UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO
FAKULTETA ZA MATEMATIKO IN FIZIKO

Matej Ugrin

Strežnik z odlogom za sistem Hadoop

DIPLOMSKO DELO
NA INTERDISCIPLINARNEM UNIVERZITETNEM ŠTUDIJU
RAČUNALNIŠTVA IN MATEMATIKE

Mentor: dr. Andrej Brodnik

Ljubljana, 2012

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00037/2012
Datum: 13.04.2012

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko ter Fakulteta za matematiko in fiziko izdaja naslednjo nalogo:

Kandidat: **MATEJ UGRIN**

Naslov: **STREŽNIK Z ODLOGOM ZA SISTEM HADOOP**
DEFERABLE SERVER FOR A HADOOP SYSTEM

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Uporaba funkcij kot spremenljivk tipa prvega reda, o kateri pišeta Cardelli in Wegner, je sprožila njihovo rabo v konstruktih kot je zavse (forall) ter v bolj strukturiranem slikaj/zloži (map/fold). Oba konstrukta je v določenih primerih moč dobro povzporediti in primer takšnega povzporejanja je nov konstrukt slikaj/zberi (map/reduce). Pri izvedbi slednjega igra eno poglavitnih vlog razporejevalnik.
V diplomski nalogi preučite implementacijo konstrukta slikaj/zberi v sistemu Hadoop. V sistemu nadgradite pravični razporejevalnik z razporejevalnikom, ki bo pravičnost časovno razširil podobno kot to počno strežniki sporadičnih poslov v sistemih v realnem času. Novi razporejevalnik primerjajte z obstoječim razporejevalnikom.

Mentor:

doc. dr. Andrej Brodnik

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan Fakultete za računalništvo in informatiko:

prof. dr. Nikolaj Zimic

Dekan Fakultete za matematiko in fiziko:

akad. prof. dr. Franc Forstnerič

za

UNIVERZA V LJUBLJANI • Fakulteta za matematiko in fiziko • 1

IZJAVA O AVTORSTVU

diplomskega dela

Spodaj podpisani/-a Matej Ugrin,

z vpisno številko 63040302,

sem avtor/-ica diplomskega dela z naslovom:

Strežnik z odlogom za sistem Hadoop

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal/-a samostojno pod mentorstvom dr. Andreja Brodnika
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 15. 10. 2012 Podpis avtorja/-ice:

Zahvala

Zahvaljujem se mentorju dr. Andreju Brodniku za napotke in popravke pri izdelavi diplomskega dela. Za pomoč in številne diskusije bi se rad zahvalil tudi Andreju in vsem prijateljem ter sošolcem, ki so me spremljali v času študijskih let.

Posebno zahvalo namenjam svoji družini, ki mi je nudila podporo in spodbudo v vseh letih študija. Hvala tudi Sari za vse lepe trenutke in oporo.

Kazalo

Povzetek **1**

Abstract **2**

1 Uvod **3**

2 MapReduce **5**
2.1 Programski model MapReduce 6
2.2 Izvajalno ogrodje 7
2.3 Primeri implementacij modela MapReduce 10

3 Ogrodje Hadoop **16**
3.1 Uvod 16
3.2 Področja in primeri uporabe 18
3.3 Arhitektura in osnovni gradniki 21
3.4 Porazdeljen datotečni sistem HDFS 22
3.5 Pregled izvajanja modela MapReduce v ogrodju Hadoop 24

4 Razporejanje v okolju Hadoop **27**
4.1 Uvod v razporejanje 27
4.2 Primeri razporejevalnikov 29
4.2.1 Prioritetni razporejevalnik 29
4.2.2 Pravični razporejevalnik 29
4.2.3 Razporejanje z določanjem kapacitet 30
4.2.4 Razporejevalnik Lsched 30
4.2.5 Razporejevalnik z dinamičnim spreminjanjem prioritet 32
4.2.6 Razporejevalnik LATE in razporejevalnik z zakasnjenim izvajanjem 32
4.3 Pravični razporejevalnik 33

4.3.1 Predstavitev razporejevalnika 34
4.3.2 Osnovni gradniki 35
4.3.3 Algoritem pravičnega razporejanja in izbire opravil 39
4.3.4 Zagotavljanje pravičnosti in prekinitve izvajanja 41
4.3.5 Izračun pravičnih deležev 42
4.3.6 Algoritem zakasnjenega razporejanja 43

5 Implementacija uteženega razporejanja s pravičnim oknom 47
5.1 Uvod 47
5.2 Model pravičnega okna 48
5.2.1 Algoritem razporejanja s pravičnim oknom 50
5.2.2 Opis delovanja pravičnega okna 52
5.2.3 Vpliv velikosti okna na pravičnost razporejanja 53
5.3 Evaluacija 55
5.3.1 Priprava testnega okolja 55
5.3.2 Ocenjevanje pravičnosti dodelitev 57
5.3.3 Meritve in diskusija rezultatov 59
5.4 Sklepne ugotovitve 64

6 Zaključek 66

A Izvorna koda razporejevalnika FWS 68

B Spremljanje izvajanja opravil z orodjem Ganglia 69

Literatura 73

Seznam uporabljenih kratic in simbolov

slot	računski vir delovnega vozlišča za izvajanje opravil
task	opravilo – osnovna enota opravljenega dela
job	posel – sestavljen je iz map in reduce opravil
pool	bazen – struktura v katero so posli izstavljeni v izvajanje
minshare	minimalni delež bazena – zagotovljeno število računskih virov na voljo bazenu ob vsakem razporejanju
sf_i	pravični delež bazena i – predvideno število dodeljenih virov bazena ob pravičnem razporejanju (angl. fair share)
sa_i	število virov, ki so bili dodeljeni bazenu i (angl. assigned slots)
dc_i	primankljaj bazena i (angl. deficit)
$\Delta(t)$	ocena pravičnosti razporejanja ob času t
tie	trenutek neodločljivosti pri razporejanju s pravičnim oknom
heartbeat	sistem za pošiljanje statusnih sporočil
FWS	Razporejevalnik s pravičnim časovnim oknom (angl. Fair Window Scheduler)

Povzetek

Naraščajoče količine podatkov, njihova raznolikost in zahteve po odzivnejših sistemih narekujejo razvoj novih tehnologij in modelov za podatkovno intenzivno ter porazdeljeno računanje.

V diplomskem delu smo obravnavali in podrobneje opisali ogrodje *Apache Hadoop*, ki skupaj s porazdeljenim datotečnim sistemov in programskim modelom *MapReduce*, omogoča hranjenje in obdelavo velikih zbirk podatkov. Pregledali smo nekaj primerov konkretnih implementacij in značilnosti modela *MapReduce* ter raziskali področja in primere uporabe ogrodja *Hadoop*. Ker razporejanje predstavlja eno ključnih komponent ogrodja, smo preučili različne vrste razporejevalnikov in jih skušali smiselno uvrstiti v skupine glede na njihov namen. Podrobneje smo opisali pravični razporejevalnik, ki predstavlja osnovo za izgradnjo novega razporejevalnika *FWS*, katerega namen je izboljšanje odzivnosti poslov in njihovo enakomernejše izvajanje z uporabo pravičnejšega dodeljevanja virov. Slednje smo skušali doseči z vpeljavo strukture pravičnega časovnega okna in mehanizmi za preprečevanje trenutkov neodločljivosti. V zaključku smo razporejevalnik *FWS* primerjali s pravičnim razporejevalnikom in izmerili vpliv različnih parametrov na pravičnost dodeljevanja virov. Rezultati meritev so pokazali, da je vpliv razporejevalnika *FWS* na odzivne čase poslov zanemarljiv, a po drugi strani dosega bistveno pravičnejše in enakomernejše dodeljevanje virov, kakor je to prisotno v obstoječem razporejevalniku.

Ključne besede:

velike zbirke podatkov, MapReduce, Apache Hadoop, pravični razporejevalnik, uteženo pravično razporejanje, razporejanje s časovnim oknom, razporejevalnik *FWS*

Abstract

An ever-growing amount of data, its variety and demand for faster and more responsive systems are some of the most common challenges in the emerging field of data intensive and distributed computing.

The diploma thesis discusses the MapReduce programming model and its implementation in Apache Hadoop framework that enables us to manage and process distributed data intensive applications. The thesis describes several implementations of MapReduce model with core emphasis on Apache Hadoop project and its scheduling, which is one of the key component of the framework. In the second part of the thesis we introduced several types of schedulers, which we grouped according to their characteristics and common uses. The main focus of this part is to present the Fair Scheduler which serves as a basis for the implementation of the new FWS scheduler. The aim of the new scheduler is to improve job response times and their fair execution with an introduction of a time window structure and tie breaking mechanisms. The thesis concludes with a comparison of the two schedulers and the impact of various parameters on slots allocations for FWS scheduler. Results have shown that the effect of FWS scheduler on job response times is negligible, whereas on the other hand it achieves a significant improvement in a more fair and equal allocation of resources.

Key words:

Big data, MapReduce, Apache Hadoop, Fair Scheduler, weighted fair sharing, fair window scheduling, FWS scheduler

Poglavje 1

Uvod

Naraščajoče količine podatkov, njihova raznolikost in zahteve po odzivnejših sistemih narekujejo razvoj novih tehnologij in modelov za njihovo učinkovitejše upravljanje in obdelavo. Z nižanjem cen pomnilniških enot in razvojem novih porazdeljenih podatkovnih modelov so poleg običajnih relacijskih baz na voljo tudi tehnologije, ki omogočajo zajem, hranjenje in obdelavo velikih zbirk podatkov, ki so bili zaradi svoje velikosti in kompleksnosti običajno izpuščeni. Izraz za množico problemov, ki se ukvarjajo s podatki, katerih ni mogoče učinkovito procesirati s tradicionalnimi orodji, imenujemo tudi *Big Data*.

Porast količine podatkov lahko pripišemo razvoju pomnilniških enot in dostopnosti novih tehnologij, ki podjetjem nudijo dodaten vpogled v poslovanje in predstavljajo cenejše ter prožnejše upravljanje njihovih podatkov. Izvor podatkov je v veliki meri povezan tudi z internetnimi storitvami, kjer običajno nastajajo pri uporabi in aktivnostih spletnih ter mobilnih storitev, kot so socialna omrežja, spletni dnevniki ali sporočilni sistemi. Z naraščajočim številom mobilnih in medsebojno povezanih merilnih naprav velik del predstavljajo samodejno ustvarjeni podatki. Slednji so prisotni tako pri znanstvenoraziskovalnem delu (npr. *LHC*) kot v avtomobilski, energetski, finančni in proizvodni industriji. Pomemben del predstavljajo tudi podatki iz poslovanja podjetij, ki izvirajo iz sistemov *CRM* in *ERP*, transakcij spletnih trgovin ali knjigovodskih izkazov [10].

Značilnosti, s katerimi se soočajo rešitve za velike zbirke podatkov, so poleg njihove količine tudi raznolikost in zahteve po hitrosti v njihovem zajemanju in obdelavi. Raznolikost se kaže predvsem v sposobnosti hranjenja različnih oblik zapisov podatkov in zmožnosti njihove skupne obdelave ob upoštevanju časovnih zahtev podanih aplikacij. V zdravstvu lahko takšen primer predstavlja napoved učinkovitosti posameznega zdravila glede na podatke različnih vi-

rov, kot so zapisi zdravstvenih kartotek in laboratorijskih testiranj. Drugi primer s časovno bolj omejenimi zahtevami lahko predstavlja sporočilno-nadzorni sistem, ki glede na zgodovino dnevniških zapisov in trenutnega stanja sistema poskuša predvideti njegove morebitne izpade.

Cilj diplomskega dela je predstavitev programskega modela *MapReduce* v odprtokodni implementaciji ogrodja *Apache Hadoop*, ki trenutno predstavlja eno vidnejših ogrodij za hranjenje in obdelavo velikih zbirk podatkov. Pomemben dejavnik pri hitrosti obdelave in odzivnosti sistema predstavlja razporejanje opravil. Njegovo delovanje v omenjenem okolju smo preučili v poglavju 4 ter si kot cilj zadali nadgradnjo obstoječega pravičnega razporejevalnika z namenom izboljšanja odzivnosti sistema in pravičnejšega dodeljevanje virov med uporabnike. V razporejevalnik smo vpeljali zamisel razporejanja s pomočjo pravičnega okna, ki ima podobne lastnosti kot strežniki z odlogom v sistemih v realnem času. Podrobnosti novega razporejevalnika smo opisali v poglavju 5.

Poglavje 2

MapReduce

MapReduce je programski model in izvajalno ogrodje za porazdeljeno obdelavo velikih zbirk podatkov, ki se nahajajo v gručah običajnih računalnikov v obliki strukturiranih ali nestrukturiranih podatkov. Razvili so ga pri podjetju *Google* [17, 35].

V vzporednih in porazdeljenih sistemih je običajno potek procesiranja, delitev podatkov in obravnava napak prepuščena uporabniku [27]. Naloge, s katerimi se mora običajno uporabnik soočati, predstavljajo sinhronizacijo dostopov do skupnih podatkovnih struktur, preprečitev smrtnih objemov ali zaznavanje tekmovanja za vire. Model *MapReduce* v primerjavi z omenjenimi sistemi ponuja višjo raven abstrakcije, s čimer se prikrijejo podrobnosti sistema, ki niso neposredno vezane na reševanje problema. Slednje zagotavlja izvajalno ogrodje, ki poskrbi za razdelitev vhodnih podatkov, razporejanje opravil delovnim vozliščem, odpravljanje napak in komunikacijo med vozlišči. Za ogrodje veljajo tudi naslednje lastnosti:

- prisotnost vzporednega kot tudi zaporednega izvajanja [25],
- prenos računanja k podatkom in prisotnost algoritmov za izvajanje podatkovno lokalnih opravil,
- linearna podatkovna in računska razširljivost z dodajanjem novih vozlišč,
- mehanizmi za samodejno razširljivost in obvladovanje napak,
- združenost podatkovnega in računskega dela na istih vozliščih,
- neodvisnost podatkovnega in računskega dela z možnostjo menjave porazdeljenega datotečnega sistema [4] in

- prilagodljivost izvajanja in razporejanja opravil tudi na običajnih računalnikih in virih v oblaku (Amazon EC2).

