Samuti on sellega mugav luua *taskide* salvestamiseks järjekord ning luua järjekorrast lugejad, mis töötavad erinevates lõimedes ja erinevates java virtuaalmasinates, kuna mõlemad virtuaalmasinad kasutavad ühte ja sama järjekorda.

RabbitMQ kasutades peab ilmselt ise implementeerima selle, et millal on kõik vajalikud *taskid* tehtud, kuid see ei ole ilmselt nii suur ajakulu võrreldes mingi uue raamistiku kasutusele võtmisega ja tundma õppimisega.

2.5 Arenduseks kasutatud tarkvara

Arendamisel oli põhiliseks tööriistaks Intellij IDEA, mis on java arenduse tööriist, mis on loodud ettevõtte JetBrains poolt [10], mida on võimalik kasutada veel paljude erinevate programmeerimiskeelte jaoks, nagu näiteks JavaScript, tööriist saab ka aru Springi süntaksist. Intellij IDEA sisaldab endas ka andmebaasi tööriistu, mis võimaldavad vaadata andmebaase ning nende põhjal otse käske käivitada, lisatud on funktsionaalsus kõigi suuremate andmebaasi süsteemide haldamiseks, nagu Oracle, PostgreSQL, MySQL ning SQL Server [11]. Samuti on integreeritud giti implementatsioon, mis lubab kasutada

praktiliselt kõiki giti funktsionaalsusi. Intellj IDEA abil on võimalik ka projekti ehitada, sisseehitatud gradle'i funktsionaalsuse abil. Gradle on avatud lähtekoodiga projektide ehitamise automatiseerimise süsteem, mis on disainitud suurte, mitmeid projekte sisaldavate, programmide ehitamiseks [12]. Intellij IDEA programmi on võimalik lisada veel lisafunktsionaalsusi, installides selleks sisseehitatud funktsionaalsuse abil pluginaid, mille abil saab lisada näiteks tuge erinevatele programmeerimiskeeltele või raamistikele [13].

Andmebaasi käskude käivitamiseks on kasutusel MyBatis raamistik, mis võimaldab kirjutada kohandatud SQLi XML failidesse, mida saab java koodis mugavalt käivitada [14]. Intellij IDEA saab aru ka XMLi süntaksist. MyBatise lihtsamaks kasutamiseks pidin Intellij IDEAsse alla laadima MyBatise plugina, mis võimaldab lihtsamat MyBatise kasutust. Samuti pidin alla laadima Lomboki plugina, sest muidu ei saa Intellij IDEA aru Lomboki annotatsioonidest. Lombok on java raamistik, mis võimaldab java andmeklasside loomist kergendada, pannes klassi peale vastavad annotatsioonid. Lombok koostab ise automaatselt väljadele *getter* ja *setter* funktsioonid ja muud java funktsioonid, objektide võrdlemiseks ja logimiseks. Versioonihalduseks kasutasin giti.

3 Lahendus

Tehtud lahendus koosneb neljast põhilisest komponendist: ülesannete järjekorda andmete panemisest, kuulajate poolt andmete lugemisest järjekorrast, andmete põhjal protsesside jooksutamisest ning automaattestidest.

3.1 Järjekord

Esmalt, et andmeid saaks lisada kuhugi järjekorda, on vaja luua järjekord. Järjekorra loomiseks pidin RabbitMQ kasutajaliidese abil selle käsitsi looma. Järjekorra loomisel pidi sellele määrama nime ning konfigureerima veel mõnda asja, millest kõige olulisem oli see, et kas andmed, mis järjekorda lisatakse salvestatakse ka kõvakettale.

Andmed lisatakse järjekorda siis, kui programmi tuleb vastab RESTi päring, mille peale võetakse andmebaasist vastavad laenude unikaalsed identifikaatorid ning lisatakse need järjekorda.

