TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Informaatikainstituut
Tarkvaratehnika õppetool

Monotoonsete süsteemide andmekaevealgoritmi paralleelne realisatsioon Apache Spark abil

Bakalaureusetöö

Üliõpilane: Silver Puulmann
Üliõpilaskood: 120819IABB
Juhendaja: Martin Rebane

Tallinn
2016

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.

(*kuupäev*) (*allkiri*)

Annotatsioon

Käesoleva töö eesmärgiks on tutvustada lugejale monotoonsete süteemide algoritmi realisatsiooni Apache Sparki abil. Apache Spark on paralleelarvutuste tehnoloogia, mis muudab arvutused kordades kiiremaks kui ilma klastrita lahendustes võimalik oleks. Apache Spark pakub mitmeid erinevaid võimalusi rakenduste kirjutamiseks nagu näiteks Java, Scala ja Python. Antud töös luuakse Sparki realiseeriv programm Javas. Lisaks praktilisele realisatsioonile tutvustatakse ka teoreetilisi tagamaid, millistel juhtudel Sparki kasutamisest abi oleks ja see efektiivsust tõstaks. Näitena tehaksegi läbi üks monotoonset süsteemi kirjeldav algoritm, mis leiab üles andmetes esinevad reeglipärasused.

Töös aluseks võetud monotoonset süsteemi kirjeldav algoritm on, nagu hiljem detailses selgituses lähemalt tutvustatakse, olemuselt tegelikult üsna lihtne. Alustuseks programmeerisin selle valmis kasutades Javat ilma Sparkita, et oleks olemas lähteülesanne mida hakata Sparkis implementeerima. Seejärel realiseerisin antud algoritmi kasutades Sparki Java APIt. Teise asjana lisaks algoritmi implementeerimisele analüüsiti kahe erineva andmestiku peal, kui palju efektiivsem Sparki kasutamine on.

Töös realiseeriti kahe programmina monotoonset süsteemi kirjeldav andmekaevealgoritm, üks neist Javas ja teine koos Sparki kasutamise toega. Mõlema programmi efektiivsust mõõdeti ajaliselt ja mälukasutuse poolest kahe erineva suurusega andmestikuga. Valitud algoritm ja andmestikud ei lubanud Sparkil näidata kogu võimekust, aga oli selgelt näha, et andmestiku kasvades toimetab Spark efektiivselt.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 41 leheküljel, 9 peatükki, 2 joonist ja 2 tabelit.

Abstract

The purpose of this thesis is to introduce Apache Spark to the readers – a new technology that promises to make cluster computing faster than ever before. Apache Spark offers different APIs for writing applications in, for example Java, Scala and Python. For this thesis Java API is used to implement an algorithm in Java. The reader is also introduced with the main cases where using Spark could be effective. For this a data mining algorithm describing a monotone system is put into practice firstly using Java and then adding the support of Spark.

The monotone system, as described later in this thesis, is actually quite simple by its principles. It was firstly realised in Java without using Spark. From there the support for Spark was added using Sparks Java API. Secondly the effecntiveness of Spark was analyzed using two datasets and comparing the results of multi-threading with one threaded operations.

As a result of the thesis two data mining algorithms for monotone systems were implemented. One of them written in Java and the second one adding the support for using Spark. The effectiveness for both programs was measured by time and by memory usage with two different datasets. It may be noted that the algorithm and given datasets did not give Spark full opportunity to show its true colors, however it was seen to be more effective with the bigger dataset.

The thesis is in estonian and contains 41 pages of text, 9 chapters, 2 figures and 2 tables.

Lühendite ja mõistete sõnastik

Mustrite sobitamine	***Pattern matching*** Andmestikus esinevate täpsete mustrite leidmine
Hadoop	Apache tarkvara raamistik, mis võimaldab suuri andmestikke klastrite vahel jagada ja neile paralleelarvutuseks osadena saata
RDD	*Resilient Distributed Dataset* Põhiline andmetüüp mida Spark kasutab. Elementide kogumik, mis on jagatud erinevate lõimede vahel paralleelseteks toiminguteks.
CSV	*Comma Separated Values* Failiformaat, mis sisaldab andmeid, mis on komadega eraldatud.

Jooniste nimekiri

Joonis 1. Lineaarset regressiooni kirjeldav sirge 14
Joonis 2. Sparki klasterdamise tööpõhimõte 18

Tabelite nimekiri

Tabel 1. Mälukasutuse võrdlus 26
Tabel 2. Ajakulu võrdlus 26

Sisukord

1. Sissejuhatus 10
1.1 Taust ja probleem 10
1.2 Ülesande püstitus 10
1.3 Metoodika 10
1.4 Ülevaade tööst 11
2. Lineaarne regressioon ja monotoonsete süsteemide algoritmid 12
2.1 Andmekaeve 12
2.2 Lineaarne regressioon 13
2.3 Monotoonsete süsteemide algoritmid 14
2.4 Implementeeritav algoritm 15
3. Spark'i ja klasterdamise tutvustus 16
3.1 Klasterdamise tutvustus 16
3.2 Klasterdamise eelised 16
3.3 Sparkist üldiselt 17
3.4 Sparki tööpõhimõte 18
4. Algoritmi muutmine paralleelkasutuseks 20
4.1 Sparki kasutamise eeldused 20
4.2 Sparki konfigureerimise võimalused 20
4.3 Algoritmi realiseerimine Sparkis 21
4.3.1 Sparki eripärad 21
4.4 Algoritmi realisatsioon 22
4.4.1 Algoritmi realiseerimine ilma Sparkita 22
4.4.2 Algoritmi realisatsioon Sparkis 23
5. Spark'i efektiivsuse analüüs 25
5.1 Testkeskkond 25
5.2 Testiandmestik 25
5.3 Mälukasutus 26
5.4 Ajakulu 26
6. Kokkuvõte 28
7. Summary 29

8. Kasutatud kirjandus 30
9. Lisad 31
9.1 Lisa 1: Andmekaevealgoritm Javas 31
9.2 Andmekaevealgoritm Sparkis 36

1. Sissejuhatus

Andmehulkadest reeglite leidmise üks põhiliseid probleeme on leida üles võimalikult huvitavad reeglid, mis andmetes esinevaid mustreid kõige paremini kirjeldavad. Reeglite leidmiseks on olemas palju erinevaid algoritme – millist algoritmi kasutada sõltub paljuski andmete liigist ja kasutamise eesmärgist. Välja töötatakse küll järjest tõhusamaid algoritme, kuid samas kasvavad ka andmemahud ning üheks heaks variandiks, kuidas reeglite leidmist kiirendada on algoritmi paralleelseks muutmine.

1.1 Taust ja probleem

Andmete analüüsimisel erinevate algoritmide kasutamine on väga levinud, leidmaks andmetes olevaid reeglipärasusi ja mustreid. Kuigi algoritmid iseenesest ei pruugi suurema süvenemise juures olla oma põhimõtetelt kuigi keerulised, teeb mustrite leidmise aeganõudvaks andmemahtude suurus. Algoritmide efektiivsemaks tegemine ja nende realiseerimine kasutades ajakohaseid tehnoloogilisi lähenemisi on küll üks edu võtmetest, samas teisalt võiks seda protsessi veelgi kiirendada kasutades klasterdamist. Üks uuemaid klasterdamise tehnoloogiaid on Spark, mida kasutatakse ka antud töös ühte monotoonset süsteemi kirjeldava algoritmi efektiivsemaks muutmiseks.

1.2 Ülesande püstitus

Realiseerida monotoonset süsteemi kirjeldav algoritm paralleelselt, et oleks võimalik kasutada Spark'i poolt kasutatavat klasterdamise võimalust.

Mõõta, kui palju efektiivsem on Sparki kasutamine võrreldes sama algoritmi jooksutamisega ilma klasterdamiseta.

1.3 Metoodika

Kasutades Apache Spark'i muuta algoritm paralleelselt kasutatavaks ja seejärel võrrelda erinevate andmete põhjal Spark'i poolt pakutava klasterdamise efektiivsust.