V nadaljevanju poglavja je predstavljen izvor in opis programskega modela ter izvajalno ogrodje, ki skrbi za koordinacijo in izvajanje aplikacij *MapReduce*. V zaključku poglavja je opisanih nekaj njegovih konkretnih implementacij.

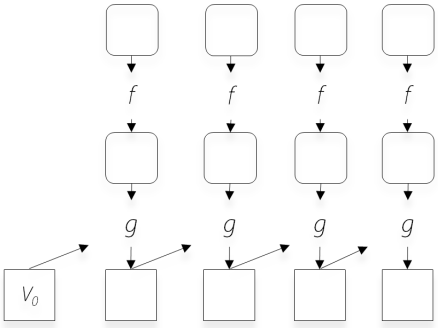
2.1 Programski model MapReduce

Za funkcijske programske jezike je značilno, da ne temeljijo na izvajanju zaporedij ukazov, kakor je to pri imperativnih jezikih, vendar izračune evaluirajo s pomočjo funkcij. Za slednje velja, da so enakovredne drugim vrednostim jezika kar omogoča, da jih lahko sprejemamo kot argumente in vračamo kot rezultate drugih funkcij. Funkcije, ki kot argument sprejemajo in vračajo funkcije tipa prvega reda imenujejo višjenivojske funkcije (angl. *higher order functions*) [15, 23]. Primer omenjenih funkcij v funkcijskih jezikih predstavljata konstrukta *map* in *fold*.

Za funkcijo *map* velja, da kot argument sprejme seznam vrednosti in funkcijo tipa prvega reda f, ki jo neodvisno aplicira na vsak element seznama in vrača nov seznam njihovih pripadajočih vrednosti. Učinkovitost vzporednega izvajanja je tako omogočena s primerno porazdelitvijo seznama med vozlišča gruče.

Za funkcijo *fold* je značilno, da elemente seznama rekurzivno združuje glede na vhodno funkcijo. Funkcija *fold* poleg vhodnega seznama in funkcije g potrebuje tudi začetno vrednost v_0, ki skupaj z začetnim elementom seznama tvori prvi vmesni rezultat. Slednji se skupaj z naslednjim elementom vhodnega seznama uporabi kot argument nove iteracije funkcije g. Postopek se rekurzivno ponavlja do zadnjega elementa seznama, kjer rezultat predstavlja končno vrednost funkcije *fold*. Vzporednost izvajanja funkcije *fold* je odvisna od asociativnih in komutativnih lastnosti vhodne funkcije g. Predstavitev skupnega delovanja obeh opisanih funkcij je povzeta na sliki 2.1.

Podoben koncept izvajanja kot ga vidimo na sliki 2.1 je v modelu *MapReduce* prisoten v izvajanju funkcij ***map*** in ***reduce***, ki ju določi uporabnik. Vhod v omenjeni funkciji ne predstavljajo več seznami, temveč pari oblike $\langle key, value \rangle$, ki so ustvarjeni iz začetne množice podatkov. Tipa vrednosti *key* in *value* sta definirana s strani uporabnika in lahko opisujeta tako primitivne tipe kot tudi kompleksnejše podatkovne strukture. V preprostem primeru štetja besed so lahko vrednosti *key* enake besedam, medtem ko vrednosti *value* predstavljajo njihovo frekvenco.



Slika 2.1: Primer uporabe višjenivojske funkcije *map* in *fold*. *Map* kot argument sprejme funkcijo f in jo aplicira na vse elemente seznama, medtem ko *fold* s funkcijo g rekurzivno združuje rezultate funkcije f in predhodne vmesne rezultate.

Funkcija *map* se aplicira na vsak vhodni par $\langle key, value \rangle$ in tvori množico novih vmesnih parov. Vhod v funkcijo *reduce* je par s ključem in množico združenih vrednosti iz vmesnih parov, ki pripadajo istemu ključu. Po zaključku izvajanja izhod funkcije *reduce* navadno predstavlja množica parov ključa in združenih vrednosti ali pa končna vrednost za vhodni ključ. Če povzamemo zgornji opis imata funkciji sledeče tipe:

$$map : \langle k_i, v_i \rangle \rightarrow \langle k_a, v_a \rangle , ... , \langle k_b, v_b \rangle \tag{2.1}$$

$$reduce : \langle k, \{v_1, ..., v_m\} \rangle \rightarrow \langle k, v_{k_1} \rangle , \langle k, v_{k_2} \rangle , ... \tag{2.2}$$

kjer k_i, k_a, k_b in k predstavljajo različne ključe, medtem ko v-ji predstavljajo njihove pripadajoče vrednosti. Funkciji se v ogrodju izvajata za vsako izmed opravil *map* in *reduce*.

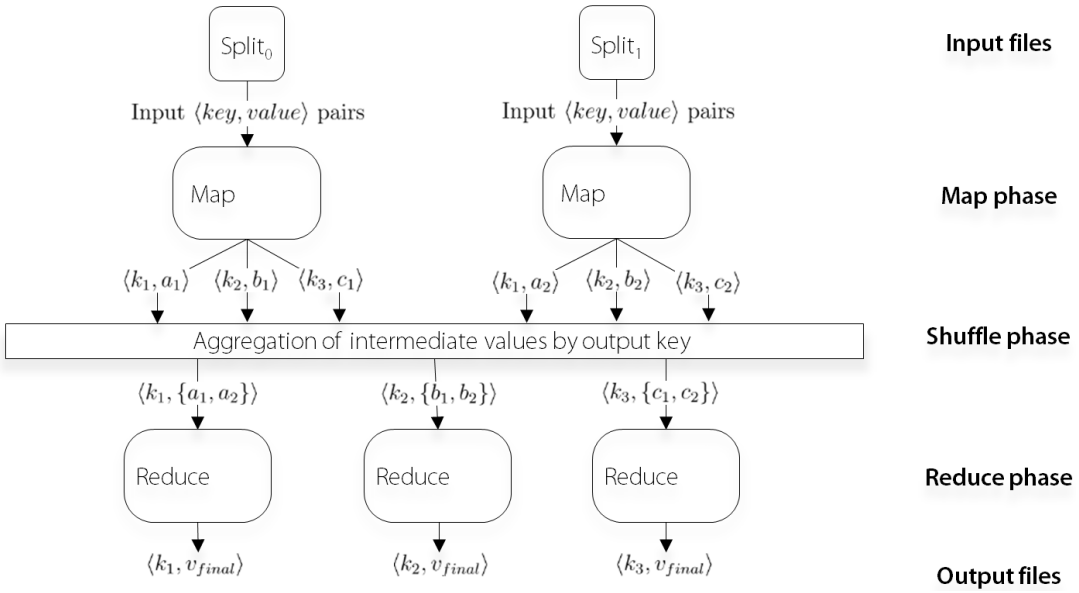
Opisan model je učinkovit predvsem pri reševanju problemov, ki jih lahko brez večjih težav razdelimo na manjše podprobleme in vzporedno izvajamo v gruči več vozlišč (angl. *embarrassingly parallel problems*.)

2.2 Izvajalno ogrodje

Izvajalno ogrodje modela *MapReduce* je sestavljeno iz stopenj ***map***, ***shuffle*** in ***reduce*** (slika 2.2). Ogrodje pred pričetkom izvajanja razdeli nabor podatkov za obdelavo v več razdelkov (angl. *splits*), ki so prisotni na različnih lokacijah gruče. Za vsak razdelek se ustvari novo opravilo *map*, katerega izvajanje je

določeno z istoimensko funkcijo. Ob začetku izvajanja opravila se vsebina razdelka, ki ga opravilo obdeluje pretvori v začetne pare $\langle key, value \rangle$. Slednji predstavljajo vhod v uporabniško funkcijo opravila *map*, ki glede na njeno definicijo tvori množico novih vmesnih parov. Parom je nato na podlagi ključa določeno opravilo *reduce*, kjer se bodo kasneje izvajali.

V primeru več vzporednih opravil *reduce* se vmesni pari razdelijo v skupine, ki jih določa delitvena funkcija uporabnika. Število skupin je enako številu opravil *reduce*, vsaka pa gre v izvajanje k natanko enemu izmed opravil *reduce*. V stopnji *shuffle* ogrodje poskrbi, da so opravilu *reduce* na voljo vse dodeljene skupine in združi vrednosti parov, ki pripadajo istemu ključu. Pred pričetkom stopnje *reduce* morajo biti opravila v stopnji *map* zaključena.[1]



Slika 2.2: Prikaz izvajanja opravil *map* in *reduce*.

Vhod v funkcijo opravila *reduce* je par s ključem in pripadajočim seznamom vrednosti. Običajno funkcija izvede agregacijo nad seznamom ključa in vrne njihovo manjšo množico ali pa končno vrednost za podani ključ. Posledica tega je, da lahko funkcija *map* vhodne ključe poljubno spreminja, medtem ko pri funkciji *reduce* to ni mogoče. Potek izvajanja opravil je povzet na sliki 2.2.

[1]Kopiranje vmesnih parov iz zaključenih opravil se sicer lahko prične tudi prej, vendar mora izvajanje opravila *reduce* počakati na vse pare iz opravil *map*.

Pred pričetkom stopnje *shuffle* lahko izvajanje in obremenjenost omrežja izboljšamo s predhodnim lokalnim združevanjem. Omenjeno nalogo opravlja združevalnik (angl. *combiner*), ki je v ogrodju *Hadoop* opisan v poglavju 3.5. Pomembno vlogo pri odzivnosti sistema ima tudi razporejevalnik opravil. Razporejanje in dodeljevanje opravil poteka v stopnjah *map* in *reduce*, za opravila pa velja, da se izvajajo vzporedno in so med seboj neodvisna.

Podrobnosti modela *MapReduce* in njegova formalna definicija so opisani tudi v člankih [25, 26].

Primer algoritma WordCount

Kot preprost primer izvajanja aplikacije v okolju *MapReduce* navajamo štetje besed v množici dokumentov (Algoritem 1). Opravila *map* se izvajajo vzporedno na več vozliščih, vsa pa so določena z enako funkcijo *map*, katere število klicev je odvisno od velikosti razdelka ter posledično števila vseh njegovih parov.

Vhod v funkcijo posameznega opravila *map* je par $\langle idDoc, docLine \rangle$, kjer je *idDoc* enolična oznaka dokumenta, medtem ko *docLine* predstavlja eno izmed njegovih vrstic. Ker nas v tem primeru ne zanima koliko besed je v posameznem dokumentu, ključ *idDoc* zanemarimo. V tretji vrstici algoritma opravilo za vsako besedo oznani par $\langle word, 1 \rangle$, kar pomeni, da smo v vrstici našli eno pojavitev omenjene besede.

Algoritem 1 Primer algoritma WordCount

```
1:  function Map(idDoc, docLine)
2:      for each word in docLine do
3:          EmitIntermediate(word,1)
4:      end for
5:  end function
6:
7:  function Reduce(word, occurenceList)        → list of word occurrences
8:      count = 0
9:      for each v in occurenceList do
10:         count += v;
11:     end for
12:     Emit(word, count)
13: end function
```

Po tvorjenju vmesnih parov iz opravil *map* ogrodje poskrbi, da so vsi pari, ki pripadajo isti besedi, na voljo skupnemu opravilu *reduce*.[2] Vhod v funkcijo opravila predstavlja par, ki vsebuje besedo in združeni seznam vseh njenih pojavitev (vrstica 7).

V deseti vrstici lahko vidimo, da funkcija le iterira po seznamu pojavitev besede in sešteje njihove vrednosti. V dvanajsti vrstici se končna vrednost za podani ključ zapiše v izhodno datoteko. Končni seznam besed in njihovih seštevkov sestavlja množica vseh izhodnih datotek opravil *reduce*.

2.3 Primeri implementacij modela MapReduce

Model *MapReduce* je zaradi svoje preprostosti, odpornosti na napake in enostavne razširljivosti kapacitet trenutno eden bolj prepoznavnih modelov za izvajanje podatkovno intenzivnih aplikacij. V sledečem razdelku je naštetih nekaj njegovih implementacij na primeru ogrodij za podatkovno intenzivne in porazdeljene aplikacije. Poleg omenjenih lahko implementacije modela zasledimo tudi pri izračunih na večjedrnih procesorjih (QT Concurrent), grafičnih procesorjih (Mars, MapReduce for the Cell B.E. Architecture) ali pa sistemih z arhitekturo deljenega pomnilnika (Phoenix).

Google MapReduce je programski model in hkrati izvajalno ogrodje, ki sta ga razvila avtorja Jeff Dean in Sanjay Ghemawat pri podjetju *Google* iz potrebe po obvladovanju in obdelavi nenehno naraščajoče količine podatkov. Njegova implementacija je vsebovana v zbirki knjižnic oziroma v izvajalnem ogrodju, napisanem v jeziku *C++*. Značilnosti ogrodja so preprosta uporaba, enostavna razširljivost računske moči in samodejna porazdelitev izvajanja. Funkcije, ki jih želi uporabnik aplicirati nad podatki, se v izvajanje podajo v obliki poslov z uporabo različnih programskih vmesnikov (*C++, Python, Java*).