Andmete järjekorda lisamisel peab arvestama paljude asjadega. Esmalt sellega, mis kujul andmed järjekorda panna. Mina lõin andmete järjekorda panemiseks java andmeobjekti, mis sisaldab mingi protsessi käivitamiseks unikaalset identifikaatorit ja pangakuupäeva. Samuti peab sõnumi lisamisel panema sinna ka muud vajalikud andmed, nagu organisatsiooni kood, kasutaja, kellaaeg ning sõnumi käsitlusmeetod. Need andmed asuvad kõik sõnumi päises.

Sõnumi käsitlusemeetodi abil saab määrata selle, kas andmed salvestatakse kõvakettale või mitte, antud juhul otsustasin seda parema jõudluse tõttu mitte teha. Jõudlus on antud olukorras oluline seetõttu, sest kõikide vajalike andmete järjekorda panemine võiks toimuda ühe RESTi päringu käigus, mille maksimaalne pikkus on antud projektis 60 sekundit. See tagab selle, et kui andmete järjekorda panemise käigus tekib kuskil viga saab sellest koheselt teada, see kohene teadlikkus protsessi ebaõnnestumisest on kasulik Camunda protsessi optimeerimiseks.

Selleks, et kuulajad saaksid teada, et kõik vajalikud andmed on läbi töödeldud, lisan ma pärast kõigi andmete järjerkorda lisamist sinna veel ühe elemendi, milleks on spetsiaalne java andmeklass, mis sisaldab protsessi nime. See kuulaja kes saab selle sõnumi, kus on sees antud andmeobjekt teab, et peab saatma sõnumi, et kõik vajalikud andmed antud

batch protsessi käigus on töödeldud. Antud lahendus, protsessi lõpetamise käsitlemiseks, tähendab seda, et kui saadetakse sõnum, et kõik andmed on töödeldud, siis teistes lõimedes võib töö alles käia, kuid nende andmetega pole uuesti vaja koheselt tegeleda, ehk selline lahendus on aktsepteeritav.

```
this.rabbitTemplate.convertAndSend("", routingKey, dto, (message) -> {
	message.getMessageProperties().setDeliveryMode(
		MessageDeliveryMode.NON_PERSISTENT);
	message.getMessageProperties().setHeader("org_code", orgCode);
	message.getMessageProperties().setHeader("user_name", user);
	message.getMessageProperties().setMessageId(componentId.toString());
	message.getMessageProperties().setTimestamp(new Date());
	return message;
});
```

Joonis 1 Sõnumi lisamine järjekorda

Joonisel on näidatud, kuidas andmeid järjekorda lisada. RabbitTemplate on klass, mille abil saab kergesti pöörduda RabbitMQ poole [15], esimene parameeter mis on tühi String on exchange'i nimi, tühja Stringi puhul kasutatakse vaikimisi exchange'i, routingKey on järjekorra nimi, mis on defineeritud rakenduse seadetes, dto on muutuja, mis sisaldab vajalikke andmeid batch protsessi täitmiseks. Sõnumi käsitlusmeetod NON_PERSISTENT määrab selle, et sõnumit ei salvestata kõvakettale. Sõnumi päisesse lähevad muud üldised andmed, nagu organisatsiooni kood, kasutaja, sõnumi identifikaator ja aeg.

Sõnumi kõvakettale salvestamine muudab andmete järjekorda panemise oluliselt aeglasemaks. Lokaalses arenduskeskonnas kulus 50 000 elemendi järjekorda panemiseks, ilma kõvakettale salvestamata, keskmiselt 15.19 sekundit, kõvakettale salvestades aga 31.31 sekundit. Ilma kettale salvestama on sõnumite järjekorda lisamine järelikult umbes 2 korda kiirem. See kiirus võimaldab lisada järjekorda palju rohkem andmeid ühe RESTi päringu käigus.