1.4 Ülevaade tööst

Esimeses osas kirjeldatakse lineaarset regressiooni ning monotoonseid süsteeme. Lineaarne regressioon on üks kõige rohkem kasutatavaid lähenemisi andmetes esinevaid mustreid kirjeldavate algoritmide leidmiseks. Monotoonsete süsteemide lähenemist omakorda saab kasutada leitud algoritmide efektiivsemaks muutmiseks.

Teises osas tutvustatakse klasterdamist, Spark'i üldiselt, Spark'i poolt pakutavaid võimalusi ja seda, millistest olukordades klasterdamine kõige rohkem efektiivsust tõsta aitab.

Kolmandas osas kirjeldatakse algoritmi paralleelseks muutmist, et seda oleks võimalik andmetest reeglite leidmiseks jooksutada kasutades Spark'i.

Neljandas osas on statistika, kui palju efektiivsemaks teeb reeglite leidmise Spark'i poolt pakutav klasterdamise võimalus kirjeldades ajakulu ja mälu kasutamist.

2. Lineaarne regressioon ja monotoonsete süsteemide algoritmid

Käesolevas peatükis tutvustatakse lugejale esmalt mis on andmekaeve ja millised on selle kasutusvaldkonnad. Andmekaeve efektiivseks teostamiseks on sellele vaja läheneda õigete vahenditega ja võrrelda tulemusi laialt levinud lahendustega. See toob meid järgmisena ühe laialdasemalt kasutatava matemaatilise analüüsi meetodini – lineaarse regressioonini. Monotoonseid süsteeme kirjeldav peatükk tutvustab lähemalt monotoonse algoritmi olemust ja põhimõtteid ning demonstreerib erinevusi lineaarse regressiooniga. Viimasena on antud peatükis juttu konkreetselt käesolevas töös realiseeritavast algoritmist.

2.1 Andmekaeve

Andmekaeveks (ing. k *data mining*) nimetatakse suurtest andmehulkadest kasuliku informatsiooni kättesaamist. Üheks väärtuslikumaks teabeks on andmetes esinevate mustrite leidmine (ing. k *pattern matching*) (Vreeken 2009). Mustrid omakorda on regulaarsused mis andmetes esinevad, näiteks millised geenid mõjutavad kõige rohkem konkreetse haiguse kulgu või missugune on kõige tüüpilisem ostukorv inimesel, kes on sinna lisaks kõigele muule pannud ühe Kalevi šokolaadi.

Mustrite leidmiseks on lisaks andmetele vaja teada, milline peab tagastatav muster välja nägema ning teiseks on vaja programmi, mis andmed läbi töötleb ja tegelikud mustrid välja annab (Vreeken 2009).

Andmetes esinevate mustrite kasulikkust on hea selgitada näiteks mõne haiguse esinemise põhjuste uurimise näite najal (*ibidem*). On olemas haigus A, mis arsti arvates sõltub kindlatest geenidest B ja C ning keskkonna teguritest D ja E. Isegi kui arstil on olemas patsiendist täielik ülevaade ning patsient märgib sajaprotsendiliselt üles, kus ta viibib ja mida ta sööb, ei pruugi arst ise andmeid läbi töötades teada saada, millised peavad antud tegurid haiguse esinemiseks olema. Õnneks on arstil olemas värskelt arendatud tarkvara, mis tegeleb just nimelt andmetest mustrite otsimisega ning püstitatud hüpoteeside kontrollimiseks ei pea ta tegema muud, kui tarkvara poolt tagastatud mustreid analüüsima, et leida milliste tegurite koosmõjul antud haigus avaldub (*ibidem).*

Andmekaevet ja mustrite leidmist on võimalik kasutada lisaks eelpool näitena toodud arstiteadusele kasutada ka ülikoolides tehtava teadustöö raames, eraettevõtetes müügitulemuste analüüsimiseks ja praktiliselt igal pool, kus töödeldavad andmemahud on niivõrd suured, et inimesed ise sellega hakkama ei saaks. Viimasel ajal on väga palju hakanud levima n-ö suure andmestiku (ing k *big data*) analüüsimine, sest tänu arenenud infotehnoloogilistele vahenditele on erinevate andmete kogumine saanud väga lihtsaks ja näiteks niivõrd lihtsat asja nagu ühel veebisaidil ühes konkreetses kohas tehtud hiireklikki analüüsides on võimalik teha hulgaliselt järeldusi nii kliendi, veebisaidi omaniku kui ka arendaja kohta.

2.2 Lineaarne regressioon

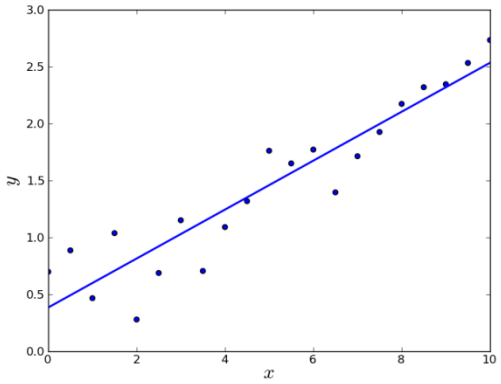
Regressioonanalüüs võimaldab luua lihtsasti hoomatava matemaatilise mudeli erinevate tunnuste vaheliste seoste kirjeldamiseks. Regressioonanalüüsi puhul vaadeldakse ühte tunnust kui teisest sõltuvat ning leitakse tunnuseid, mille põhjal sõltuva tunnuse väärtuseid kirjeldada saaks. Ühtlasi saab tehtud analüüsi põhjal prognoosida ka väärtusi, mis oletuslikult tekkida võivad (Möls 2012).

Lineaarne regressioon on kõige klassikalisem regressioonanalüüsi meetod, kus kaks muutujat (tunnust) on omavahelises sõltuvuses. Lineaarse regressioonivõrrandi üldkuju on järgnev:

$$Y= b_0 + b_1x_1 + b_2x_2 + K + b_nx_n$$

kus y on sõltuv tunnus ning x_1, K ja x_n on omavahel sõltumatud tunnused. Loomulikult on olemas ka keerulisemaid lineaarset regressiooni kirjeldavaid võrrandeid, kus on sees rohkem tunnuseid ning sisse saab tuua ka erinevate tunnuste omavahelise sõltuvuse. (Niglas 2013)

Lineaarset regressiooni on kõige lihtsam selgitada joonise abil, kus tunnustevahelised seosed on kantud xy-graafikule kordajate abil, kasutades vähimruutude meetodit. Vähimruutude meetodi idee on, et seost iseloomustavat punktiparve esindab selline sirge, mille kõikide üksikpunktide kauguste ruutude summa sellest sirgest on minimaalne.



Joonis 1. Lineaarset regressiooni kirjeldav sirge

2.3 Monotoonsete süsteemide algoritmid

Monotoonsete süsteemide teooria töötati Tallinna Tehnikaülikoolis välja mitukümmend aastat tagasi Leo Võhandu ja Joosep Mullati poolt. Monotoonsete süsteemide meetodi võib pidada algoritmide loomise metoodikaks. (Aab 2002)

Monotoonsed süsteemid koosnevad kahest osast (*ibidem*) :

1) Töödeldavate elementide hulk

2) Monotoonsuse tingimusele vastav kaalufunktsioon.

Monotoonsuse tingimustele vastab kaalufunktsioon, kui see annab väärtuse igale üksikelemendile suvalise alamhulga koral ning hulkade muutumisel muutub kaalufunktsiooni poolt antav väärtus olenevalt kaalufunktsioonist kas ainult väiksemaks või ainult suuremaks. Erandina võib väärtus alati jääda ka samaks – sellisel juhul on tegemist nõrga monotoonsusega. (Aab 2002)

Monotoonse süsteemi tööpõhimõtet aitab kirjeldada järgnev näide:

Meil on elementide hulk A, mis on lähtehulgaks, elementide hulk B, mis on lõpphulgaks (kusjuures hulk B on esialgu tühi) ning kaalufunktsioon K, mis vastab monotoonse süsteemi tingimustele.