Kot zanimivost navajamo podatke, ki nakazujejo naraščajoči trend podatkov in njihove obdelave pri podjetju *Google* [16]. V prvi in drugi vrstici slike 2.3 lahko zasledimo porast izvajanja uporabniških poslov *MapReduce* in njihovih povprečnih izvajalnih časov, medtem ko je v tretji vrstici prikazano naraščanje števila vozlišč namenjenih njihovemu izvajanju. Zadnji vrstici opisujeta trend rasti vhodnih podatkov in količino novih, ki se pri tem ustvarijo. Skupno so podatki leta 2010 v podjetju presegali 1000PB.

[2] V našem primeru je opravilu *reduce* za vsako besedo vrstice poslan nov par, kar pa lahko zaradi velike količine besed slabo vpliva na obremenjenost omrežja. V poglavju 3.5 je predstavljena možnost združevanja parov pred njihovim nadaljnjim pošiljanjem.

	Aug, '04	Mar, '06	Sep, '07	May, '10
Number of jobs	29K	171K	2,217K	4,474K
Average completion time (secs)	634	874	395	748
Machine years used	217	2,002	11,081	39,121
Input data read (TB)	3,288	52,254	403,152	946,460
Output data written (TB)	193	2,970	14,018	45,720

Slika 2.3: Trend naraščanja podatkov in števila obdelav pri podjetju *Google*.

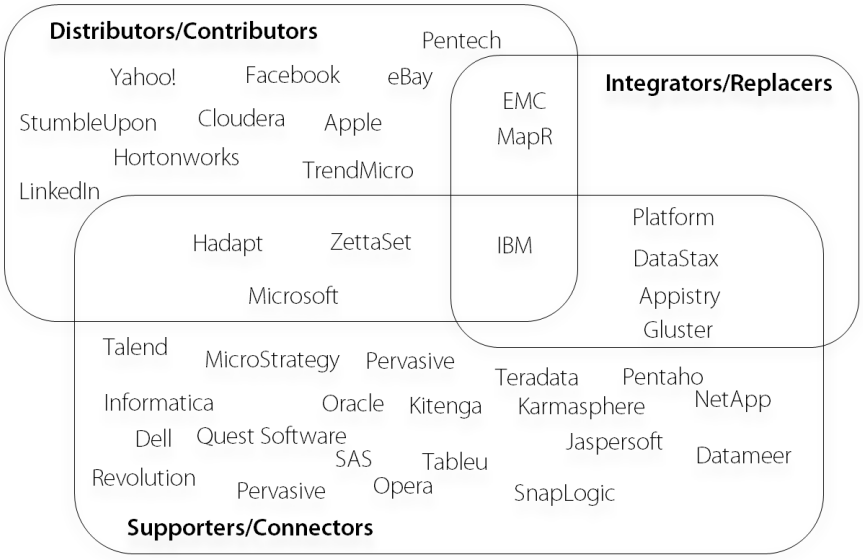
Podjetje se je odločilo, da ogrodje ne prispeva k odprtokodni skupnosti, kljub temu pa je leta 2004 objavilo članek [39], ki je prispeval k razvoju številnih implementacij modela *MapReduce*.

Apache Hadoop je odprtokodna in ena izmed bolj razširjenih implementacij *Googlovega* modela *MapReduce*, ki s pripadajočim porazdeljenim podatkovnim sistemom *HDFS* omogoča porazdeljeno procesiranje in hranjenje velike količine podatkov v gručah računalnikov [5]. Primeren je predvsem za izvajanje podatkovno intenzivnih aplikacij in daljših paketnih obdelav (angl. *batch processing*).

Začetki razvoja segajo v leto 2005 s projektom iskalnika *Nutch*, katerega idejni vodja in razvijalec je Doug Cutting. V naslednjem letu je projekt prešel pod okrilje podjetja *Yahoo!*, kjer trenutno gostijo *Hadoop* na več kot 43.000 vozliščih za namene indeksacije strani, oglaševanja in optimizacije vsebin [13,30]. V letu 2010 je njihova največja gruča štela 4000 vozlišč. Drugo največjo gručo s 21PB prostora na skupno 2000 vozliščih so imeli v istem letu pri podjetju *Facebook*. Posamezno vozlišče je vsebovalo 32GB delovnega pomnilnika, 12TB prostora in od 8 do 16 procesorskih jeder.

Zgoraj omenjeni podjetji sta poleg podjetij *Cloudera*, *Hortonworks* in *StumbleUpon* glavni razvijalki in podporniki projekta *Apache Hadoop* [31]. Razširjenost njegove uporabe lahko vidimo na sliki 2.4.

Poleg različice *Apache Hadoop* smo zasledili tudi njene komercialne izpeljanke, ki se od osnovne razlikujejo predvsem v orodjih za upravljanje in nadzor gruče, izpopolnjeni avtentikaciji in načinu dostopa, podpori za statistična in analitična orodja ter drugih dodatkih. Opise lastnosti in podrobnosti lahko najdemo v naslednjih bolj prepoznavnih različicah: *Amazon Elastic MapReduce*, *Cloudera CDH*, *MapR*, *EMC Greenplum HD* in *IBM InfoSphere BigIn-*



Slika 2.4: Seznam podjetij v povezavi s projektom *Hadoop* [12].

sights. Podrobnejši opis implementacije modela *MapReduce* v okolju *Apache Hadoop* se nahaja v poglavju 3.1.

Disco je ena izmed mnogih različic implementacije modela *MapReduce*, katerega so razvili v raziskovalnem centru podjetja *Nokia*. Ogrodje poleg procesnega dela vsebuje tudi porazdeljen datotečni sistem *DDFS*, ki je tesno povezan z izvajanjem poslov *MapReduce* [6].

Osnove ogrodja so napisane v funkcijskem jeziku *Erlang*, ki je namenjen razvoju robustnih, porazdeljenih in napakam odpornih aplikacij za izvajanje ohlapno v realnem času [6, 7]. Uporabniki posle *MapReduce* navadno pišejo v jeziku *Python*.

Sector/Sphere je odprtokodni projekt, katerega začetek razvoja sega v leto 2006 na univerzi Illinois pod okriljem *NCDM* (*National Center for Data Mining*). Namen projekta je upravljanje in obdelava velikih količin podatkov med različnimi gručami računalnikov, ki so med seboj lahko povezane prek prostranih omrežij (angl. *WAN*) [21]. Projekta sta v veliki meri podobna modelu *MapReduce*, čeprav ju zaradi njihovih razlik ne moremo v celoti enačiti s tem modelom. Podobno kot pri ostalih ogrodjih za porazdeljeno procesiranje je tudi v tem projektu prisoten porazdeljen podatkovni sistem *Sector* in podat-

kovno procesno ogrodje *Sphere*. Glavna motiva razvoja sta bila poenostavitev porazdeljenega programiranja in porazdelitev ter zbiranje podatkov med več lokacij podatkovnih gruč.

Poizvedovanje v ogrodju *Sphere* je mogoče z uporabniško definiranimi funkcijami (angl. *UDF*) ali preko vmesnika za poizvedovanje v modelu *MapReduce*. Projekt je zasnovan v jeziku *C++*, dopolnjuje pa ga lasten komunikacijski protokol *UDT*, ki skupaj z optimiziranim pretokom podatkov predstavlja alternativo ogrodjem, kot je *Hadoop*. Avtorji navajajo, da je njegova uporaba primerna predvsem za obdelavo podatkov razdeljenih na več ločenih lokacij ter za izvajanje analitike podatkovno zahtevnih aplikacij. Projekta se lahko uporabljata tudi kot porazdeljen podatkovni sistem in napredna platforma za souporabo podatkov.

Microsoft Dryad je splošnonamensko izvajalno ogrodje namenjeno izvajanju podatkovno porazdeljenih aplikacij. Značilnosti ogrodja izhajajo iz treh računskih modelov za porazdeljeno procesiranje: senčilni jeziki za izvajanje na grafično procesnih enotah, *Googlov* model *MapReduce* in porazdeljene podatkovne baze [24]. Potek in izvajanje programa predstavlja aciklični graf, ki vsebuje vozlišča z opisi izvajanj in povezave, ki določajo način njihove komunikacije. Za povzporejanje skrbi razporejevalnik, ki posamezno vozlišče grafa dodeli v izvajanje več procesorskim jedrom znotraj enega računalnika ali množici večih računalnikov gruče. V primerjavi z ostalimi modeli *MapReduce*, ogrodje poleg opisa in poteka izvajanja omogoča vozliščem poljubno število vhodov in izhodov. Modeliranje grafa in opisi izvajanj potekajo z namenskim jezikom *DryadLINQ* napisanem v jeziku *C++*.

Stratosphere je odprtokodno ogrodje, ki ponuja posplošitev modela *MapReduce* in vključuje dodaten nabor višjenivojskih funkcij, ki so v modelu *MapReduce* težje opredeljive [11]. Izvajalno okolje *Nephele* je podobno kot v ogrodju *Dryad*, medtem ko je posplošitev modela povzeta v programskem modelu *PACT*. Poizvedovanje poteka v deklarativnem jeziku podobnemu *SQL*.

Model temelji na strukturah *PACT* (*Parallelization Contracts*), katerih povzporejanje zagotavlja izvajalno ogrodje. Vsak *PACT* vsebuje vhodno in po potrebi izhodno višjenivojsko funkcijo (*Input/Output Contract*), ki kot parametre izvajajo uporabniško definirane funkcije prvega reda. V modelu *MapReduce* tovrstni funkciji predstavljata *map* in *reduce*. Razširitev modela *MapReduce* je zajeta v dodatnih višjenivojskih funkcijah, kot so *cross*, *match* in *cogroup*, ki se izvajajo na dveh ali več vhodih. Funkcije se za razliko od modela *MapReduce* ne izvajajo nad pari oblike $\langle key, value \rangle$ ampak nad n-tericami

podatkov. Izvajanje v modelu *MapReduce* poteka po principu cevovoda v stopnjah *map-shuffle-reduce*, medtem ko je v opisanem modelu z uporabo struktur *PACT* možno opredeliti zahtevnejše pretoke izvajanj. Izvajalne načrte in strategijo povzporejanja z ozirom na majhno pretočnost podatkov opravlja prevajalnik.

Primeri komercialnih rešitev

Aster Data predstavlja eno trenutno vodilnih ponudnikov analitičnih rešitev za obdelavo velikih količin podatkov. Produkt *Teradata* je osnovan na patentirani tehnologiji *SQL/MapReduce*, ki je vključena v porazdeljeni podatkovni gruči *Aster nCluster* [20]. Poizvedovanje omogoča pisanje uporabniško definiranih funkcij, ki jih lahko zasledimo tudi v relacijskih bazah. Izvajalno ogrodje poizvedovanja temelji na principu *MapReduce*.

GridGain ponuja odprtokodno in komercialno rešitev za obdelavo velikih količin podatkov ohlapno v realnem času. Sistem vsebuje podatkovno in računsko gručo, ki je zgrajena po modelu *MapReduce*. Posebnost podatkovne gruče je prisotnost porazdeljenega predpomnjenja (angl. *distributed caching*) s katerim se zagotavlja visoko razpoložljivost, predvsem pa krajšo odzivnost izvajanja opravil. Integracija podatkov je možna s sistemi za trajno shranjevanje podatkov ali podatkovnimi skladišči. Takšen primer so relacijske podatkovne baze, sistemi *ERP* ali porazdeljen podatkovni sistem *HDFS*.

LexisNexis Risk Solutions je podjetje, ki se ukvarja z upravljanjem tveganj podjetij s pomočjo preučevanja velikih množic podatkov. V ta namen je razvilo sistem *Thor*, ki je sicer po svojih lastnostih, izvajalnem okolju in zmogljivosti podoben ogrodju *Google MapReduce*, vendar je njegov razvoj potekal neodvisno od modela podjetja *Google* [8]. Značilnost njihove rešitve je prisotnost poizvedovalnega deklarativnega jezika *ECL*, ki je preveden v izvorno kodo jezika *C++* ter porazdeljen na vozlišča gruče. Sistem v trenutni različici omogoča integracijo s porazdeljenim sistemom *HDFS*.

Primeri ogrodij za algoritme z iterativnim izvajanjem

Opisali smo, da je model *MapReduce* učinkovit predvsem pri reševanju problemov, ki so deljivi na manjše neodvisne dele. Ostali problemi, katerih izvajanje konvergira h končni vrednosti v več zaporednih iteracijah, se v okolju *MapReduce* rešujejo z več obhodi. Vsak obhod predstavlja novo iteracijo algoritma,

kjer se vhodni podatki v stopnjo *map* prenesejo s prejšnjega obhoda stopnje *reduce*.

V ogrodju *Hadoop* za izvajanje algoritmov v več obhodih lahko bodisi poskrbi uporabnik bodisi koordinacijo poslov prepusti orodju *Apache Oozie*.

Podpora večobhodnega izvajanja je prisotna tudi v ogrodju **HaLoop**, katerega glavni namen je optimizacija iterativnega izvajanja algoritmov, ki jo skušajo doseči s prilagojenim razporejanjem in mehanizmi predpomnenja (angl. *caching*). Pri slednjem se zmanjšajo aktivnosti branja in obdelave podatkov med zaključkom prejšnjega in začetkom naslednjega obhoda, poleg tega pa se omogoči hiter dostop do podatkov, ki se ohranjajo v iteracijah algoritma [14].

Podobno ogrodje, ki spada v isto področje reševanja problemov in uporablja model *MapReduce*, se imenuje **Twister** [19].