3.2 Järjekorrast lugemine

Järjekorrast lugemise jaoks on vajalik luua kuulajad, mis võtavad järjekorrast järjest elemente, kui neid sinna tekib. Spring võimaldab kuulajaid lihtsalt luua annotatsiooni *@RabbitListener* abil, mis pannakse java klassi peale [16].

```
@RabbitListener(bindings =
@QueueBinding(value = @Queue(value = "${queue.name.interestCalculation}"),
              exchange = @Exchange(value = ""),
              key = "${key.loan}"),
              containerFactory = "rabbitListenerContainerFactoryBatch")
```

Joonis 2 Järjekorrast andmete lugemiseks vajalik annotatsioon

Joonisel on näha, millise Springi annotatsiooni abil on võimalik luua järjekorra kuulaja. Bindings näidab seda, millega loodud kuulaja siduda, ehk antud juhul järjekorraga, mille jaoks on annotatsioon @QueueBinding, mille parameetriks on @Queue, mille parameetriks on järjekorra nimi, mis loetakse aplikatsiooni seadetest. Exchange määrab ära, millise exchange'iga on tegu, antud juhul kasutatakse vaikimisi väärust, containerFactory määrab selle, millist Springi beani kasutatakse kuulaja loomiseks.

Kuulajate loomiseks on vaja luua Springi *bean*, mis tagastab *factory*, mis lubaks luua kuulajaid. Springi *beani* loomiseks kasutatakse annotatsiooni *@Bean*, mida saab panna nii klassi kui ka funktsiooni peale [17]. Antud juhul kasutan seda funktsiooni peal, sest seda *Beani* on vaja kasutada ainult korra ning see sisaldab ainult ühte funktsiooni. See *Bean* määrab ka seda, mitmes lõimes andmeid loetakse ja töödeldakse.

```
@Bean
public SimpleRabbitListenerContainerFactory
rabbitListenerContainerFactoryBatch
(ConnectionFactory connectionFactory,
SimpleRabbitListenerContainerFactoryConfigurer configurer,
MessageConverter messageConverter) {
       SimpleRabbitListenerContainerFactory factory =
       super.rabbitListenerContainerFactory(
              connectionFactory,
              configurer,
              messageConverter);
       factory.setConcurrentConsumers(numberOfParallelBatchWorkers);
       return factory;
}
```

Joonis 3 Beani loomine, mille abil saab andmeid lugeda

Koodijupp asub klassis, mis laiendab klassi, mille abil saab luua RabbitListenerContainerFactory tüüpi objekte [18]. Muutuja numberOfParallelBatchWorkers määrab ära selle, mitu kuulajat luuakse, millest igaüks

on eraldi lõimes. Paralleelselt töötavate lõimede arvuks sai valitud 5, et mitte koormata serverit üle ning et lõimed ei peaks pidevalt ootama andmebaasi kasutusvõimalust, sest iga lõimes lukustatakse andmebaas sellega töötades.

Klassis, mis loeb sõnumeid, millel on annotatsioon @RabbitListener peab olema annotatsioon @RabbitHandler, mis määrab ära funktsiooni, kuhu jõuavad järjekorrast sõnumid [19]. Selle annotatsiooniga funktsioonis saab määrata ära, mis parameetritega sõnumeid see funktsioon loeb.

```
@RabbitHandler
public void process(
       @Header("org_code") String orgCode,
       @Header("user_name") String userName,
       @Header("amqp_receivedRoutingKey") String routingKey,
       @Header("amqp_messageId") String id,
       @Payload InterestComponentDto message) {
```

Joonis 4 Funktsiooni päis, järjekorra lugemiseks

Koodijupist on näha, mis muutujad jõuavad funktsioonini, mis tegeleb andmetega. InterestComponentDto on andmeklass, mis sisaldab panganduskuupäeva ja laenu identifikaatorit. @Header annotatsioon tähistab andmeid mis on sõnumi päises ning @Payload tähistab sõnumi sisu [20].

Funktsiooni päis, kuhu jõuab see sõnum, mis tähistab viimast sõnumit on peaaegu identne eelnevalt esitatud päisele, välja arvatud sõnumi sisu andmetüübile. Selle andmetüübi järgi jõuabki sõnum õigesse funktsiooni.