Monotoonne süsteem töötab Aabi kirjelduse kohaselt järgmiselt:

1. Igale elemendile hulgas A leitakse funktsiooni K väärtus.

2. Hulgast A valitakse element, millele arvutati kas kõige suurem või kõige väiksem kaal (vastavalt sellele, millist lähenemist antud süsteemi puhul kasutatakse).

3. Antud element eemaldatakse hulgast A ja liidetakse lõpphulka B.

4. Kui hulka A kuulub veel elemente, alustatakse uuesti punktist number 1.

5. Protsessi lõpp (ehk hulk A on tühi ja hulk B sisaldab kõki hulgas A olnud elemente).

Iga elemendi liigutamisel hulgast A hulka B muutub nende hulkade kooslus, mis mõjutab kaalufunktsiooni tulemust. Jälgides ükshaaval ja samm-sammult, kuidas iga elemendi tõstmisel hulgast A hulka B kaalufunktsiooni tulemused muutuvad saab elementide kohta teha vastavaid järeldusi ja seeläbi töötada välja tõhusamaid algoritme. (Aab 2002)

2.4 Implementeeritav algoritm

Antud töös realiseeritav algoritm on kasutatav igas monotoonses süsteemis leidmaks esimese taseme reegleid. Algoritm võtab sisendiks binaarandmed ning tagastab iga veeru kohta reegli kui neid peaks leiduma. Järgnevalt on esitatud algoritmi töötamise põhimõte:

1. Valitakse välja veerg, mille kohta teadmisi kogutakse, seda nimetame edaspidi klassiveeruks.

2. Andmestiku kohta koostatakse sagedustabel – iga veeru kohta tuuakse välja selles esinev ühtede arv, kusjuures ühtede arv on mõõduks, mis monotoonselt ühes suunas liigub kui andmestikku muuta.

3. Andmestik jagatakse kaheks klassiveeru järgi: ühte osasse jäävad andmeread, kus klassiveerg on väärtusega 0 ning teise osasse read, kus klassiveerg on väärtusega 1.

4. Leitakse sagedustabel ka klassiveeru järgi koostatud alamhulkade kohta.

5. Võrreldakse alamhulga ühtede sagedust ja kogu andmestiku ühtede sagedust. Kui need sagedused kattuvad, oleme järelikult leidnud tunnuse, mis 1. punktis valitud klassiveergu iseloomustab.

Kui antud protsess käia läbi kogu andmestikuga ehk võtta igat veergu andmestikus ühe korra klassiveeruna, leiamegi kõik antud süsteemi iseloomustavad reeglid.

3. Spark'i ja klasterdamise tutvustus

Käesolevas peatükis tuleb lähemalt juttu klasterdamisest. Tutvustatakse selle lähenemise eeliseid võrreldes alternatiivsete lähenemistega ning kirjeldatakse põgusalt Sparki evolutsiooni. Samuti tuleb juttu, milliseid võimalusi Spark pakub, mis on Sparki tööpõhimõte ning mispärast on Sparki kasutamine viimasel ajal järjest populaarsemaks muutumas.

3.1 Klasterdamise tutvustus

Arvutiklastreid saab jagada erinevate tööülesannete järgi enamasti kaheks: töökindlus- ja arvutusklastriteks. Arvatavalt võisid esimesed klastrid tekkida juba hilistel 50'datel, kui andmemahud kasvasid ning kogu töö ei mahtunud ära ühte arvutisse või oli vajadus sellest varukoopiad teha. Sarnane süsteem oli tol ajal kasutuses USA õhujõududes, kus IBM-i ja MIT poolt loodud kahearvutilises klastris oli võimalik ühe riknemise korral ülesanded teisele arvutile üle anda.

Kuna klasterdamine sõltub paljuski arvutite omavahelisest ühendusest, on selle areng tihedalt seotud ka arvutivõrgu arenemisega. 80'datel hakati järjest rohkem uurima klastrite kasutamise võimalusi just arvutusvõimsuse tõstmiseks ning 1989. aastal arendati välja Paralleelse Virtuaalmasina (PVM) tarkvara, mis koordineeris andmeedastust erinevate klastrisse kuuluvate arvutite vahel.

Arvutusklastrid on tänapäeval laialt levinud näiteks ülikoolides, kuna need võimaldavad suhteliselt soodsalt luua piisava arvutusvõimsuse keerukate ja palju jõudlust nõudvate arvutuste jaoks. Samas kasutatakse näiteks töökindlusklastreid pea kõigi tähtsamate süsteemide töös hoidmisel, et rikete korral oleks võimalik süsteemi töökindlus tagada.

3.2 Klasterdamise eelised

Suurte andmehulkade läbi töötamiseks ja nende põhjal arvutuste tegemiseks lisaks arvutiklastri kasutamisele olemas veel üks alternatiiv: ehitada superarvuti. Superarvuti kasutamine annab eelise, kui pidevalt on tegemist väga keeruliste reaalajas tekkivate probleemide ja arvutustega, kus arvutuseks kuluv aeg on kõige tähtsam. Samas kasutatakse

tänapäeval üha enam arvutiklastreid, sest võrreldes superarvutiga on klastri ehitamine ja ülalpidamine tunduvalt soodsam ning arvutusvõimsuse lisamine ei nõua kuigi palju ressurssi.

Näiteks ilmaennustamine on tüüpiline ülesanne, mida superarvutitega seostatakse. See võtab sisendiks temperatuuri, tuule, niiskuse ja veel palju muid erinevaid tegureid, mis on pidevas muutumises. Saates sellise andmete hulga arvutiklastrisse, kuluks tulemuste saamisele märgatavalt rohkem aega ning ilmaennustus ei oleks sedavõrd täpne kui superarvutit kasutades.

Otsides mingist andmete hulgast seoseid või tehes keerulisi teaduslikke arvutusi on otstarbekam kasutada klasterdamist, kuna tulemused ei ole sõltuvuses neile kuluvast ajast. Superarvuti kasutamine raiskaks antud juhul tarbetult palju ressurssi. Suurimaks kitsaskohaks antud hetkel ongi erinevate klastris olevate arvutite vahel oleva ühenduse kiirus, mis arvutuskiirusele piirid seab. Samuti on arvutiklastri eeliseks võimalus klastris olevaid arvuteid kasutada eraldi mitmeks väiksemaks operatsiooniks.

Antud töös uuritakse lähemalt Apache Sparki, mis on üks uusimaid klasterdamise tehnoloogiaid. Tänu viimasel ajal üha soodsamaks muutunud vahemälu hindadele on Spark arendatud just selle põhjal, et arvutitevahelise ühenduse koormamise asemel kasutatakse võimalikult palju ära vahemälu.

3.3 Sparkist üldiselt

Apache Sparki arendus sai alguse Caliofornia Ülikoolis, Berkeleys (USA) 2009. aastal. 2013. aastal annetati projekt Apache Software Foundationile misjärel on see kogunud populaarsust üle kogu maailma.