Poglavje 3

Ogrodje Hadoop

V sledečem poglavju je opisana implementacija programskega modela *MapReduce* v ogrodju *Apache Hadoop*. Model smo spoznali v prejšnjem poglavju, njegove značilnosti pa so opisane v delovanju porazdeljenega datotečnega sistema in pregledu izvajalnega ogrodja.

Ogrodje *Apache Hadoop* smo izbrali predvsem zaradi njegovega zanesljivega in preizkušenega delovanja, razširjenosti uporabe in podpore odprtokodne skupnosti.

3.1 Uvod

Ogrodje *Hadoop* je odprtokodna implementacija modela *MapReduce*, ki se uporablja za izvajanje podatkovno intenzivnih in paketnih aplikacij nad velikimi zbirkami podatkov. Le-ti lahko predstavljajo bodisi strukturirane zapise v porazdeljenih podatkovnih bazah bodisi datoteke shranjene v porazdeljenih datotečnih sistemih, kot je npr. *HDFS*. Kljub temu, da je *Hadoop* v celoti napisan v programskem jeziku *Java*, je možno aplikacije *MapReduce* izvajati tudi v skriptnih jezikih z uporabo orodja *Hadoop Streaming*.

Implementacija ogrodja je razdeljena v sklop *Common*, *MapReduce* in porazdeljen datotečni sistem *HDFS*. Slednja sta v ogrodju tesno povezava, a vseeno medsebojno neodvisna. *HDFS* poleg porazdelitve podatkov omogoča tudi njihovo replikacjo, s čimer je v sistemu po eni strani zagotovljena podatkovna redundačnost, po drugi pa optimalnejše razporejanje, ki z upoštevanjem kopij podatkov omogoča krajše odzivne čase poslov (Poglavje 4).

Značilnost okolja je združenost procesnega in podatkovnega dela ter njuna skorajda linearna razširljivost z dodajanjem novih vozlišč. Ker so vozlišča namenjena tako procesiranju kot hranjenju podatkov, se omenjena lastnost

izrablja za izvajanje podatkovno lokalnih opravil, s čimer se zmanjšuje obremenjenost omrežja in izboljšuje odzivnost poslov (Poglavje 4.3.6). Dodajanje in vzpostavitve vozlišč potekajo samodejno.

V ogrodju se napake v izvajanju navadno pojavijo zaradi neuspešno dokončanih opravil ali zaradi odpovedi delovnih vozlišč. V prvem primeru težavo rešuje razporejevalnik, ki poskuša neuspešno opravilo ponovno dodeliti v izvajanje drugim vozliščem. V primeru odpovedi ali počasnosti vozlišča ogrodje samodejno poskrbi za njihovo izolacijo in jim s tem prepreči nadaljnje izvajanje.

V nadaljevanju je opisanih nekaj podprojektov, ki dopolnjuje okolje *Apache Hadoop*. Sledi jim opis porazdeljenega datotečnega sistema *HDFS* in pregled izvajanja modela *MapReduce* v implementaciji ogrodja *Hadoop*.

BigTop projekt je še v začetni razvojni fazi, njegov namen pa je avtomatizacija postavitev in zagotavljanje usklajenega delovanja med projekti v ekosistemu *Hadoop*.

HBase je nerelacijska porazdeljena podatkovna baza v tesni povezanosti z ogrodjem *Hadoop* in njegovim datotečnim sistemov. Njena pomembnejša lastnost je odzivnost pri izvajanju naključnih bralno-pisalnih dostopov in hranjenje ogromnih količin podatkov (milijarde vrstic z milijoni stolpcev).

Mahout je zbirka algoritmov strojnega učenja prilagojena okolju *MapReduce*. Večina algoritmov izhaja iz klasifikacijskega učenja, razporejanja v gruče in področja skupinskega filtriranja.

Pig je platforma za analizo večjih zbirk podatkov, ki skupaj z jezikom *Pig Latin* omogoča tvorjenje *MapReduce* poslov v okolju *Hadoop*. Sintaksa jezika je podobna *SQL*.

Hive je podatkovno skladišče za poizvedovanje in analizo podatkov shranjenih v podprtih datotečnih sistemih ogrodja *Hadoop*. Poizvedovanje poteka z jezikom *HiveQl*, katerega poizvedbe se prevedejo v *MapReduce* posle.

Flume je orodje za učinkovito zbiranje večjih količin dnevniških datotek za kasnejšo obdelavo in skupno hranjenje. Branje lahko poteka iz več virov (datoteke *Log*, paketi *Syslog*, procesi *Unix*), ki so kasneje procesirani s *Hadoopom*

ali shranjeni na skupni lokaciji (*HDFS*, *HBase*). Podobne lastnosti lahko zasledimo v projektu *Apache Chukwa*.

ZooKeeper je centralizirana visoko razpoložljiva storitev za koordinacijo porazdeljenih aplikacij. Storitev ponuja nabor elementov, ki jih aplikacije lahko implementirajo za namene sinhronizacije, informacije o nastavitvah, upravljanje skupin, imenovanje vodje in ostale storitve.

Sqoop je orodje, ki omogoča uvažanje in izvažanje strukturiranih podatkov iz relacijskih podatkovnih baz in podatkovnih skladišč ali baz tipa *NoSQL*.

Avro omogoča serializacijo podatkovnih struktur ali objektov v obliko, ki jo razume ciljni nosilec podatkov. Podatkovni tipi in protokoli so definirani v notaciji *JSON*. Ogrodje omogoča binaren in *JSON* zapis kodiranja. V *Hadoopu* se uporablja kot povezovalni člen za komunikacijo med vozlišči in dostop odjemalcev do storitev gruče. Podobne lastnosti lahko zasledimo v projektu *Apache Thrift*.

3.2 Področja in primeri uporabe

Pri pisanju diplomskega dela smo tovrstno ogrodje zasledili v primerih uporabe naprednejših podatkovnih analiz kot tudi na področju poslovnega obveščanja (angl. *business intelligence*). Kar nekaj primerov smo našli na področju znanstvenega računanja.

Skupna lastnost skorajda vseh primerov je prisotnost velikih količin nestrukturiranih podatkov, ki lahko izvirajo iz več sistemov in katerih obdelava poteka paketno oziroma v intervalih. Zbiranje vzorcev in zakonitosti podatkov poteka običajno s statističnimi modeli in algoritmi podatkovnega rudarjenja (angl. *data mining*). Nekaj primerov njihove uporabe smo povzeli v nadaljevanju:[1]

Modeliranje tveganj (angl. *risk modeling).* Navadno se pri uvajanju ogrodja v novo okolje podatki iz več podatkovnih virov združijo v enega. V primeru bančništva je s centraliziranjem podatkov omogočen natančnejši in

[1]Področja uporabe smo v večini povzeli in dopolnili iz [22]. Konkretni primeri uporabe so bili povzeti po različnih virih omenjenih podjetij.

celovitejši pregled uporabnikovega finančnega stanja in zadovoljstva s storitvami banke, medtem ko je skupinam analitikov omogočeno poizvedovanje na celotnih zbirkah podatkov institucije.

Odkrivanje goljufij in vsiljivih vsebin (angl. *fraud detection*). Ogrodje pri *Yahooju* uporabljajo za odkrivanje neželene pošte, medtem ko ga pri podjetju *Last.fm* poleg glasbenih priporočil uporabljajo tudi za detekcijo neprimernih objav uporabnikov. Na oglaševalskem področju se izkaže koristen pri preprečevanju goljufij avtomatiziranih klikov in s tem povezanih stroškov oglaševanja. V *IBM*-u kot primer uporabe navajajo odkrivanje goljufij glede na zaporedje in lastnosti vzorcev, ki so pridobljeni iz finančno transakcijskih sistemov.

Analize prehajanja uporabnikov (angl. *customer churn analysis*). Telekomunikacijska podjetja z analiziranjem dnevniških zapisov klicev, podatkov socialnih omrežij in ostalih virov lahko opredelijo zadovoljstvo uporabnika in verjetnost njihovega odhoda k drugemu operaterju. Pri kitajskem telekomunikacijskem podjetju CMCC ogrodje dodatno uporabljajo za izboljšavo prodajnih aktivnosti in storitev ter optimizacijo mrežnega delovanja.

Priporočilni sistemi (angl. *recommendation engine*). Tipičen primer implementacije takšnega sistema z zaledjem ogrodja *Hadoop* sta storitvi *People You May Know* podjetja *LinkedIn* in *Frequently Bought Together* podjetja *Amazon*. Napovedovanje skupnih značilnosti se navadno opravljajo s pristopom, ki temelji na opisu vsebine elementov (angl. *content based*) ali s strategijo filtriranja na podlagi medsebojnih značilnosti (angl. *collaborative filtering*).

Usmerjeno oglaševanje (angl. *ad targeting*) je področje spletnega oglaševanja, katerega cilj so postavitve oglasov na podlagi vsebin ciljnih strani (angl. *contextual advertising*) in njihovo prikazovanje, ki se sklada z aktivnostmi ter vedenjskimi značilnostmi uporabnika (angl. *behavioral targeting*). Podjetje *Facebook* nekatere izmed zgoraj naštetih pristopov uporablja nad anonimnimi podatki, in sicer za namene učinkovitejšega oglaševanja in izboljšanja uporabniških storitev. Ogrodje *Hadoop* se dodatno uporablja za spremljanje stanja omrežja s sistemom *ODS* in skupaj s porazdeljeno podatkovno bazo *HBase* predstavlja odziven sporočilni sistem za hranjenje več kot 135 milijard mesečnih sporočil. Ogrodje za namene usmerjenega oglaševanja skupaj z dodatnimi mehanizmi predpomnjenja uporabljajo tudi pri podjetju *AOL* in *Adobe*.

Nadzorno metrični sistemi (angl. *metric systems*) služijo predvsem spremljanju in zaznavanju nenavadnih dogodkov v delovanju sistema. Takšen primer lahko predstavljajo podatki različnih senzorskih mrež ali podatki iz trgovalnih okolij, kjer se z uporabo statističnih modelov spremlja in ugotavlja nenavadne vzorce v aktivnosti trgovanj. Drugi primer takšnega sistema smo zasledili pri francoskemu podjetju *EDF*, ki je eden večjih svetovnih upravljavcev električne energije. Cilj njihove prototipne postavitve okolja *Hadoop* je bilo izvajanje nadzora in analiz ter spremljanje 35 milijonov pametnih uporabniških merilnikov in senzorskih podatkov generatorjev elektro distribucijskih mrež (angl. *smart grids*). Ogrodje se običajno uporablja v večjih podatkovnih centrih, kjer se z analiziranjem velikih količin dnevniških zapisov aplikacij in strežniških sistemov omogoča širši vpogled v delovanje in stanje infrastrukture.

Poslovno obveščanje (angl. *business intelligence*). Ogrodje sicer v osnovi ne ponuja orodij za podporo pri odločanju in upravljanju poslovanja, a kljub temu lahko opravlja vlogo podatkovnega skladišča in procesnega ogrodja v povezavi z drugimi analitičnimi orodji. Integracijo in povezljivost z omenjeno arhitekturo zagotavljajo mnogi večji proizvajalci analitičnih rešitev, kot so *Tableu Software, Talend, MicroStrategy, Pentaho* in *Jaspersoft*. V primeru spletnih prodajaln se lahko orodja uporabljajo za poročanja o prodajnih aktivnostih in zadovoljstvu kupcev, trendih nakupov in planiranju prodajnih zalog.

Iskalni indeksi (angl. *search index*). Pri podjetju *Yahoo!* poleg usmerjenega oglaševanja in detekcije neželenih sporočil ogrodje z uporabo algoritmov strojnega učenja uporabljajo za izgradnjo iskalnega indeksa. Vhod v algoritem predstavlja acikličen graf vseh znanih spletnih strani z njihovimi pripadajočimi podatki. Ogrodje oziroma natančneje *HDFS* lahko služi kot osnova za gradnjo indeksne infrastrukture, ki z upoštevanjem zgodovine iskanj in preferenc uporabnikov omogoča izgradnjo iskalnega indeksa in napovedovanja uporabnikom pomembnih iskalnih rezultatov.

Bioinformatika in biomedicina (angl. *bioinformatics, biomedicine*) je področje, katerih algoritmi vsebuje veliko število podatkovno intenzivnih in večinoma neodvisnih opravil, zaradi česar so primerni za uporabo v okoljih *MapReduce* [9]. V enem izmed največjih znanstvenoraziskovalnih centrov NERSC navajajo uporabo testnega okolja *Hadoop* poleg bioinformacijskih aplikacij tudi za izvajanje podnebnih analiz in za algoritme numerične linearne algebre. Kot primer opisujejo prepoznavanje peptidov iz masno spektralnih podatkov, gručenje proteinskih zaporedij in razcepe QR matrik. Na ogrodju

Hadoop slonijo sledeča bioinformacijska orodja [36]: *CloudBLAST*, *CrossBow*, *Contrail*, *Bowtie*, *Biodoop* in *BioPig*.

3.3 Arhitektura in osnovni gradniki

Izvajanje opravil v okolju *Hadoop* poteka v gruči računalnikov oziroma množici ohlapno povezanih vozlišč. Gručo sestavljajo glavno vozlišče in množica podrejenih delovnih vozlišč, ki so namenjena shranjevanju in procesiranju podatkov.

Glavno vozlišče navadno sestavlja **upravljalec poslov** (angl. *jobtracker*) in **imensko vozlišče** (angl. *namenode*), ki skrbi za upravljanje s podatki in imenskim prostorom datotečnega sistema. Naloga upravljalca poslov je koordinacija in razporejanje opravil podrejenim **delovnim vozliščem** (angl. *tasktracker*), ki so zadolženi za izvajanje opravil in obveščanje upravljalca o njihovem poteku. Obdelovalni podatki se nahajajo na **podatkovnih vozliščih** (angl. *datanodes*).