3.3 Protsesside käivitamine

Protsesside jooksutamisel peab arvestama mitmete faktoritega, millest üheks tähtsaimaks on transaktsionaalsus. Et säilitada andmete konsistentsus peavad kõik protsessid toimuma transaktsionaalselt, ehk kui midagi ebaõnnestub siis peab taastama olukorra, mis oli enne protsessi alustamist. Springis on sisseehitatud funktsionaalsus transaktsionaalsuse tagamiseks, mida saab kasutada kasutades klassil annotatsiooni @Transactional [21].
Kui kuulajasse saabub objekt, mis teavitab protsessi lõppemisest, tuleb sellest moodulit, mis protsessi alustas teavitada. Protsessi lõpetamise teavitamiseks peab lisama sellest kirje andmebaasi tabelisse, mida loeb RabbitMQ moodul, mis seejärel lisab kirje protsessi alustanud mooduli andmbaasi. Selle kirje peale tehakse Camundasse RESTi päring, ning Camunda asub järgmise protsessi sammu juurde.

3.4 Automaattestimine

Tehtud lahenduse töötamise tagamiseks lõin ma erinevat tüüpi automaatteste. Automaattestid tagavad projekti töökindluse ning selle, et uued arendused ei tee midagi juba olemasolevat katki. Testid tagavad ka olemasolevate asjade muutmise kergemaks, näidates arendajale mugavalt, kas midagi on muudatustega seoses katki läinud. Automaattestid käivitatakse antud projektis iga koodi giti laadimise korral automaatselt.

Kõige kõrgema taseme testideks on REST liidese testid, mis kontrollivad, kas liides töötab ja kontrollib sisendeid korrektselt. RESTi liidese testid ei kontrolli aga seda, kas kood, mis päringute peale käima pannakse töötab korrektselt. Sellist tüüpi testide nimetus on „*Smoke testing*“. Selliste testite mõte on pinnapealselt testida läbi suur osa rakendusest, et vaadata kas rakendus üldse töötab ning kas on mõtet hakata käivitama madalama taseme teste, mis testivad täpsemalt madalama taseme funktsionaalsust.

Liidese testimiseks tegin kaks automaattesti, millest esimeseks on test, mis tagab selle et liides saadab korrektsete sisendite korral tagasi eduka staatuse koodi 200. Kood 200 tähendab, et päring oli edukas ning vajalikud protsessid said tehtud [22]. Teiseks testiks tegin ma testi, mis kinnitab selle, et ebakorrektsete sisendandmete korral saadetakse koodis tagasi erind. See test tagab selle, et sisendandmed kontrollitakse õigesti üle.

Kuulaja testimiseks tegin samuti kaks testi. Testide eesmärgiks on olla kindel, et järjekorra kuulaja alustab õigete sisendite korral õigeid protsesse. Üks testidest kontrollib, kas kuulaja alustab korrektse sisendi korral õiget protsessi ning teine kontrollib, kas protsess lõpetatakse, kui sisse tulevad vastavad sisendid.

Samuti on vaja teha integratsiooni teste, mis kontrollivad, kas erinevad komponendid töötavad üksteisega koos korrektselt. Antud juhul olid protsesside kontrollimiseks vajalikud integratsiooni testid juba olemas. Järjekorda panemise ja lugemise automaattestimist hõlbustava funktsionaalsuse olemasole hetkel projektis kahjuks ei ole, sest need lisati Springi uuemast versioonist. Ise sellise funktsionaalsuse ehitamine, mis laseks automaatselt kontrollida järjekorda panemise ja lugemise funktsionaalsust tervikuna osutuks väga ajamahukaks ja keeruliseks. Tulevikus oleks kindlasti vaja koostada testid, mis kontrolliksid andmete järjekorda panemise ja lugemise protsessi terviklikumalt, kui on võimalik *unit* testidega. Antud ülesande raames oleks see kujunenud aga väga ajamahukaks, ehk selliste testide koostamine jääb tulevikku.