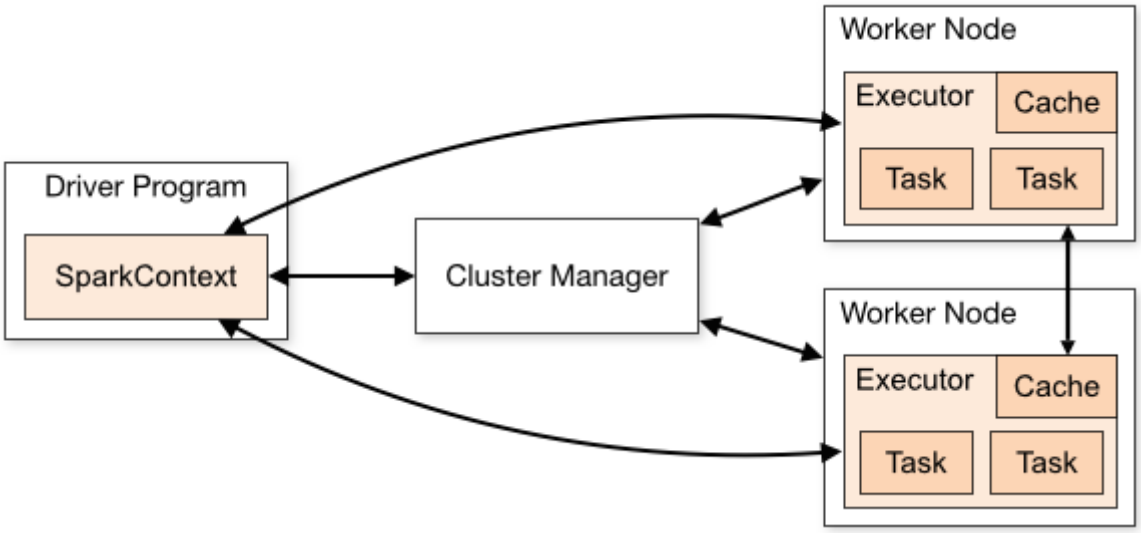
Spark on avatud lähtekoodiga klasterarvutuse raamistik, mis on arendamise algusest peale disainitud orientatsiooniga kiirusele ning kasutab erinevaid optimeerimise tehnikaid nagu näiteks mälusisene arvutamine. Tänu sellele võib Spark suurte andmekogude töötlemisel olla kuni 100 korda kiirem kui Hadoop. Näiteks 100 terabaidi andmete sorteerimiseks kulus Sparkil võrdluses Hadoopiga 3 korda vähem aega kasutades samal ajal kõigest ühte kümnendikku jõudlusest (Ostrowski 2014).

Spark pakub erinevaid lihtsalt kasutatavaid rakendusliideseid, et kirjutada programme nii Javas, Scalas, Pythonis kui Ris. Lisaks on Sparkil erinevaid mooduleid:

- Spark SQL – võimaldab töödelda andmeid sarnaselt relatsioonilistele andmebaasidele. Samuti pakub võimalust kasutada Sparki SQL päringute tegemisel.
- Spark Streaming – võimaldab töödelda andmete vooge.
- MLlib – võimaldab masinõpet, pakkudes selleks mitmeid lihtsaid algoritme.
- GraphX – võimaldab Sparki kasutada graafi andmemudelite põhjal tehtud arvutuste jaoks.

3.4 Sparki tööpõhimõte

Sparki rakendused töötavad klastris individuaalsete protsessidena. Klastri juhtimiseks saab kasutada Amazon EC2te, Hadoop YARNi ja Apache Mesost. Testimiseks ja arendamiseks on võimalik Sparki rakendada ka lokaalselt ühel arvutil – sellisel juhul jagunevad protsessid protsessori tuumade vahel.



Joonis 2. Sparki klasterdamise tööpõhimõte *(„Cluster Mode Overview“)*

Kogu Sparki tööd juhib rakenduses olev SparkContext objekt, mis ühendub klastri juhiga (Cluster Manager), mis omakorda eraldab klastris olevates arvutites tööks vajaminevad ressursid. Seejärel omandab Spark erinevatest töösõlmedest (Worker Node) saadud täitjad (Executor), mis hoiustavad andmeid ja sooritavad rakenduses olevaid arvutusi. Täitjatele saadetakse rakenduse kood (JAR või Pythoni fail mis SparkContexti saadeti) ja seejärel saadab SparkContext igale Täitjale konkreetse ülesande. („Cluster Mode Overview“)

Selline tööpõhimõte vähendab ühte põhilist klasterdamisega seotud probleemi, milleks on võrgu koormamine. Senini kasutatavad tehnoloogiad saatsid andmeid pidevalt edasi-tagasi, ehk kui üks arvutus lõppes, saadeti andmed kõigepealt tagasi klastri kontrollerile ning seejärel uuesti töösõlme. Sparki kasutades on võimalik sellist vahepealset andmete saatmist vältida (ja seega aega kokku hoida), saates väljundiks olnud andmed kohe uuesti arvutusse neid vahepeal püsimälusse kirjutamata.

Veel võimaldab Spark kirjutada lähteandmestiku vahemälusse, mistõttu saab iga ülesande juures andmed lugeda kohe mälust mitte ei pea neid lugema püsimälust.

Kolmanda suurema eelisena hoiab Spark töösõlmes JVMi (Java Virtual Machine) koguaeg töös. Seni oli levinud praktika, et koos ülesandega pannakse tööle ka uus JVM, mis võib küll iga kord võtta kõigest mõne sekundi, kuid suurte andmehulkade juures toimuks neid käivitusi mitutuhat – Sparki kasutamine annab siinkohal taaskord ajalise kokkuhoiu. (Neumann 2014)

4. Algoritmi muutmine paralleelkasutuseks

Peatükis on kirjeldatud, kuidas käib algoritmi realiseerimine kasutades Apache Sparki, alustades Sparki kasutamise eeldustest. Seejärel tutvustatakse Sparki iseärasusi ja võimalusi, mis teevad algoritmi realiseerimise mugavaks ja lihtsaks.

4.1 Sparki kasutamise eeldused

Sparki kasutamise eelduseks on viimase versiooni allalaadimine (töö kirjutamise hetkel oli selleks 1.5.2). Paralleelarvutust kasutava rakenduse jaoks on kõige mugavam kasutada Mavenit, lisades sinna Sparki sõltuvuse. Sparkiga tuleb kaasa palju erinevaid skripte, millega on võimalik teha ühes arvutis ära kogu rakenduse testimine ennem suuremas klastris katsetamist.

4.2 Sparki konfigureerimise võimalused

Spark on vaja eraldi konfigureerida iga rakenduse jaoks. Konfigureerimiseks on kolm erinevat võimalust:

1) Otse rakenduse lähtekoodis, saates vajalikud seaded otse SparkContext-i, mis tegeleb Sparki ühendusega klastris. Näide on järgnev:

```
val conf = new SparkConf()
                .setAppName(„TestRakendus“)
                .setMaster(local[2]);
Val sc = new SparkContext(conf);
```

   Antud näites seadistatakse rakenduse nimeks „TestRakendus“ ning käivitatakse see lokaalselt kasutades kahte lõimet.

2) Laadides seadistused dünaamiliselt otse käsurealt või kirjutades need eraldi konfiguratsioonifaili. Kõiki parameetreid on võimalik seadistada kirjutades rakenduse käivitamisel käsureale --conf ning selle järele seadistatava parameetri nime.

Enamkasutatavate parameetrite muutmiseks saab kasutada ka lühendeid, näiteks kasutatavate lõimede arvu saaks käsurealt muuta kirjutades --master local [2]. Konfiguratsioonifailis toimub seadistuste tegemine järgnevalt:

spark.eventLog.enabled true

Nagu näitest näha, on failis tehtavatel seadistustel seadistatav parameeter ja selle väärtus eraldatud tühikuga. Käsureale samasisulist seadet kirjutades näeks see välja selline:

--conf spark.eventLog.enabled = true

Siinkohal on kindlasti vajalik ära mainida, et parameetrite väärtuste võtmine toimub kindlas järjekorras. Kõigepealt väärtustatakse parameetrid SparkConf seadistuse järgi. Seejärel loetakse sisse parameetrite väärtused käsurealt ning kõige viimasena pöördutakse konfiguratsioonifaili poole. Seega kui SparkConf-is on rakendus seadistatud kasutama kuni 4 gigabaiti mälu, siis seda käsurealt või konfiguratsioonifailist enam muuta ei saa.

Kõiki rakenduse seadistusi on võimalik näha kasutades rakenduse toimimise ajal Sparki veebipõhist kasutajaliidest, mis üldjuhul asub aadressil http://<ajaja>:4040, kus ajaja on Sparki rakenduse aadress.

4.3 Algoritmi realiseerimine Sparkis

4.3.1 Sparki eripärad

Ennem algoritmi realiseerimise juurde asumist tasub välja tuua mõned suuremad eripärad, mida Sparki kasutamisel kindlasti silmas tuleb pidada.