Problem, katerega želi uporabnik reševati nad vhodnimi podatki, je določen s funkcijama *map* in *reduce*. Funkciji skupaj z informacijo o vhodnih podatkih in ostalih parametrih okolja predstavljajo **posel** (angl. *job*), ki sestoji iz več manjših enot dela, imenovanih **opravila** (angl. *tasks*), katerih število je določeno glede na delitev nabora vhodnih podatkov.

Ogrodje pred vsakim pričetkom izvajanja poskrbi za delitev vhodnih podatkov na manjše razdelke (angl. *splits*), katerih delitev je odvisna od velikost bloka datotečnega sistema. Za vsakega izmed njih se ustvari novo opravilo *map*, katerega izvajanje je določeno v istoimenski uporabniški funkciji. Zanjo velja, da se uporabi na vsakem zapisu razdelka, ki ga opravilo trenutno obdeluje.

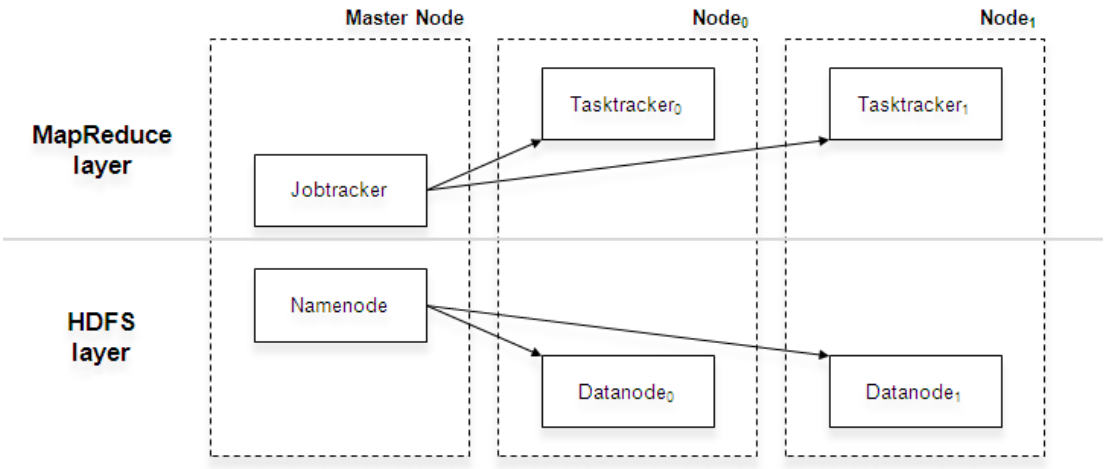
Če povzamemo, je posel sestavljen iz več enakih opravil *map*, katerih število je določeno z velikostjo bloka datotečnega sistema, medtem ko je število opravil *reduce* določeno s strani uporabnika.

Izbiro izvajalnega posla in dodeljevanje opravil opravlja komponenta **razporejevalnik** (angl. *scheduler*), ki je del upravljalca poslov. Razporejanje poteka po principu proženja, torej ob zahtevah delovnih vozlišč po novih opravilih.

Dodeljena opravila se izvajajo v navideznih **računskih virih** delovnih vozlišč (angl. *slots*). Njihovo število se določi pred vzpostavitvijo posameznega vozlišča na podlagi zmogljivosti vozlišča oziroma glede na število vsebovanih procesorskih jeder.

Navadno je na vozliščih gruče prisotno tako podatkovno kot tudi delovno

vozlišče, vendar je možna njuna ločitev. Upravljalec poslov in imensko vozlišče običajno zasedata lastno vozlišče na katerem se opravila ne izvajajo. Primer običajne gruče in vozlišč lahko zasledimo na sliki 3.1.



Slika 3.1: Običajen primer gruče s prisotnostjo glavnega in delovno podatkovnih vozlišč z delitvijo na računski in podatkovni sloj.

3.4 Porazdeljen datotečni sistem HDFS

HDFS je porazdeljen datotečni sistem (angl. *Hadoop Distributed File System*), ki ponuja visoko prepustnost podatkov, zanesljivo delovanje in visoko odpornost na okvare. Poleg zagotavljanja visoke prepustnosti sta njegovi pomembnejši lastnosti tudi samodejna zaznava in odprava napak. Sistem je zasnovan za delo z velikimi količinami podatkov, zato je čas, potreben za branje celotnega nabora, pomembnejši od dostopnega časa za branje začetnega zapisa. Primeren je predvsem za uporabo daljših paketnih obdelav, kjer so prisotne velike količine podatkov. Za aplikacije, kjer je odzivnost poizvedb zelo pomembna, avtorji [2] kot primernejšo izbiro navajajo uporabo nerelacijske porazdeljene podatkovne baze *HBase*. Sistem je zasnovan po principu enkratnega pisanja z večkratnim branjem (angl. *write once read many*).

Omenili smo, da je upravljanje podatkov naloga imenskega vozlišča. Slednji skrbi za strukturo podatkovnega sistema, koordinacijo operacij in metapodatke datotek, kot so pravice dostopa in njihove lokacije [38]. Podatki so shranjeni na

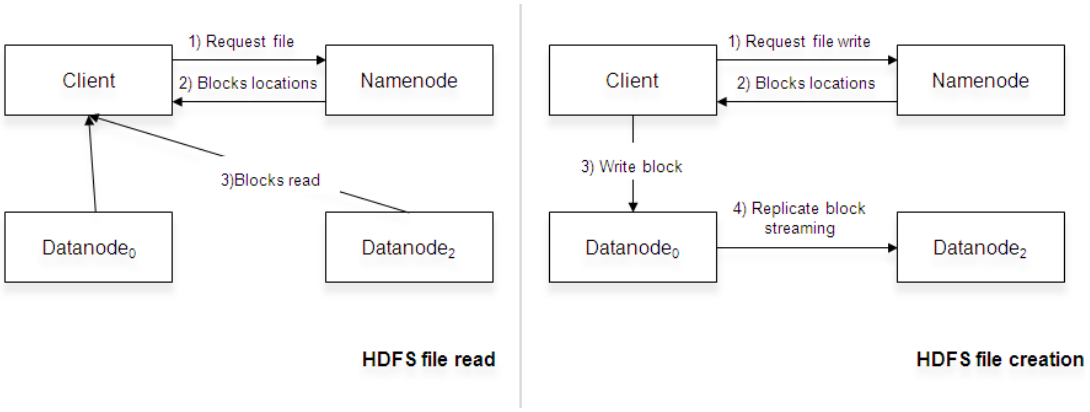
podatkovnih vozliščih, ki so zadolžena za izvrševanje zahtev branja in pisanja s strani imenskega vozlišča ali odjemalca *HDFS*.

Podatki se v datotečnem sistemu shranjujejo v blokih velikosti 64MB. Za vsakega izmed njih navadno obstaja kopija na vsaj še dveh različnih vozliščih. Slednje prispeva k redundačnosti sistema in boljši odzivnosti poslov pri razporejanju opravil (Poglavje 4.3.6).

Branje podatkov poteka s pomočjo odjemalca *HDFS* (angl. *HDFS client*). Začetek branja se prične s poizvedbo odjemalca, ki od imenskega vozlišča pridobi informacije o lokacijah blokov. Za sestavo blokov poskrbi odjemalec, medtem ko branje poteka neposredno iz delovnih vozlišč. Pri izbiri vozlišč imensko vozlišče upošteva tudi topologijo omrežja [38].

Pisanje nove datoteke se prične z zahtevo odjemalca imenskemu vozlišču. Če datoteka še ne obstaja in če ima uporabnik zadostne pravice, se pisanje na podatkovno vozlišče lahko prične. Replikacije blokov se izvršijo po principu cevovoda, kjer se iz začetnega podatkovnega vozlišča posredujejo na preostala. Proces in dodeljevanje lokacij replikacijskih blokov nadzira imensko vozlišče. Bralni dostopi so prikazani na levi strani slike 3.2, medtem ko pisalne lahko zasledimo na desni strani.

V primeru izpada imenskega vozlišča je delovanje sistema onemogočeno. Dodatno vozlišče s kopijo imenskega vozlišča je prisotno od različice *Hadoop 0.23* dalje.



Slika 3.2: Prikaz bralno-pisalnih dostopov v *HDFS*.

3.5 Pregled izvajanja modela MapReduce v ogrodju Hadoop

Opisi izvajalnih stopenj in definiciji funkcij modela *MapReduce* so predstavljeni v prejšnjem poglavju, zato jih bomo tukaj izpustili. Poleg pregleda izvajanja smo v sledečem poglavju navedli še delitveno funkcijo, s katero določamo lokacijo izvajanj parov in možnost optimizacije izvajanja opravil s komponento združevalnika.

Izstavitve poslov opravlja odjemalec (angl. *jobclient*). Slednji za posamezen posel poskrbi za razdelitev vhodnih podatkov na M razdelkov. Razporejevalnik za isti posel sestavi seznam opravil, pri čemer za vsakega izmed razdelkov vhodnih podatkov ustvari novo opravilo. Poleg M opravil *map* se ustvari še R opravil *reduce*, katerih število je opredeljeno v konfiguraciji posla. Navadno se število opravil *reduce* določa glede na vrsto problema in lastnosti gruče, medtem ko je število opravil *map* odvisno od količine vhodnih podatkov in velikosti bloka *HDFS*. Potek izvajanja opravil prikazuje slika 3.3 (sledeči opisi so v skladu s sliko):

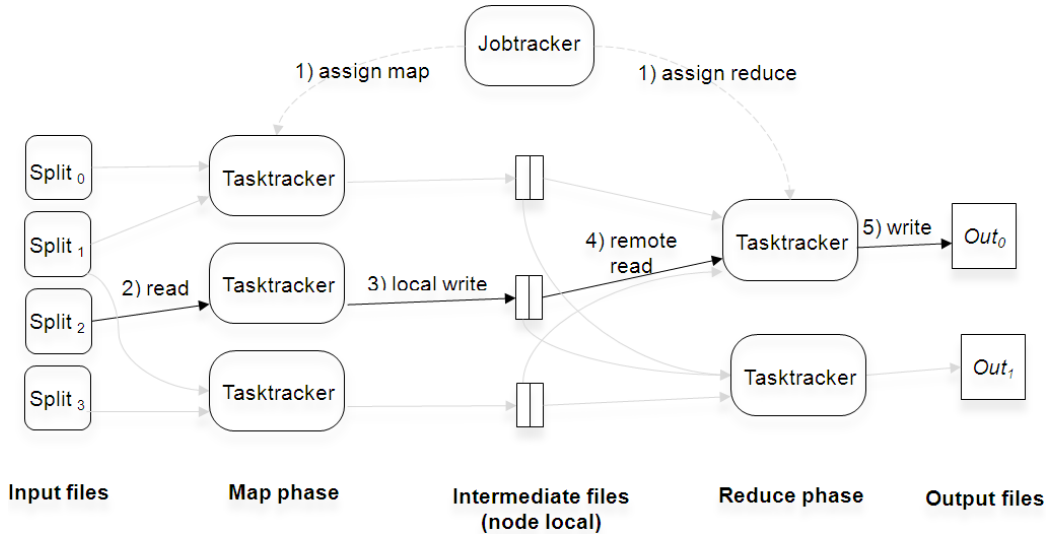
1) Poleg delovnih vozlišč (Tasktracker) je v gruči prisoten upravljalec poslov (Jobtracker), ki z razporejanjem poslov skrbi za dodeljevanje opravil med prosta vozlišča. Komunikacija med delovnimi vozlišči in upravljalcem poteka s statusnimi sporočili (angl. *heartbeats*), preko katerih se vozliščem prenašajo tudi seznami dodeljenih opravil.

2) Vozlišče, kateremu je bilo v izvajanje dodeljeno opravilo *map*, prebere vsebino pripadajočega razdelka (Split) in jo pretvori v ustrezne pare, ki služijo kot vhod v funkcijo *map*. Izhod funkcije so vmesni pari, ki so začasno shranjeni v vmesnem pomnilniku vozlišča (angl. *buffer*).

3) Občasno se vmesni pomnilnik vozlišča shrani na lokalni disk, pri čemer so pari razdeljeni v R skupin glede na ključ in delitveno funkcijo. Lokacije skupin so posredovane glavnemu vozlišču, ki poskrbi, da se v nadaljevanju iste skupine izvajajo na skupnem delovnem vozlišču.

4) Vozlišče, ki mu je dodeljeno opravilo *reduce* in je prejel informacije o lokacijah svojih skupin, izvede branje iz oddaljenih delovnih vozlišč. Po prejetju vseh vmesnih parov iz pripadajočih skupin sledi sortiranje parov glede na njihov ključ. Pri tem se pari z istem ključem združijo, tako da nov par predstavlja ključ in vse njegove vrednosti iz zaključenih opravil *map*. Sortiran seznam parov iz ključev in pripadajočih vrednosti je nato posredovan v funkcijo opravila *reduce*.

5) V zadnji stopnji se pari oblike $\langle k, \{v_1, ..., v_m\}\rangle$ posredujejo v funkcijo *reduce* (enačba 2.2, str. 7). Rezultat funkcije *reduce* je dodan v izhodno datoteko trenutnega opravila.



Slika 3.3: Izvajalno okolje modela *MapReduce* in osnovni gradniki ogrodja *Hadoop*, ki smo jih spoznali v poglavju 4.3.2.