Kõige madalama taseme testimist tehakse *unit* testidega, mis kontrollibki üksikute komponentide korrektset töötamist. Vajaminevad protsessid olid juba kaetud *unit* testidega, ehk neid ei pidanud ma ise koostama.

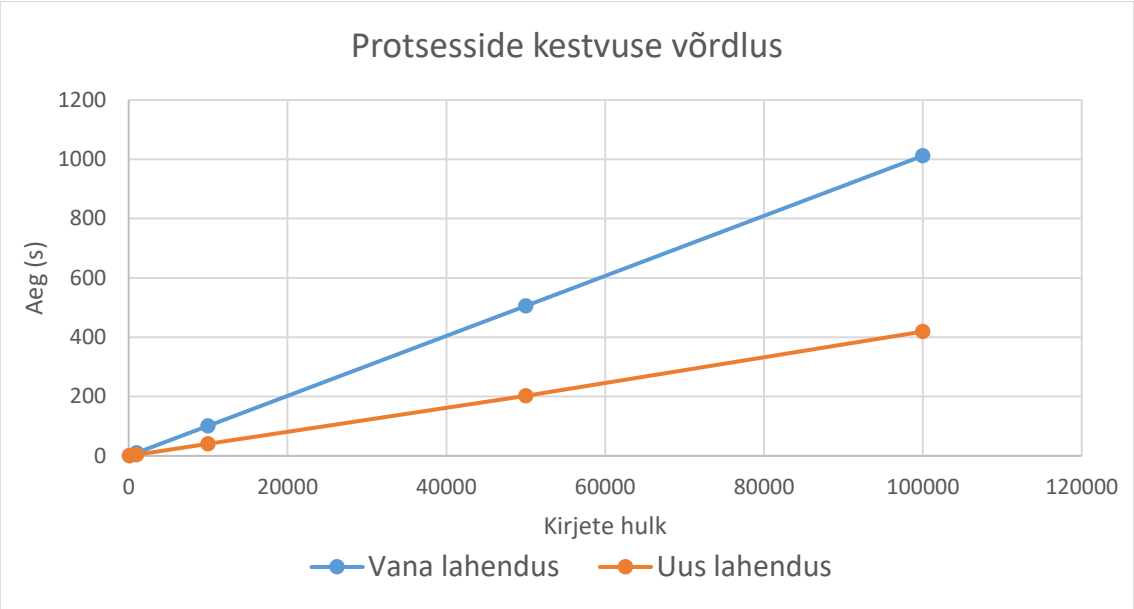
4 Lahenduse analüüs

Varem töötas *batch* protsess ühes java virtuaalmasinas ühes lõimes ning nüüd töötab see kahes masinas ning mõlemas neist viies lõimes, ehk kokku kümnes lõimes. Teoorias võiks see tähendada, et protsess on 10 korda kiirem, kuid tegelikkuses kulub paralleliseerimise tõttu aega muude asjade peale, mille peale varem ei kulunud. Näiteks peab uue lahendusega panema kõik vajaminevad andmed järjekorda, ning seejärel neid sealt lugema, mis on lisa funktsionaalsus, mida varem ei eksisteerinud.

4.1 Koormustesti tegemine

Tulemuse testimiseks tegin ma koormusteste erinevate andmemahtudega, et hinnata lahenduse effektiivsust erinevate andmemahtudega ning võrrelda seda eelneva lahendusega, mis töötas ühes lõimes. Koormusteste sain ma teha ainult enda arvutis, sest mul puuduvad andmebaasi õigused muudes keskkondades, kus rakendus töötab mitmes java virtuaalmasinas. Ehk koormustestide tulemused põhinevad ühe java virtuaalmasina põhjal.

Esmalt koormustesti läbimiseks lõin ma andmebaasi tabeli, kuhu salvestada testide tulemused. Andmebaasi salvestan ma millise protsessiga on tegu, kui suure andmemahuga on tegu, algusaja, lõpuaja ning kogu protsessi jaoks kulunud aja. Seejärel tegin REST API, kus on kaks funktsiooni, üks vana lahenduse testimiseks ning teine uue, kuhu mõlemasse saab anda ette kui palju kirjeid on vaja luua. Funktsioon loob esmalt sisendi põhjal vastava koguse andmeid, seejärel sisestab andmebaasi protsessi algusaja, protsessib andmed ning lõpuks sisestab andmebaasi protsessi lõpuaja ja arvutab kogu kestvuse.