- Spark põhineb RDD (Resilient Distributed Dataset) kasutamisel. RDD on elementide (või objektide) kogum, mis on jagatud erinevate lõimede vahel ning millega saab teostada paralleelarvutusi.
- RDD loomiseks on kaks võimalust:

  1) Laadides kogumi väljast sisse.

  2) Olemasoleva kollektsiooni RDDks teisendamine.

- RDDd poolt pakutavad operatsioonid jagunevad kaheks: transformatsioonideks ja tegevusteks. Transformatsioonide tulemusel tekib uus RDD, näiteks leitakse mingi tingimuse põhjal esialgsest RDDst mingi alamhulk, mille põhjal edasi toimetada. Tegevused aga tagastavad vastavalt valitud funktsioonile mingi kindla tulemuse või teevad mõne kindla operatsiooni. Tegevuse näitena võib tuua RDD salvestamise failina.

4.4 Algoritmi realisatsioon

4.4.1 Algoritmi realiseerimine ilma Sparkita

Algoritm realiseeriti kõigepealt Javas ilma Sparkita, et näha, milliseid operatsioone ja millises järjekorras on vaja andmestikus leiduvate reeglite leidmiseks läbi viia.

Realiseeritud programmi lähtekood on nähtav peatükis 9.1. Järgnevalt on kirjeldatud olulisemad punktid, mida täpselt andmestikuga ühe reegli välja selgitamiseks tehakse.

1) Loetakse sisse CSV fail, mis sisaldab testiandmestikku. Antud töös eeldatakse, et failis on rangelt binaarandmed ehk andmestikus vigu ei ole.

2) CSV faili põhjal luuakse ArrayList<int[]> tüüpi muutuja *dataAsList*.

3) Leitakse esialgsed ühtede ja nullide esinemissagedused.

4) Käivitub reeglite leidmise funktsioon. Tsükliliselt käiakse läbi kõik veerud, koheldes iga tsükli jooksul ühte veergu klassiveeruna.

5) Reeglite leidmiseks leitakse *dataAsList* loendist alamhulk klassiveeru ja klassiveeru väärtuse (kas 0 või 1) järgi.

6) Leitud alamhulgale arvutatakse ühtede ja nullide esinemissagedused.

7) Võrreldakse ühtede ja nullide esinemissagedusi ja tagastatakse selle põhjal leitud reeglid. Kui esialgsed sagedused ja alamhulga sagedused kattuvad, alamhulk ei ole võetud klassitunnuse järgi ning klassiväärtus ja tunnuse väärtus on võrdelised, oleme leidnud reegli (kaks võimalust reegli tekkimiseks, klassiväärtus 1 ja tunnuse väärtus 1 või klassiväärtus 0 ja tunnusväärtus 0).

Teine reegel ilmneb, kui klassiväärtus ja tunnuse väärtus ei ole võrdelised ning leitud alamhulgas on antud tunnuse esinemissagedus 0.

Nagu eelnevast kirjeldusest näha võib, ei ole kuni 3. punktini paralleelsust vaja, kuna tegemist on triviaalsete operatsioonidega nagu faili sisselugemine ja teisendamine. Samas alates neljandast punktist saaks antud realisatsiooni tõepoolest muuta paralleelseks. Selleks oleks vaja ette anda lähteandmestik, klassitunnusena võetava veeru number ning selle veeru väärtus.

4.4.2 Algoritmi realisatsioon Sparkis

Algoritmi realiseerimisel Sparkis võeti aluseks punktis 4.4.1 kirjeldatud Java programm, millele lisati paralleelsuse tagamiseks Sparki tugi.

Sparkis realiseeritud programm on näha peatükis 9.2. Järgnevalt selgitatakse, millised muudatused koodi sisse viia vaja oli ja kuidas Spark antud juhul paralleelselt tegutseb.

Esmalt lisati SparkConf muutuja, mis määrab ära Sparki konfiguratsiooni. Lokaalseks testimiseks piisab kui määrata ära rakenduse nimi ja rakenduse URL, mille peal see käima pannakse. Spark läheb käima kui JavaSparkContext saab sisendiks SparkConf-tüüpi muutuja.

Järgmisena teisendatakse failist sisse loetud ja 2-dimensioonilisest massiivist ArrayList <int[]> tüüpi muutujaks tehtud andmestik sobilikuks paralleelkasutusele ehk teisendatakse JavaRDDks. Olemasoleva ArrayList või List tüüpi andmestiku saab lihtsasti teisendada JavaRDDks kasutades .parallelize() meetodit.

Antud töös otsustati paralleelsus realiseerida nii, et iga veergu oleks võimalik paralleelselt töödelda. Selleks koostati for tsükkel, mis võtab muutujaks veeru indeksi. Veeru indeksi järgi koostatakse alamhulk mis teisendatakse .collect() funktsiooni kasutades List<int[]> tüüpi muutujaks. Selline teisendus võimaldab ära kasutada eelnevalt realiseeritud getSequences2 meetodit alamhulga sageduste leidmiseks. See omakorda lubab järgmise sammuna reeglite leidmiseks ära kasutada getRules() meetodit täpselt samamoodi nagu puntkis 4.4.1 realiseeritud programmis.

Paralleelsus on tagatud tänu sellele, et Spark on üles ehitatud n-ö laisana, ehk andmetega ei hakata reaalselt ennem midagi tegema, kui tuleb mõni funktsioon mis neid välja kutsub (antud juhul on selles .collect() alamhulga teisendamisel). See võimaldab maksimaalselt ära kasutada vahemälu ja ei koorma klastrite vahelist võrguühendust, kuna andmete pidev edasi-tagasi

saatmine jääb ära. Teisalt on selline lähenemine halb, eriti just programmi arendamisel, kuna see ajab keeruliseks andmete ühest arvutusest/teisendusest teise viimise ning vea korral on raske leida, kuskohast see täpselt pärineb. Suurte andmekogude ja hästi valmis programmeeritud Sparki programmi puhul on tänu sellele võimalik hoida kokku väga palju aega.

Ilmselt oleks järgmisel korral Sparkis mõnda algoritmi realiseerides mõistlikum võtta kasutusele natukene keerulisem algoritm, kuna Sparki efektiivsus teiste lahenduste ees tuleb esile just siis, kui ühe andmestikuga järjest teha keerulisemaid operatsioone.

5. Spark'i efektiivsuse analüüs

Sparki efektiivsuse analüüsimiseks võrreldakse punktis 4.4.1 realiseeritud Java programmi puntkis 4.4.2 realiseeritud Sparki kasutava Java programmiga. Testimiseks kasutatakse kahte erineva suurusega andmestikku.

5.1 Testkeskkond

Testkeskkonna spetsifikatsioon on järgnev:

- Protsessor: Intel Core i5-3210M 2.5GHz
- Vahemälu: 6 GB DDR3
- Operatsioonisüsteem: Windows 7 64-bit
- Java versioon 1.8
- Spark versioon 1.5.2

Mõlemad testiprogrammid käivitatakse samadel tingimustel. Erinevuseks on see, et Sparki kasutavale programmile antakse konfiguratsioonis ette, et ta saab paralleelselt toimetada neljas lõimes.

5.2 Testiandmestik

Testimiseks on kasutusel kaks andmestikku:

1) Esimene väiksem andmestik koosneb 23 reast ja 6 veerust. Edaspidi nimetatakse seda kui A1.
2) Teine andmestik on tervisehoiust pärinev ning koosneb 187 reast ja 23 veerust. Edaspidi nimetatakse seda kui A2.

5.3 Mälukasutus

Mälukasutuse mõõtmiseks jooksutati mõlemat testprogrammi mõlema andmestikuga 100 korda ja leiti nende keskmised. Tulemused mälukasutuse kohta megabaitides on esitatud alljägnevas tabelis:

	Java	Spark	Spark*
A1	2	76	64
A2	12	80	68

Tabel 1. Mälukasutuse võrdlus

Mälukasutust vaadates on selge, et Spark on ehitatud maksimaalselt ära kasutama sisemälu. Selle tulemusena koormatakse palju vähem võrguühendust, mis tagab kogu klastri efektiivsema toimimise. Samas on näha, et mälukasutus ei suurenenud oluliselt andmestiku suurendes. Puhtalt Javat kasutades on näha, et mälukasutus 31 korda suurema andmestikuga A2 on 6 korda suurem kui andmestiku A1 korral. Samas Sparki puhul oli vastava näitaja muutus oluliselt väiksem, kusjuures seda ka arvuliselt võttes.