Po uspešno končanem izvajanju je na voljo R izhodnih datotek, ki so lahko bodisi vhod v novo iteracijo bodisi združeni v končni rezultat.

V stopnji *map* lahko zasledimo, da se pari v primeru večjega števila opravil *reduce* razdelijo v več skupin, pri katerih je opravilo *reduce* oziroma vozlišče, kjer se bodo skupine nadaljnje izvajale točno določeno (točka 3). V poslu velja, da so vsi pari z istim ključem zbrani pri istemu opravilu [38, str.194]. Število skupin je določeno s številom opravil *reduce*, medtem ko **delitveno funkcijo** (angl. *partition funcion*) lahko definira uporabnik sam. Določitev skupine para običajno poteka glede na njegov ključ s pomočjo funkcije $hash(key)\ mod\ R$. Pri definiciji nove delitvene funkcije je potrebno paziti na enakomerno delitev prostora ključev.

Pred izvajanjem opravil *reduce* poteka zbiranje vseh izhodnih parov opravil *map*, kar pri njihovem večjem številu lahko predstavlja veliko obremenitev omrežja. *Hadoop* tako pred pošiljanjem parov iz posameznega vozlišča omogoča njihovo združevanje, in sicer tako, da na njih izvede funkcijo določeno v **združevalniku** (angl. *combiner*). Združevanje si lahko predstavljamo podobno kot operacijo *reduce*, ki se izvaja nad izhodnimi pari izbranega opravila

map. V primeru štetja besed iz poglavja 2 bi to pomenilo, da na izhodnih parih opravil *map* opravimo še seštevanje istih besed in jih šele nato pošljemo v izvajanje opravilom *reduce*. S slednjim se bistveno zmanjša obremenjenost omrežja, saj bi v tem primeru namesto vseh pojavitev parov $\langle word, 1\rangle$ vhod v opravilo *reduce* predstavljali le pari $\langle word, freq\rangle$. Uporaba združevalnika se izvaja ob zapisu parov iz vmesnega pomnilnika na lokalni disk (točka 3).

V zgornjem odstavku smo opisali eno izmed možnih izboljšav pretoka podatkov in posledično obremenjenosti omrežja. Kljub temu pa v poglavju nismo omenili, kakšno vlogo ima razporejanje pri odzivnosti opravil. Razporejanje v okolju *Hadoop* je opisano v naslednjem poglavju.

Poglavje 4

Razporejanje v okolju Hadoop

V tem poglavju obravnavamo lastnosti razporejanja in razporejevalnikov v okolju *Apache Hadoop*. Podrobneje je opisan pravični razporejevalnik, katerega strukturo smo uporabili kot osnovo za implementacijo časovnega okna v naslednjem poglavju. V poglavju je poleg skupin razporejevalnikov navedenih še nekaj njihovih bolj znanih predstavnikov, pri katerih lahko zasledimo, kako rešujejo problem pravičnega deljenja računskih virov, izkoriščenosti sistema in krajše odzivnosti poslov.

4.1 Uvod v razporejanje

V prejšnjem poglavju smo omenili, da posel uporabnika sestavljajo *map* in *reduce* opravila, ki jih razporejevalnik dodeljuje med proste računske vire delovnih vozlišč. V *Hadoopu* njihovo dodeljevanje poteka po principu proženja, kjer delovno vozlišče s statusnim sporočilom (angl. *hearbeat*) obvesti upravljavca poslov o prostih virih, številu opravil v izvajanju in ostalih informacijah vozlišča. Sledi izbira posla, ki se pri razporejevalnikih razlikuje glede na cilje razporejanja in parametre okolja.

Po izbiri posla se izvajanje nadaljuje z dodeljevanjem opravil, ki je enako za vse razporejevalnike. Ogrodje jim ponuja le možnost izbire opravila glede na želeno lokalnost podatkov. Z določanjem lokalnosti imajo razporejevalniki tako možnost ali vozlišču dodeliti opravilo, katerega podatki se nahajajo na istem vozlišču ali pa izbrati najbližje nelokalno opravilo. V tem primeru se iz oddaljenega vozlišča prične prenos podatkov, ki so potrebni za njegovo izvajanje.

Pomemben parameter pri izbiri opravil predstavlja replikacijski faktor podatkov, ki poveča verjetnost prisotnosti lokalnih opravil na vozlišču, ki je opra-

vilo zahtevalo. Dodeljena opravila se v izvajanje delovnim vozliščem posredujejo preko povratnih statusnih sporočil.

V zadnjih letih je bilo optimizaciji razporejanja v okolju *MapReduce* posvečeno veliko pozornosti. Kot primer možnosti izboljšav študija [32] povzema številne parametre, kot je razčlenjevanje zapisov, vrsta sortiranja in način V/I branja (neposredno v primerjavi s tokovnim branjem). Dodatno obravnava parametre, kot so število računskih virov na procesorsko jedro, število sočasnih poslov v izvajanju, vpliv lokacije podatkov ter dolžina intervala pri komunikaciji med razporejevalnikom in delovnimi vozlišči.

Večino razporejevalnikov in predlaganih rešitev bi lahko v grobem uvrstili v naslednje skupine:

Optimizacija izvajanja. V to skupino spadajo predvsem razporejevalniki, ki z inteligentnim razporejanjem prispevajo h krajšim odzivnim časov in večji prepustnosti poslov ter vplivajo na večjo učinkovitost gruče. Slednje navadno dosegajo z reševanjem katerega izmed zgoraj naštetih parametrov. Primeri: Fairscheduler [39], LATE [41], Delay [40].

Namensko razporejanje je skupina v katero sodijo razporejevalniki prisotni v gručah, kjer je cilj in uporaba gruče s strani uporabnikov natančno določena. V sistemu so prisotne različne politike uporabe, ki jih razporejevalnik mora upoštevati. Takšen primer je dodeljevanje virov na podlagi ponudbe uporabnika (angl. *cost-based model*). Primeri: Lsched [18], Dynamic Priority Scheduler [34].

Razporejanje v realnem času. Sem sodijo razporejevalniki, ki skušajo zagotovi zaključek izvajanj poslov ohlapno ali strogo v realnem času. Pri oceni trajanja posla upoštevajo različne hevristike izvajanja, ter z upoštevanjem parametrov okolja skušajo napovedati njegov zaključek. Primeri: EDF/MR, EDF/TD [33].

Zgornje razporejevalnike smo poskušali smiselno razvrstiti v skupine, pri čemer je pomembno vedeti, da stroga ločnica med njimi ni začrtana. Takšen primer predstavlja prva skupina, v katero lahko uvrstimo večino naštetih razporejevalnikov. Nekaj bolj razširjenih je opisanih v nadaljevanju.

4.2 Primeri razporejevalnikov

4.2.1 Prioritetni razporejevalnik

V začetnih različicah ogrodja je bilo razporejanje opravil urejeno glede na čas prispetja posla z uporabo razporejanja *FIFO*, ki mu je bila kasneje dodana možnost razporejanja na podlagi prioritete poslov. Razporejevalnik omogoča določiti prioritete poslov v petih različnih vrednostih. Navkljub prioritetam poslov pa pomanjkanje mehanizmov za prekinitev izvajanja opravil lahko privede do njihovega zastoja. Primer, kjer se slednje odraža, je prispetje posla z visoko prioriteto v času izvajanja daljšega posla z nižjo prioriteto. Zaradi pomanjkanja omenjenih mehanizmov se bo posel z višjo prioriteto lahko izvajal šele po zaključku daljšega posla. V različici ogrodja *Hadoop* 0.20.205 ga zasledimo kot privzeti razporejevalnik.

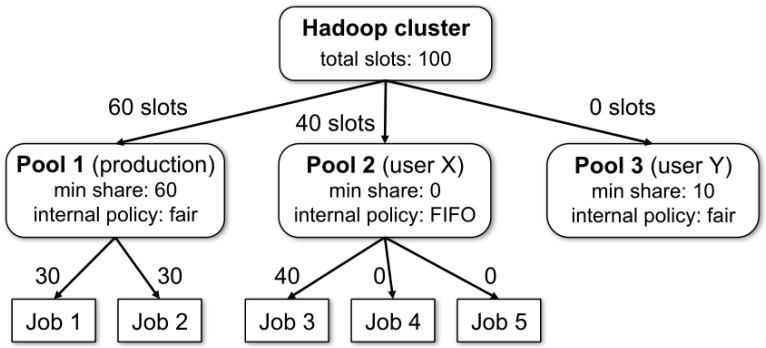
4.2.2 Pravični razporejevalnik

Pravični razporejevalnik (angl. *Fairscheduler*) so razvili v podetju *Facebook* iz potrebe po učinkovitejši delitvi računskih virov med več uporabnikov sistema in izboljšanju odzivnih časov krajših poizvedb. Razporejevalnik računske vire med posle dodeljuje enakomerno, pri čemer je možno določati zajamčeno količino virov, ki je poslom vedno na razpolago (angl. *minshare*). Pravični razporejevalnik temelji na treh lastnostih:

- Vsak uporabnik ali skupina odda svoj posel v bazen, ki mu je določen.
- Vsakemu bazenu je možno določiti minimalni delež virov za izvajanje *map* ali *reduce* opravil, ki so mu vedno na voljo.
- Presežek računskih virov, ki niso namenjeni zagotavljanju minimalnih deležev, se pravično in enakomerno dodeli ostalim bazenom ali poslom, odvisno od tega, kaj razporejamo.

V razporejevalniku je prisotna možnost dvostopenjskega razporejanja. Zunanje predstavlja pravično razporejanje virov med bazeni, medtem ko notranje predstavlja razporejanje znotraj bazena. Na sliki 4.1 je opisan primer hierarhije bazenov in pravičnega dodeljevanja računskih virov.

Praktični del diplomskega dela temelji na implementaciji časovne komponente opisanega razporejevalnika, zato so njegove podrobnosti in algoritmi predstavljeni v naslednjem poglavju.



Slika 4.1: Primer hierarhije razporejanja in pravičnega dodeljevanja virov.

4.2.3 Razporejanje z določanjem kapacitet

Razporejevalnik z možnostjo določanja kapacitet čakalnih vrst (angl. *Capacity Scheduler*) so pri podjetju *Yahoo!* zasnovali z namenom preprostega deljenja virov gruče med različne organizacijske enote podjetja [3].

Funkcionalnosti razporejevalnika so v veliki meri podobne pravičnemu razporejevalniku. Uporabniki posle izstavljajo v čakalne vrste, katerih hitrost izvajanja je odvisna od njihove kapacitete. Slednja predstavlja delež računskih virov gruče, ki so vrstam na voljo za izvajanje njihovih poslov. V primeru nedejavnosti vrste se prosti viri razdelijo med ostale čakalne vrste, katerim je dovoljeno, da za čas izvajanja posla prekoračijo svojo kapaciteto. Kopičenje prostih virov v posamezni vrsti je mogoče preprečiti z določitvijo največjega števila dodatnih virov, ki so vrsti lahko dodeljeni.

Posli se v vrsti razporejajo glede na čas prispetja po principu *FIFO*, medtem ko razporejevalnik omogoča njihovo razporejanje tudi na podlagi prioritet. Ob izstavitvi posla njegova prekinitev ni več možna, kar velja tudi v primeru prihoda posla z višjo prioriteto.

Dodeljevanje računskih virov zgoraj opisanega in pravičnega razporejevalnika poteka glede na trenutno aktivne posle in kapacitete vrste oziroma teže bazena. Oba razporejevalnika ne hranita zgodovine dodeljenih virov za posamezno vrsto oziroma bazen.

4.2.4 Razporejevalnik Lsched

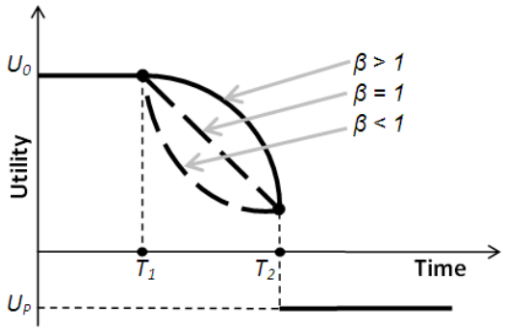
Razporejevalnik *Lsched* (*Learning Scheduler*) rešuje problem razporejanja in nadzora nad sprejemanjem novih poslov (angl. *Task Assignment and Admis-*

sion Control). Pri tem upošteva zadnje odločitve, ki so sprejete na podlagi naivnega Bayesovega klasifikatorja in odloča o sprejetju in dodeljevanju poslov. Klasifikator se uporablja v obeh primerih.

Njegova uporaba je zamišljena predvsem v *SaaS MapReduce* okoljih, kjer se na eni strani upošteva pričakovanja uporabnika na drugi pa zahteve ponudnika takšne storitve [18]. Razmerje med obema je določeno s funkcijo koristnosti $U(t)$, ki predstavlja sklenjen kompromis med obema. Primer takšne funkcije lahko predstavlja ceno, ki jo je uporabnik pripravljen plačati glede na čas zaključka izvajanja posla.

$$U(t) = \begin{cases} U_0 & 0 < t \leq T_1 \\ U_0 - \alpha\,(t - T_1)^{\beta} & T_1 < t \leq T_2 \\ U_P & t > T_2 \end{cases} \tag{4.1}$$

V zgornji enačbi lahko vidimo, da je uporabnik za izvajanje posla z zaključkom do mehkega roka T_1 pripravljen plačati U_0. Z dodatno zakasnitvijo se funkcija zmajšuje vse do striktnega roka T_2. U_P je po roku T_2 lahko tudi negativen, kar prinaša kazen ponudniku storitev. Slika 4.2 prikazuje obnašanje funkcije za različne parametre β.