Joonis 5 Protsesside kestvuse võrdlus

Joonisel on näha, kui palju kiirem on uus lahendus võrreldes eelnenud lahendusega. Joonise y-teljel on protsessi kestvus sekundites ning x-teljel on kirjete hulk, mis tuli läbi töödelda. Uus lahendus on ühes Java virtuaalmasinas vanas lahendusest keskmiselt 2,5 korda kiirem, välja arvatud väikeste andmemahtude korral, näiteks 100 kirje puhul oli paralleelselt töötav lahendus ainult 1,5 korda kiirem.

4.2 Tulemi vastavus nõuetele

RabbitMQ abil paralleliseeritud batch protsess on ühes Java virtuaalmasinas keskmiselt umbes 2,5 korda kiirem, kui ühes lõimes töötav lahendus. Kuna lahendus on juba ühes Java virtuaamasinas peaaegu kolm korda kiirem kui vana lahendus, võib õelda, et vajalikud eesmärgid kiiruse osas on kindlasti täidetud. Tulemused võivad serveris küll natukene erineda, kuid ilmselt mitte nii palju, et uus protsess ei oleks vähemalt kaks korda kiirem kui vana protsess.

4.3 Võimalikud arendused tulevikus

Antud töö raames tehtud lahendusel puuduvad integratsiooni testid, sest projektis oli töö tegemise hetkel liiga vana Springi versioon, mis ei võimaldanud koostada integratsiooni teste Springi uuemasse versiooni lisatud funktsionaalsuse abil. Tulevikus, pärast Springi versiooni uuendamist tuleks lahendusele kindlasti lisada integratsiooni testid, mis

testiksid lahendust tervikuna, lisades järjekorda andmeid, lugedes neid ning nende põhjal protsesse alustades.

Antud lahendus on võimalik ka võtta eeskujuks teiste protsesside kiirendamiseks. Enamus päevavahetus protsesse ei saa aga kahjuks sarnasel meetodil kiirendada, sest enamasti sõltuvad protsessid suuresti varasemalt läbi töödeldud andmetest, mis juhul võivad paralleliseerimisega tekkida vead, või ei tegele protsessid sarnaselt järjest andmete töötlemisega.

5 Kokkuvõte

Käesoleva lõputöö raames uuriti kuidas panga päevavahetusprotsessi kõige aeglasemat osa kiirendada. Arutleti erinevate võimaluste üle, kuidas seda teha võiks ning jõuti järeldusele, et panna protsess paralleelselt jooksma mitme Java virtuaalmasina peal on kõige optimaalsem variant.

Protsessi paralleliseerimiseks võrreldi erinevaid raamistikke, mille abil saaks ülesannet teostada, ning toodi esile nende positiivsed ja negatiivsed küljed. Lõpuks otsustati RabbitMQ kasuks.

Ühes masinas oli lahendus umbes 2,5 korda kiirem kui vana lahendus, ehk ülesande jaoks püstitatud eesmärgid on täidetud.

Protsessi paralleliseerimiseks pidi panema andmed järjekorda ning mitmes Java virtuaalmasinas, neid sealt lugema hakkama ja seejärel töötlema. Kood sai ka kaetud automaattestidega, mis vähendab tehnilise võla tekkimist.

Kasutatud kirjandus

[1] „Camunda koduleht,“ [Võrgumaterjal]. Available: https://camunda.com/. [Kasutatud 21 4 2018].

[2] „Google Cloud koduleht,“ [Võrgumaterjal]. Available: https://cloud.google.com/functions/features/. [Kasutatud 20 4 2018].