5.4 Ajakulu

Ajakulu mõõtmiseks jooksutati mõlemat testprogrammi mõlema andmestikuga 100 korda ja leiti nende keskmised. Tulemused ajakulu kohta sekundites on esitatud alljärgnevas tabelis:

	Java	Spark	Spark*
A1	0,13	11,33	2,87
A2	0,96	14,36	4,89

Tabel 2. Ajakulu võrdlus

Nagu näha, on nii andmestiku A1 kui andmestiku A2 puhul efektiivsem kasutada puhtalt Javas kirjutatud rakendust. Spark* veeru jaoks alustati aja mõõtmist sellest kui Spark ise oli käima pandud ehk see veerg väljendab reaalselt Sparkis programmi jooksmise aega. Testiandmestike väiksusest tingituna võtab Sparki programm aega kuni 100 korda rohkem kui puhtalt Javas realiseeritud programm. Samas on ka näha, et lahutades sellst Sparki taustaprotsesside töölerakendumise aja ei ole tulemused sugugi nii erinevad, andmestiku A2 korral kõigest 4 kordsed.

Veel tasub märkimist, et testandmestik A2 on 31 korda suurem kui A1. Javas realiseeritult tähendas selline muutus, et reeglite leidmiseks andmestiku A2 korral kulus ligikaud 7 korda

rohkem aega kui andmestiku A1 korral. Samas Sparki kasutades kasvas puhtalt Sparki osale kulunud aeg kõigest 2 korda. Arvestades, et realiseeritud programm kasutas paralleelsust ära minimaalsel tasemel saab järeldada, et suuremate andmehulkade ja keerukamate algoritmide realiseerimisel tuleneks siinkohal oluline ajavõit.

6. Kokkuvõte

Töö põhieesmärgiks oli uurida ja tutvustada lugejale moodsat klasterarvutamise lahendust – Apache Sparki. Selleks tutvustati andmekaevet ja monotoonseid süsteeme ning anti ülevaade Sparki poolt pakutavatest põhilistest võimalustest. Töö põhiosa moodustab ühe monotoonset süsteemi kirjeldava andmekaevealgoritmi realiseerimine Javas ning sellele seejärel Sparki toe lisamine ja tulemuste analüüs.

Suurte andmestike analüüsimine ja andmekaeve on järjest enam populaarsust koguvad teemad. Kuna antud protsessid on küllaltki ressursimahukad, pakub Apache Spark hea võimaluse ressursside jagamiseks klasterdamise läbi. Andmekaevealgoritmi realiseerimine Javas ning seejärel Sparkis kasutuskõlbulikuks tegemine seisneb suuresti JavaRDD andmeobjektide olemuse mõistmises. Samas peab märkima, et nagu antud töös järeldus, ei ole mõistlik liiga lihtsaid algoritme ja väikeste andmehulkadega tehtavaid operatsioone Sparki jaoks kasutamiseks teisendada, kuna need võivad võtta rohkem ressursse kui esialgu arvatud ning neid on Sparkis parimat efektiivsust silmas pidades ka ebamõistlikult raske realiseerida.

Üheks variandiks antud lähenemisega edasi minna, oleks testida Sparki mitte lokaalselt, vaid reaalses arvutiklastris, kasutades selleks ka mõnda keerulisemat algoritmi või suuremat hulka operatsioone. Lihtne oleks kasutada näiteks Amazoni poolt pakutavat pilvelahendust EC2, mis pakub arvutusvõimsuse rentimise võimalust ja mille kasutamise jaoks vajalikud Sparki skriptid tulevad Sparkiga kaasa.

Lisaks oleks võimalik algoritmi realiseerimiseks ära kasutada Sparki poolt pakutavat MLlib teeki, mis pakub selleks erinevaid võimalusi.

Töö tegemisel seatud eesmärgid saavutati ja tõestati monotoonsete süstemide paralleelseks muutmise võimalus. Algoritm realiseeriti kõigepealt Javas ning seejärel Sparkis, kasutades paralleelarvutusi.

7. Summary

The purpose of this thesis was to introduce Apache Spark - a modern cluster computing solution. Data mining and monotone systems approach was also introduced and a short summary of what Sparks possibilities was made. The main part of this thesis is implementing a data mining algorithm for a monotone system firstly using only Java and then adding Spark support for it.

Big data analysis and data mining are getting popular day by day. Because both of the aforementioned are recourse costly, a great way to improve effectiveness is to use cluster computing supported by Spark. The biggest challenge in Spark is getting used to JavaRDD data objects. As was found out during the implementation of the algorithm in this work, Spark is not best suited for implementing simpler algorithms because of the somewhat complexity JavaRDD type objects introduce. However if the datasets given are big and multiple complex computations are done based on them, Spark can significantly improve effectiveness.

One could go further with Spark using it not locally but as a real cluster. For example using Amazon EC2 cloud computing solution for the realisation of some more complex algorithm could give yet better example of the ease-of-use and effectiveness Spark is built on.

Furthermore it would also be possible to use teh MLlib library provided by Spark to implement different algorithms as the MLlib library provides many possibilities concering machine learning.

The objectives set in the start of this thesis were accomplished. Data mining algorithm was firstly implemented in Java and then in Java using Spark.

8. Kasutatud kirjandus

Aab, Eik, (2002), „Uus vaade monotoonsetele süsteemidele“. [Võrgumaterjal]. Saadaval: http://deepzone0.ttu.ee/aa/modules.php?name=News&file=article&sid=223

Cluster Mode Overview. [Võrgumaterjal]. Saadaval: http://spark.apache.org/docs/latest/cluster-overview.html

Möls, Märt, (2012), Lihtne lineaarne regressioon. [Võrgumaterjal]. Saadaval: http://www-1.ms.ut.ee/mart/biomeetria2012/loeng6A.pdf

Neumann, Saggi, (2014), Spark vs. Hadoop MapReduce. [Võrgumaterjal]. Saadaval: https://www.xplenty.com/blog/2014/11/apache-spark-vs-hadoop-mapreduce/

Niglas, Katrin, (2013), Regressioonanalüüs. [Võrgumaterjal]. Saadaval: http://www.cs.tlu.ee/~katrin/wp/wp-content/uploads/2013/11/regressioon.pdf

Ostrowski, Radek, (2014), Introduction to Apache Spark with Examples and Use Cases. [Võrgumaterjal]. Saadaval: http://www.toptal.com/spark/introduction-to-apache-spark

Roosmann, Peeter, Võhandu, Leo, Kuusik, Rein, Treier, Tarvo ja Grete Lind, (2008), „Monotone Systems approach in Inductive Learning“. International Journal of Applied Mathematics and Informatics 2.2 (2008): 47-56.