Slika 4.2: Primer funkcije koristnosti za različne parametre β.

Sprejemanje novih poslov v izvajanje razporejevalnik *Lsched* rešuje z ločeno komponento. Naloga slednje je zagotavljanje, da s sprejetjem novega posla gruča ne bo preobremenjena ter maksimiranje funkcije koristnosti v prid ponudnika gruče. Posel je izbran s funkcijo

$$Selected\,job = max_{j \in Jobs}\,(U_j \cdot P(J = Success|E)))\ , \tag{4.2}$$

kjer E predstavlja trenutne parametre okolja, U_j pa izračunano vrednost na podlagi funkcije koristnosti za posel j. Dogodek $J = Success$ predstavlja

uspešen sprejem dogodka glede na podane kriterije ponudnika. Uspešnost dogodka je pogojena s trenutnimi parametri okolja, sprejete odločitve in rezultat izvajanja pa vplivajo na nadaljnje odločanje. Parametre okolja in podrobnosti algoritma lahko bralec prebere v [18].

Dodeljevanje opravil in klasifikacija poslov poteka podobno kot pri nadzoru sprejemanja poslov. V množici kandidatov za izvajanje se posli najprej označijo kot *dobri* ali *slabi*, kjer prvi ne povzročajo preobremenjenosti gruče. Izmed teh se izbere tistega, ki maksimira funkcijo koristnosti $U(t)$ in verjetnost uspešnosti posla glede na prejšnja izvajanja. Izbira opravila v poslu poteka enako kot pri ostalih razporejevalnikih v ogrodju *Hadoop*.

4.2.5 Razporejevalnik z dinamičnim spreminjanjem prioritet

Razporejevalnik z dinamičnim spreminjanjem prioritet (angl. *Dynamic Priority Scheduler*) omogoča uporabnikom optimizacijo in prilagajanje izvajanja poslov glede na pomembnost in potrebe posameznega posla. Dodeljevanje opravil poteka po tržnem principu [34], kjer ima vsak uporabnik na začetku določeno količino navideznega denarja. Ob izstavitvi posla uporabnik lahko s ceno za računski vir določa hitrost izvajanja posla in tako delež virov, ki mu bodo na voljo. Razporejevalnik izbere posel z največjo hitrostjo izvajanja oziroma tistega, za katerega je uporabnik pripravljen plačati največ v danem intervalu. Postopek izbire je sledeč:

1. Spremenljivka p predstavlja skupno ceno gruče in je seštevek vseh ponujenih cen uporabnikov v določenem intervalu, $p = \sum s_i$.

2. Vsakemu uporabniku i je dodeljen delež gruče glede na ceno, ki jo je pripravljen plačati. Število prejetih računskih virov za uporabnika i je $(s_i/p) \cdot c$, kjer je c število računskih virov, ki so v gruči na voljo.

3. Na koncu se vsakemu uporabniku iz proračuna b odšteje vrednost porabljenih računskih virov $s_i \cdot u_i$, kjer je s_i cena za računski vir, ki jo je uporabnik pripravljen plačati, u_i pa dejansko število porabljenih virov.

4.2.6 Razporejevalnik LATE in razporejevalnik z zakasnjenim izvajanjem

Spekulativno izvajanje razporejevalnika LATE (Longest Approximate Time to End) in razporejevalnik z zakasnjenim razporejanjem (angl. *Delay Scheduler*)

sta opisana v člankih avtorja Matea Zaharie in ostalih [40, 41]. Po navedbah avtorja so mehanizmi spekulativnega izvajanja implementirani v verzijah *Hadoop* 0.21, 0.22 in 0.23, medtem ko smo v verziji 0.20.205 in 1.0.1 zasledili, da so implementirani le delno. Algoritem zakasnjenega razporejanja je prisoten še v verziji 0.20.205, zato je predstavljen v poglavju 4.3.6.

Razporejevalnik *LATE* v heterogenih okoljih izboljšuje odzivne čase[1] krajših poizvedb s spekulativnim izvajanjem opravil, ki so izbrana glede na največji preostanek časa do njihovega zaključka. Slednja so izbrana zato, ker je zanje najverjetneje, da se bodo izvedla v krajšem času kot prvotna opravila in tako prispevala h krajši odzivnosti poslov. Pri oceni preostalega časa do zaključka opravila razporejevalnik ne upošteva le deleža opravljenega dela (angl. *progress score*), temveč uvaja tudi preprosto hevristiko hitrosti izvajanja (angl. *progress rate*)

$$ProgressRate = ProgressScore/T\ , \tag{4.3}$$

kjer T predstavlja čas izvajanja opravila do trenutka meritve. Ob predpostavki, da se opravilo izvaja konstantno hitro, je preostanek časa do njegovega zaključka enak

$$(1 - ProgressScore)/ProgressRate\ . \tag{4.4}$$

Razporejevalnik spekulativno izvaja le opravila, ki vplivajo na krajši odzivni čas poslov, zato ne izvaja vseh počasnih opravil. Z izbiro opravil z največjim preostankom časa do njihovega zaključka je tako tudi zagotovljeno, da počasna opravila tik pred zaključkom izvajanja niso dodatno spekulativno zagnana, saj skoraj zagotovo ne bodo izvedena hitreje kot prvotno opravilo.

Heterogenost okolja se upošteva pri izbiri vozlišča za izvajanje spekulativnega opravila, kjer imajo prednost odzivnejša vozlišča. Število spekulativnih opravil je možno omejiti.

4.3 Pravični razporejevalnik

V tem poglavju bomo predstavili pravični razporejevalnik, ki se uporablja za razporejanje poslov v večuporabniških *MapReduce* gručah. Najprej bomo prikazali lastnosti razporejevalnika in probleme, katere rešuje ter v nadaljevanju predstavili algoritme, ki omogočajo, da se posli v takem okolju razporejajo pravično ter dosegajo visoko prepustnost.

Praktični del diplomskega dela temelji na omenjenem razporejevalniku in implementaciji časovnega okna. Zanj smo se odločili predvsem zaradi velikega

[1] Odzivni čas posla je določen kot razlika med časom zaključka in časom izstavitve posla.

nabora konfiguracijksih možnosti in razširjenosti uporabe. Opis razporejevalnika je povzet po navedeni literaturi, priročniku [1] in izvorni kodi verzije *Hadoop* 0.20.205, ki je v času pisanja zagotavljala najstabilnejšo različico implementacije razporejevalnika.

4.3.1 Predstavitev razporejevalnika

Razvoj pravičnega razporejevalnika izhaja iz potrebe po deljenju računskih virov med več uporabnikov in zmanjšanju odzivnih časov poslov, ki so prisotni v gruči. Razporejanje temelji na pravičnem deljenju virov, kjer vsak posel prejme v povprečju enak delež računskih virov, ki so v gruči na voljo. Poleg pravičnega razporejanja so v razporejevalnik vključeni tudi algoritmi, ki zagotavljajo krajšo odzivnost poslov glede na lokacijo podatkov in upoštevajo odvisnost izvajanja med *map* in *reduce* opravili.

Glavni lastnosti razporejevalnika sta doseganje zagotovljene ravni storitve z uporabo **minimalnih deležev** (angl. *minimum share*) in izboljšanje **odzivnih časov** manjših interaktivnih *adhoc* poizvedb. Slednji so bili v primeru produkcijskih poslov in uporabe razporejevalnika *FIFO* večkrat predolgi, kar je vplivalo na uporabnost in odzivnost sistema [39].

Deljenje računskih virov je urejeno z **dvostopenjsko hierarhijo** razporejanja. Zunanje razporejanje skrbi za delitev virov med uporabniškimi bazeni, medtem ko notranje služi razporejanju poslov v posameznem bazenu (slika 4.1). V obeh primerih se uporablja princip statističnega multipleksiranja [39], kjer se neuporabljene vire uporabnika dodeli ostalim uporabnikom gruče. Uporabniki imajo možnost izvajanja tako daljših paketnih kot tudi krajših poizvedb nad različnimi zbirkami skupnih podatkov, ki lahko pripadajo skupinam iz različnih oddelkov (angl. *batch and interactive jobs*).

Razporejanje v podatkovno intenzivnih *MapReduce* okoljih se od razporejanja v običajnih gručah razlikuje predvsem v upoštevanju podatkovne lokalnosti opravil in odvisnosti v njihovem izvajanju. Zaradi velikih količin podatkov ima **podatkovna lokalnost** (angl. *data locality*) ključno vlogo pri zagotavljanju krajše odzivnosti poslov in zmanjšanju obremenjenosti omrežja. Običajni gručni razporejevalniki, kot je npr. Torque, dodelijo uporabniku le fiksno število delovnih vozlišč, medtem ko so njihovi podatki porazdeljeni med vsa vozlišča gruče. V prejšnjih različicah razporejevalnika (*Hadoop On Demand*) je takšna omejitev v veliki meri vplivala na slabšo odzivnost poslov. Vzrok zanjo se je nahajal v zmanjšani lokalnosti opravil in potrebi po prenašanju podatkov iz vozlišč, ki se niso nahajala v skupini delovnih vozlišč. Dodatna pomanjkljivost takšne izvedbe se je pokazala v neizkoriščenosti nekaterih vozlišč. Tista

neaktivna bi sicer lahko bila izrabljena s strani drugih uporabnikov, vendar zaradi statične omejitve dodeljenih vozlišč to ni bilo mogoče [39]. Drugi primer prepoznavnega razporejevalnika za sicer procesno intenzivnejše aplikacije je Condor [37]. Slednji sicer omogoča določeno raven lokalnosti, vendar le na nivoju geografskih lokacij in ne na nivoju posameznih vozlišč.

Pravični razporejevalnik rešuje problem podatkovne lokalnosti z algoritmom **zakasnjenega razporejanja**. Po izbiri posla sledi določitev opravila, ki pa ga razporejevalnik opravi glede na prisotnost podatkov na vozlišču, katero je podalo zahtevo za izvajanje novega opravila. Če vozlišče ne vsebuje podatkov za nobenega izmed opravil v izbranem poslu, se izvajanje posla zakasni za določen čas, prosti vir pa postane na razpolago drugemu poslu. Kljub zakasnitvi posla, se slednje bistveno ne pozna pri pravičnem razporejanju. Algoritem na omenjeni način dosega skorajda optimalno lokalnost in do dvakrat večjo prepustnost poslov v primerjavi s *FIFO* razporejevalnikom [40].

Druga pomembna lastnost, ki jo razporejevalnik upošteva je **zaporednost izvajanja** opravil *map* in *reduce*, kjer je začetek opravila *reduce* pogojen z zaključkom vseh opravil *map* v skupnem poslu. V standardnih okoljih za porazdeljeno procesiranje odvisnost izvajanja med opravili ni prisotna. V okoljih *MapReduce* odvisnost izvajanja prinaša slabšo izkoriščenost virov in zastoje poslov, katerih vzrok se lahko nahaja v izvajanju daljših opravil. Izvajanje opravila *reduce* se prične z dodelitvijo opravila delovnemu vozlišču, ki zaradi nedokončanih opravil *map* še ne more uporabiti metode *reduce*. Problem se omili z razdelitvijo opravil *reduce* na stopnji *kopiraj* in *izračunaj*, ki iz zaključenih opravil *map* omogoči takojšnje kopiranje rezultatov. Po prejetju vseh rezultatov opravilo *reduce* preide v stopnjo *izračunaj*, kjer se nad podatki izvede funkcija *reduce*.

4.3.2 Osnovni gradniki

Bazen (angl. *pool*) je struktura za hranjenje in upravljanje poslov. Naloga razporejevalnika je zagotavljanje pravičnosti pri dodeljevanju računskih virov med bazene. Običajno je v sistemu prisotnih več bazenov, kjer si vsak uporabnik ali skupina lasti svojega. Poleg razporejanja med bazeni obstaja tudi možnost notranjega razporejanja poslov v bazenu po principu *FIFO* ali *fairshare*.

Pravični delež (angl. *fairshare*) je predvideno število računskih virov, ki so bazenu dodeljeni na podlagi njegove teže, potreb (angl. *demand*), minimalnega deleža in števila vseh prisotnih računskih virov v gruči. Izračun se v

ogrodju izvaja s pomočjo binarnega iskanja in je opisan v poglavju 4.3.5.

Minimalni delež (angl. *minshare*) predstavlja zagotovljeno število računskih virov, ki so v vsakem trenutku bazenu na voljo ne glede na izračun pravičnega deleža. Pri tem velja, da se v primeru neaktivnosti bazena zagotovljeni minimalni delež ne ohranja, temveč se porazdeli med ostale aktivne bazene. Z minimalnimi deleži je tako v gruči zagotovljena določena raven storitve produkcijskih poslov, hkrati pa so lahko prisotni tudi neprodukcijski posli, katerih izvajanje nima vpliva na delovanje prvih. Za minimalne in pravične deleže veljajo naslednje lastnosti:

- Pravični delež bazena je vsaj toliko velik kot je njegov minimalni delež. V primeru, ko seštevek minimalnih deležev bazenov presega število skupnih računskih virov v sistemu, se izračunani pravični deleži sorazmerno zmanjšajo pri vseh bazenih.[2]

- Bazeni, ki imajo v izvajanju manj opravil, kot je njihov minimalni delež, imajo prednost pri dodeljevanju virov pred bazeni, ki so minimalni delež že dosegli.

- Za doseganje dogovorjene ravni storitve se poleg minimalnih deležev uporablja možnost prekinjanja poslov z določitvijo prekinitvene časovne omejitve (angl. *preemption timeout*). Če v prekinitvenem intervalu bazenu ni bilo dodeljeno dovolj virov za dosego minimalnega deleža, se prosti viri zanj zagotovijo s prekinitvijo opravila, ki je bil nazadnje izbran v izvajanje.