[3] „Amazon lambda koduleht,“ [Võrgumaterjal]. Available: https://aws.amazon.com/lambda/. [Kasutatud 20 4 2018].

[4] „Flux Scheduler koduleht,“ [Võrgumaterjal]. Available: https://flux.ly/. [Kasutatud 24 4 2018].

[5] „Obsidiani koduleht, litsentsid,“ [Võrgumaterjal]. Available: https://obsidianscheduler.com/licensing/. [Kasutatud 24 4 2018].

[6] „Quartzi koduleht,“ [Võrgumaterjal]. Available: http://www.quartz-scheduler.org/documentation/faq.html#FAQ-springHelp. [Kasutatud 20 4 2018].

[7] „RabbitMQ koduleht,“ [Võrgumaterjal]. Available: https://www.rabbitmq.com/. [Kasutatud 20 4 2018].

[8] „Springi dokumentatsioon,“ [Võrgumaterjal]. Available: https://docs.spring.io/spring/docs/4.2.x/spring-framework-reference/html/scheduling.html. [Kasutatud 20 4 2018].

[9] „Obsidiani dokumentatsioon,“ [Võrgumaterjal]. Available: http://obsidianscheduler.com/wiki/Deployment_Models. [Kasutatud 22 4 2018].

[10] „Intellij IDEA koduleht,“ [Võrgumaterjal]. Available: https://www.jetbrains.com/idea/. [Kasutatud 10 5 2018].

[11] „Intellij IDEA andmebaaside dokumentatsioon,“ [Võrgumaterjal]. Available: https://www.jetbrains.com/help/idea/relational-databases.html. [Kasutatud 10 5 2018].

[12] „Gradle'i koduleht,“ [Võrgumaterjal]. Available: https://gradle.org/. [Kasutatud 11 5 2018].

[13] „Intellij IDEA koduleht, pluginad,“ [Võrgumaterjal]. Available: https://www.jetbrains.com/help/idea/installing-updating-and-uninstalling-repository-plugins.html. [Kasutatud 24 4 2018].

[14] „MyBatis koduleht,“ [Võrgumaterjal]. Available: http://www.mybatis.org/mybatis-3/. [Kasutatud 10 5 2018].

[15] „Springi dokumentatsioon, RabbitTemplate,“ [Võrgumaterjal]. Available: https://docs.spring.io/spring-amqp/api/org/springframework/amqp/rabbit/core/RabbitTemplate.html. [Kasutatud 20 4 2018].

[16] „Springi dokumentatsioon, RabbitListener,“ [Võrgumaterjal]. Available: https://docs.spring.io/spring-amqp/api/org/springframework/amqp/rabbit/annotation/RabbitListener.html. [Kasutatud 20 4 2018].

[17] „Springi dokumentatsioon, Bean,“ [Võrgumaterjal]. Available: https://docs.spring.io/spring-javaconfig/docs/1.0.0.M4/reference/html/ch02s02.html. [Kasutatud 20 4 2018].

[18] „Springi dokumentatsioon, EnableRabbit,“ [Võrgumaterjal]. Available: https://docs.spring.io/spring-amqp/api/org/springframework/amqp/rabbit/annotation/EnableRabbit.html. [Kasutatud 20 4 2018].

[19] „Springi dokumentatsioon, RabbitHandler,“ [Võrgumaterjal]. Available: https://docs.spring.io/spring-amqp/api/org/springframework/amqp/rabbit/annotation/RabbitHandler.html. [Kasutatud 20 4 2018].

[20] „Springi dokumentatsioon, Messaging,“ [Võrgumaterjal]. Available: https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/messaging/handler/annotation/package-summary.html. [Kasutatud 21 4 2018].

[21] „Springi dokumentatsioon, Transactional,“ [Võrgumaterjal]. Available: https://docs.spring.io/spring/docs/4.2.x/spring-framework-reference/html/transaction.html. [Kasutatud 22 4 2018].

[22] „W3 koduleht, status codes,“ [Võrgumaterjal]. Available: https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html. [Kasutatud 22 4 2018].