Testimiseks kasutatud andmestik tervisehoiu andmestik. https://archive.ics.uci.edu/ml/datasets/SPECT+Heart

Vreeken, Jilles, (2009), „Making Pattern Mining Useful“. ACM SIGKDD Explorations Newsletter 12.1, lk 75-76, Saadaval: http://www.cs.uu.nl/groups/ADA/pubs/2009/making_pattern_mining_useful-vreeken.pdf

9. Lisad

9.1 Lisa 1: Andmekaevealgoritm Javas

Algoritm on saadaval ka Gitis: https://github.com/silverpuulmann/loputoo

Andmekaevealgoritmi esialgne realisatsioon Javas, ilma paralleelarvutust kasutamata.

```
package monsa2;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Scanner;

public class test {

	static int[] sequencesZero;		// Esialgse hulga nullide sageduste massiiv.
	static int[] sequencesOne;		// Eialgse hulga ühtede sageduste massiiv.
	static int[] subSequences;		// Alamhulga puhul leitav sageduste massiiv.

	static ArrayList<int[]> dataAsList;	// 2-dimensiooniline massiiv andmete hoidmiseks.
	static ArrayList<int[]> sequenceClass; //
	private static Scanner sc;			// Skänner faili sisselugemiseks.

	static ArrayList<String> rules = new ArrayList<>();		//2-dimensiooniline tekstimassiiv reeglite hoidmiseks.


	public static void main(String[] args) throws IOException {

		File file = new File("SPECT.csv");	// Testimiseks kasutatav binaarandmetega monotoonset süsteemi kirjeldav fail.
		int data[][] = readCSV(file);		// Csv faili sisselugemine ja sellest 2d massiivi tegemine.
		dataAsList = dataToList(data);		// Teeb tehtud 2d massiivi massiivi listiks.
		printData(dataAsList);			// Sisseloetud csv-faili välja printimine massiivina.

		System.out.println("Algab reeglite leidmine!");
		sequencesZero = getSequences(data, 0); // Leitakse sagedustabel nullide jaoks.
```

```
                sequencesOne = getSequences(data, 1);  // Leitakse sagedustabel ühtede jaoks.

                monoSys(data);                  // Reeglite leidmiseks kasutatav funktsioon.

                printRules();               // Reeglite väljatrükkimiseks kasutatav funktsioon.

        }
        /*
                Reeglite leidmise funktsioon.
                sequences2 - esialgsed sagedused
                subSequences2 - alamhulga sagedused
                classVariable - klassitunnuse veeru number, mille kohta tunnuseid leitakse
                classValue - klassitunnuse väärtus (1 või 0)
                classChar - võrreldava veeru väärtus (1 või 0)
        */
        private static void getRules(int[] sequences2, int[] subSequences2, int classVariable,
int classValue, int classChar) {

                for (int i = 0; i < sequences2.length; i++){

                        if (sequences2[i] == subSequences2[i] && i != classVariable &&
classValue == classChar){
                                rules.add("Klassiveeruks on " + (classVariable + 1) + ". veerg.
Kui veerg " + (i+1)
                                                + " on " + classChar + " on klassiveerg (" +
(classVariable + 1) + ". veerg) " + classValue + " !");
                        }
                        if (subSequences2[i] == 0 && classValue != classChar){
                                rules.add("Klassiveeruks on " + (classVariable + 1) + ". veerg.
Kui veerg " + (i+1)
                                                + " on " + classChar + " on klassiveerg (" +
(classVariable + 1) + ". veerg) " + classValue + " !");
                        }
                }
        }
        /*
                Reeglite väljatrükkimise funktsioon. Trükib välja kõik 'rules' listis olevad
reeglid.
        */
        private static void printRules(){
                System.out.println("Andmestikus olevad reeglipärasused: ");
                for (String rule : rules){
                        System.out.println(rule);
                }
        }

        /*
                Andmete teistendamise funktsioon 2d-massiivist list-massiiv tüüpi muutujasse.
                Tagastab list-massiiv tüüpi muutuja.
        */
```

```
	private static ArrayList<int[]> dataToList(int[][] data) {

		ArrayList<int[]> list = new ArrayList<int[]>(data.length);
		for(int[] foo : data) {
		   list.add(foo);
		}
		return list;
	}

	/*
		Alamhulga leidmise funktsioon.
		classDefiner - klassitunnuse väärtus, mille järgi alamhulk leitakse (kas 0 või 1)
		dataAsList2 - sisendiks saadav andmete hulk (kogu andmestik)
		columnVariable - klassitunnust sisaldav veerg, mille järgi alamhulk leitakse
		Tagastab list-massiiv tüüpi alamhulga.
	*/
	private static ArrayList<int[]> getByClass(int classDefiner, ArrayList<int[]>
dataAsList2, int columnVariable) {

		ArrayList<int[]> newList = new ArrayList<int[]>();

		for (int[] foo : dataAsList2) {
			if (foo[columnVariable] == classDefiner) {
				newList.add(foo);
			}
		}
		return newList;
	}

	/*
		Sageduste leidmise funktsioon.
		list - sisendiks saadav hulk
		definer - määrab, millele sagedused leitakse (kas 0 või 1)
		Tagastab hulga sagedused.
	*/
	private static int[] getSequences(ArrayList<int[]> list, int definer) {

		int rowLength = list.get(0).length; //6
		int colLength = list.size();		//23
		int[] countOfVar = new int[rowLength];
		int count = 0;

		for (int col = 0; col < rowLength; col++) {

			for (int row = 0; row < colLength; row++) {

				int[] curRow = list.get(row);

				if (curRow[col] == definer) {
					count++;
```

```
                        }
                    }
                    countOfVar[col] = count;
                    count = 0;
                }

                System.out.println("Alamhulga sagedused loetud! " +
Arrays.toString(countOfVar));
                return countOfVar;
        }

        /*
                CSV faili sisselugemise funktsioon.
        */
        public static int[][] readCSV(File file) throws FileNotFoundException{

                int[][] fxRates = new int[187][23];
                String delimiter = ";";
                int line = 0;

                    sc = new Scanner(file);

                  while (sc.hasNextLine()) {
                    String line2 = sc.nextLine();
                    String[] fxRatesAsString = line2.split(delimiter);
                    for (int i = 0; i < fxRatesAsString.length; i++) {
                      fxRates[line][i] = Integer.parseInt(fxRatesAsString[i]);
                    }
                    line++;
                  }
                  //System.out.println(Arrays.deepToString(fxRates));
                  System.out.println("Fail edukalt sisse loetud!");
                    return fxRates;
        }

        /*
                Sisseloetud failist tehtud massiivi välja trükkimise funktsioon.
        */
        public static void printData(ArrayList<int[]> data) {

                for (int[] foo : data) {
                    System.out.println("Prindin v2lja listi" + Arrays.toString(foo));
                    }
                System.out.println("Listi l6pp!");
        }

        /*
                Andmestikust reeglite leidmine
                Tehakse läbi 4 korda:
                    1) Juhul kui klassitunnus on 1 ja tunnuse veerg on 1
```

```
                        2)Juhul kui klassitunnus on 0 ja tunnuse veerg on 0
                        3)Juhul kui klassitunnus on 0 ja tunnuse veerg on 1
                        4)Juhul kui klassitunnus on 1 ja tunnuse veerg on 0
        */
        public static void monoSys(int[][] data){

                for (int i = 0; i < data[0].length; i++){
                        System.out.println("Leian reeglid kui klassiveerg on " + (i + 1) + "
v22rtusega 1 ja tunnuse veerg on v22rtusega 1.");
                        sequenceClass = getByClass(1, data, i);
        //klassiveeru järgi alamhulga leidmine
                        subSequences = getSequences2(sequenceClass, 1);  //alamhulgale
sageduste arvutamine
                        getRules(sequencesOne, subSequences, i, 1, 1);                  //reeglite
leidmine kui klassiveerg on 1 ja reegli tekkimise veerg on 1
                }

                for (int i = 0; i < data[0].length; i++){
                        System.out.println("Leian reeglid kui klassiveerg on " + (i + 1) + "
v22rtusega 0 ja tunnuse veerg on v22rtusega 0.");
                        sequenceClass = getByClass(0, data, i);
                        subSequences = getSequences2(sequenceClass, 0);
                        getRules(sequencesZero, subSequences, i, 0, 0);
                }

                for (int i = 0; i < data[0].length; i++){
                        System.out.println("Leian reeglid kui klassiveerg on " + (i + 1) + "
v22rtusega 0 ja tunnuse veerg on v22rtusega 1.");
                        sequenceClass = getByClass(0, data, i);
                        subSequences = getSequences2(sequenceClass, 0);
                        getRules(sequencesOne, subSequences, i, 1, 0);
                }

                for (int i = 0; i < data[0].length; i++){
                        System.out.println("Leian reeglid kui klassiveerg on  " + (i + 1) + "
v22rtusega 1 ja tunnuse veerg on v22rtusega 0.");
                        sequenceClass = getByClass(1, data, i);
                        subSequences = getSequences2(sequenceClass, 1);
                        getRules(sequencesZero, subSequences, i, 0, 1);
                }
        }
}
```

9.2 Andmekaevealgoritm Sparkis

Algoritm on saadaval ka Gitis: https://github.com/silverpuulmann/loputoo

Andmekaevealgoritmi realisatsioon kasutades Sparki.

```
package monsa2spark;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Scanner;

import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.function.Function;

public class test {

	static ArrayList<int[]> dataAsList;
	static ArrayList<String> rules = new ArrayList<>();

	static int[] sequencesZero;
	static int[] sequencesOne;
	static ArrayList<int[]> sequenceClass;
	static int[] subSequences;



	public static void main(String[] args) throws FileNotFoundException{


		/*if (args.length != 3) {
		    System.err.println("Midagi on valesti! Sisesta argumendid kujul <fail> <ridasid> <veerge>");
		    System.exit(1);
		  }
		File file = new File(args[0]);*/

		//File file = new File(args[0]);
		//int rows = Integer.parseInt(args[1]);
		//int columns = Integer.parseInt(args[2]);

		File file = new File("test.csv");
		int rows = 23;
		int columns = 6;
```

```
		SparkConf sparkConf = new SparkConf().setAppName("MonsaSpark")

	.setMaster("local[4]");
		JavaSparkContext sc = new JavaSparkContext(sparkConf);

		int data[][] = readCSV(file, rows, columns); // CSV faili sisselugemine ja sellest 2d massiivi tegemine.
		dataAsList = dataToList(data); // 2d massiivi Arraylistiks teisendamine.

		System.out.println("Algab reeglite leidmine!");
		sequencesZero = getSequences(data, 0); // esialgse sagedustabeli loomine
		sequencesOne = getSequences(data, 1);

		JavaRDD<int[]> dataRDD = sc.parallelize(dataAsList); // Muudab listi JavaRDD<int[]> tüüpi massiiviks, mis võimaldab

			// teostada selle peal operatsioone paralleelselt.

		/*
		 *	Veergude läbi käimine reeglite leidmiseks, mis kasutab Sparki pakutavat paralleelsust.
		 * Tööpõhimõtet on täpselt seletatud töös peatükis 4.4.2
		 */

		for (int i = 0; i < columns; i++) {
			JavaRDD<int[]> thisRDD = getSubClasses(dataRDD, i);
			List<int[]> thisRDDAsList = thisRDD.collect();
			int[] thisRDDSequences = getSequences2(thisRDDAsList, 1);
			getRules(sequencesOne, thisRDDSequences, i, 0, 1);

		}

		printRules();

	}

	/*
		Täiendavalt Sparki jaoks lisatud funktsioon, mida kasutatakse alamhulkade leidmiseks.
		.filter() funktsioon tagastab ainult need elemendid, mis vastavad if lauses seatud tingimusele.
	 */

	private static JavaRDD<int[]> getSubClasses(JavaRDD<int[]> dataRDD, int col) {

		JavaRDD<int[]> onesRDD = dataRDD.filter(
				data -> {
					if (data[col] == 1){
```

```
                                        return true;
                                    }
                                    return false;
                                });

                return onesRDD;
        }

        /*
                CSV faili sisselugemise funktsioon. Saab sisendiks faili, ridade arvu ja veergude arvu.
                Tagastab 2-dimensioonilise täisarvude massiivi.
         */
        public static int[][] readCSV(File file, int rows, int columns) throws FileNotFoundException{

                int[][] fxRates = new int[rows][columns];
                String delimiter = ";";
                int line = 0;

                        Scanner sc = new Scanner(file);

                   while (sc.hasNextLine()) {
                      String line2 = sc.nextLine();
                      String[] fxRatesAsString = line2.split(delimiter);
                      for (int i = 0; i < fxRatesAsString.length; i++) {
                         fxRates[line][i] = Integer.parseInt(fxRatesAsString[i]);
                      }
                      line++;
                   }
                   sc.close();
                   System.out.println("Fail edukalt sisse loetud!");
                        return fxRates;
        }

        /*
                Andmete teistendamise funktsioon 2d-massiivist list-massiiv tüüpi muutujasse.
                Tagastab list-massiiv tüüpi muutuja.
         */

        private static ArrayList<int[]> dataToList(int[][] data) {

                ArrayList<int[]> list = new ArrayList<int[]>(data.length);
                for(int[] foo : data) {
                   list.add(foo);
                }
                return list;
        }

        /*
```

```
                Sageduste leidmise funktsioon.
                list - sisendiks saadav hulk
                definer - määrab, millele sagedused leitakse (kas 0 või 1)
                Tagastab hulga sagedused.
         */
        private static int[] getSequences(int[][] data, int definer) {

                int[] sequences = new int[data[0].length];
                int count = 0;

                for (int i = 0; i < data[0].length; i++) {  // i on veeru indeks ehk antud juhul 0-6

                        for (int j = 0; j < data.length; j++){ // j on rea indeks ehk antud juhul 0-
23

                                if(data[j][i] == definer){ // j on rea indeks ja i on veeru indeks
                                count++; // leiti yks, yhtede arvu suurendatakse
                                }
                        }
                        sequences[i] = count;
                        count = 0;
                }
                System.out.println("Leitud esialgsed sagedused " + definer + "-de kohta: " +
Arrays.toString(sequences));
                return sequences;
        }

        /*
                Reeglite leidmise funktsioon.
                sequences2 - esialgsed sagedused
                subSequences2 - alamhulga sagedused
                classVariable - klassitunnuse veeru number, mille kohta tunnuseid leitakse
                classValue - klassitunnuse väärtus (1 või 0)
                classChar - võrreldava veeru väärtus (1 või 0)
         */

        private static void getRules(int[] sequences2, int[] subSequences2, int classVariable,
int classValue, int classChar) {

                for (int i = 0; i < sequences2.length; i++){
                //System.out.println("i v22rtus on : " + i);
                        if (sequences2[i] == subSequences2[i] && i != classVariable &&
classValue == classChar){
                                System.out.println("Leidsin reegli!" + i + ", " + classVariable);
                                rules.add("Klassiveeruks on " + (classVariable + 1) + ". veerg.
Kui veerg " + (i+1)
                                                + " on " + classChar + " on klassiveerg (" +
(classVariable + 1) + ". veerg) " + classValue + " !");
                        }
                        if (subSequences2[i] == 0 && classValue != classChar){
```

```
                                System.out.println("Leidsin reegli!" + i + ", " + classVariable);
                                rules.add("Klassiveeruks on " + (classVariable + 1) + ". veerg.
Kui veerg " + (i+1)
                                                + " on " + classChar + " on klassiveerg (" +
(classVariable + 1) + ". veerg) " + classValue + " !");
                        }
                        //else{System.out.println("Ei leidnud reeglit!");}

                }
        }

        /*
                Reeglite väljatrükkimise funktsioon. Trükib välja kõik 'rules' listis olevad
reeglid.
         */
        private static void printRules() {
        System.out.println("Andmestikus olevad reeglip2rasused: ");
                for (String rule : rules){
                System.out.println(rule);
                }
        }
        /*
                Alamhulgast sageduste leidmise funktsioon
                Saab sisendiks alamhulga andmestiku ja väärtuse millele sagedusi leida (0 või
1).
                Tagastab alamhulga elementide esinemise sagedused.
         */

        private static int[] getSequences2(List<int[]> list, int definer) {

                int rowLength = list.get(0).length; //6
                int colLength = list.size(); //23
                int[] countOfVar = new int[rowLength];
                int count = 0;

                for (int col = 0; col < rowLength; col++){

                        for (int row = 0; row < colLength; row++){

                                int[] curRow = list.get(row);

                                if (curRow[col] == definer){
                                        count++;
                                }
                        }
                        countOfVar[col] = count;
                        count = 0;
                }
```

```
                System.out.println("Alamhulga sagedused loetud! " +
Arrays.toString(countOfVar));
                return countOfVar;
                }
        }
```