Potreba posla (angl. *job demand*) predstavlja seštevek vseh nezaključenih opravil. To so opravila v izvajanju, spekulativna opravila in tista, ki čakajo na izvajanje.

Prioriteta in teža posla sta pomembna parametra pri notranjem razporejanju v posameznem bazenu. Posli se lahko v njem razporejajo v *FIFO* ali po pravičnem načinu. Pri prvem velja, da se posli razporejajo glede na njihove prioritete in čase prispetja, medtem ko so pri pravičnem načinu prioritete poslov vključene v izračune njihovih tež.

[2]Trenutne izdaje Hadoopa funkcionalnosti ne podpirajo, kar v omenjenem primeru vpliva na napačen izračun pravičnih deležev. Obstoj funkcionalnosti je bil preverjen v izdaji *Hadoop* 0.20.205 in 1.0.2.

Teža posla pri pravičnem razporejanju uravnava delež virov, ki so poslu na voljo. Posel s težo 2,0 tako prejme dvakrat več virov kot navaden posel. Razporejevalnik omogoča izračun teže na sledeče načine:

- izračun teže glede na prioriteto

$$weight = weight \times priority \ , \tag{4.5}$$

- izračun teže glede na velikost posla

$$weight = \frac{\ln\left(runnableTasks + 1\right)}{\ln 2} \ , \tag{4.6}$$

- izračun teže z implementacijo lastnega urejevalnika tež. Takšen primer je razporejanje krajših poslov pred daljšimi, kjer se za krajši čas dodeli večjo težo krajšim poslov, ne da bi pri tem škodovali izvajanju daljših.

Teža bazena vpliva na neenakomerno delitev virov med bazeni. Bazen s težo 2,0 prejme dvakrat več virov kot običajen bazen s težo 1,0.

Omejitev izstavljanja poslov. S to možnostjo je mogoče omejiti število izstavljenih poslov tako za uporabnika kot tudi za bazen. Omejitev vpliva na količino vmesnih podatkov, ki se ustvarijo med izvajanjem poslov. Posli, ki gredo v izvajanje, so izbrani na podlagi prioritet in časa prispetja. Ob prekoračitvi omejitve se izvajanje posla zakasni do zaključka enega izmed prejšnjih poslov.

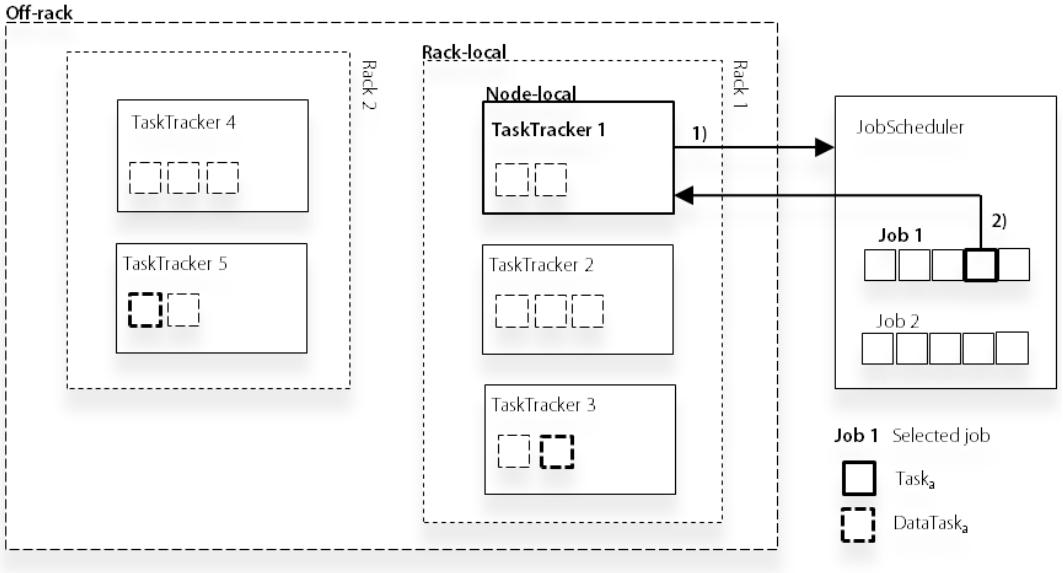
Lokalnost posla določa množico opravil v izbranem poslu, ki se lahko izvajajo na trenutnem vozlišču. Namen omejitve lokalnosti je izbira opravila, ki ima podatke najbližje trenutnemu vozlišču. Za posel so lahko predpisane naslednje lokalnosti:

- *Node-local*: Določa množico opravil v poslu, katerih podatki za obdelavo se nahajajo le na trenutnem vozlišču, ki je podalo zahtevo po izvajanju novega opravila.

- *Rack-local*: Določa množico opravil v poslu, katerih podatki se nahajajo v skupini bližje povezanih vozlišč trenutnega vozlišča. Za izvajanje opravila je potreben prenos podatkov iz bližnjega vozlišča.

- *Off-rack*: Določa množico opravil v poslu, katerih podatki se nahajajo v širši okolici trenutnega vozlišča.

Razporejevalnik stremi k izvajanju vozlišču lokalnih opravil, saj s tem pripomore k manjšemu pretoku podatkov in posledično manjši obremenitvi omrežja. Ker čakanje na lokalno opravilo posla lahko privede do njegovega zastoja, razporejevalnik njegovo lokalnost spreminja na podlagi pretečenega časa, ki je bil porabljen za njegovo iskanje. V poglavju 4.3.6 lahko zasledimo kako se vršijo spremembe lokalnosti posla, medtem ko je na sliki 4.3 prikazan primer izbire opravila za raven lokalnosti *Rack-local*. V primeru navajamo izbiro opravila za izbrani posel *Job 1* ob pozivu za izvajanje novega opravila delovnega vozlišča $TaskTracker$ 1.



Slika 4.3: Dodelitev opravila vozlišču $TaskTracker$ 1 z upoštevanjem ravni lokalnosti *Rack-local*.

Razporejevalnik najprej poizkuša vozlišču $TaskTracker$ 1 iz posla *Job1* določiti lokalno opravilo. Ker nobeno izmed opravil posla ne vsebuje podatkov na trenutnem vozlišču, razporejevalnik poskuša iz istega posla določiti opravilo katerega podatki se nahajajo v množici bližje povezanih vozlišč (*Rack-local*). Kot prikazuje slika je v našem primeru takšno opravilo $Task_a$[3]. Podatki za izvajanje omenjenega opravila se nahajajo na vozliščih $Tasktracker$ 3

[3]Opravilo in lokacije njegovih podatkov so v sliki odebeljene.

in $TaskTracker$ 5. Zaradi omejitve ravni lokalnosti *Rack-local* se prenos podatkov izvrši iz vozlišča $Tasktracker$ 3. Po prenosu podatkov se lahko prične izvajanje omenjenega opravila.

Če v zgornjem primeru vozliča $Tasktracker$ 1, $Tasktracker$ 2 in $Tasktracker$ 3 ne bi vsebovala podatkov za izvajanje vsaj enega od opravil posla Job_1, bi se njegovo izvajanje zakasnilo do poziva razporejanja naslednjega vozlišča.

4.3.3 Algoritem pravičnega razporejanja in izbire opravil

V sledečem razdelku je prikazano razporejanje opravil in njihovo dodeljevanje delovnim vozliščem. Opisi in sklici vrstic se nanašajo na Algoritem 2, ki je opisan v nadaljevanju in zaradi preglednosti ne vsebuje mehanizmov zakasnjenega razporejanja.

Dodeljevanje opravil se prične ob prejemu statusnega sporočila, ki ga upravljavec poslov prejme od delovnega vozlišča (vrstica 1). Če ima delovno vozlišče na razpolago proste računske vire in če se z dodatnim opravilom ne preobremeni vozlišča, sledi izbira bazena, iz katerega je določen posel za dodelitev novega opravila. Izbira bazena običajno poteka najprej glede na delež primanjkljaja, ki označuje koliko je posamezen bazen pod svojim minimalnim deležem, nato pa glede na število opravil v izvajanju:

- Bazeni, katerih število opravil v izvajanju je manjše od njihovega minimalnega deleža, imajo prednost pred bazeni, ki so svoj minimalni delež že dosegli. Izbran je bazen, katerega primanjkljaj do minimalnega deleža je največji. Povedano drugače je to tisti bazen, katerega delež poslov v izvajanju v odnosu do minimalnega deleža je najmanjši

$$\min_{p \in pools} \left(\frac{runningTasks_p}{minShare_p} \right) . \tag{4.7}$$

- V primeru, ko so vsi bazeni izpolnili svoj minimalni delež, je bazen izbran glede na delež opravil v izvajanju in težo bazena

$$\min_{p \in pools} \left(\frac{runningTasks_p}{weight_p} \right) . \tag{4.8}$$

V ogrodju je zgornji postopek razporejanja implementiran v sortirnem primerjalniku (vrstica 3), ki se ga lahko nadomesti s svojo lastno implementacijo.

Izbiri bazena sledi izbira posla (vrstica 5), ki je določen glede na način notranjega razporejanja (*FIFO* ali *pravično razporejanje*). Po določitvi posla sledi izbira opravila (vrstica 7). Pred njegovo izbiro se z algoritmom zakasnjenega razporejanja najprej določi reven lokalnosti, nato pa vrsto opravila. Prednostno so najprej izbrana opravila, ki so bila v prejšnjih razporejanjih neuspešna, nato opravila, ki še niso v izvajanju ter nazadnje spekulativna opravila.

V primeru, da za izbrani posel razporejevalnik ne najde ustreznega opravila, se iskanje nadaljuje ali pri naslednjem poslu ali pa v naslednjem bazenu, če za nobenega izmed poslov v trenutnem bazenu ne najde ustreznega opravila. Ko je opravilo najdeno, se ga doda na seznam opravil za izvajanje (vrstica 10), kateri se na koncu razporejevalnega cikla posreduje delovnemu vozlišču, ki je sprožil postopek razporejanja (vrstica 15). V opisu algoritma lahko razberemo, da izbranemu bazenu dodelimo naenkrat le eno opravilo. Če so po dodelitvi opravila izpolnjeni vsi pogoji izvajanja in z novim opravilom ne presegamo kapacitete vozlišča, se razporejanje nadaljuje s ponovnim sortiranjem bazenov in izbiro novega opravila (vrstica 2).

Algoritem 2 AssignTasks

Input: TaskTracker n
Output: *tasks* for execution on n

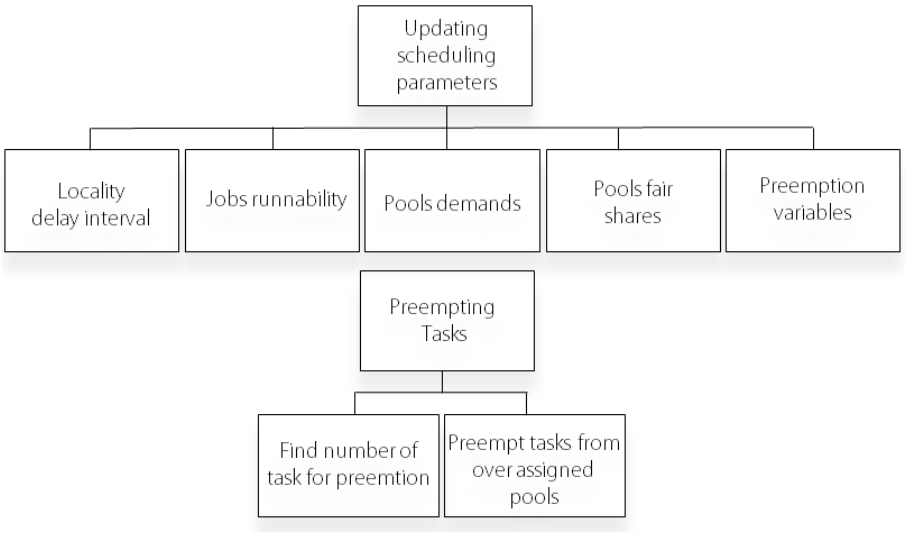
```
 1: heartbeat is received from node n
 2: while another task can be assigned on n do
 3:     sort pools in increasing order based on fairshare policy
 4:     for p in pools do
 5:         sort jobs in p depending on internal pool scheduling
 6:         for j in jobs do
 7:             assign task from j to launch on n taking account task locality
 8:         end for
 9:         if task is found then
10:             add task to tasks
11:             break          → Assign only one task per loop
12:         end if
13:     end for
14: end while
15: return tasks
```

4.3.4 Zagotavljanje pravičnosti in prekinitve izvajanja

V prejšnjem razdelku smo spoznali, da je izbira posla in kasneje opravila odvisna od bazena, v katerem se posel nahaja. Videli smo, da se pri razporejanju bazenov najprej upoštevajo primanjkljaji do minimalnih deležev in nato deleži opravil v izvajanju v odnosu s težami bazenov.

Za zagotavljanje pravičnega razporejanja je v razporejevalniku dodatno prisoten *mehanizem prekinitev* opravil, ki bazenom omogoča pravično izrabo virov. Razporejevalnik število prekinitvenih opravil določi glede na primanjkljaje bazenov, ki v določenem intervalu niso dosegli svojega minimalnega deleža in vsaj polovice pravičnega deleža. Če v določenem intervalu meja ni bila dosežena, se razlika do minimalnega in celotnega pravičnega deleža šteje kot primanjkljaj bazena. Pri njegovem izračunu se upošteva dejansko potrebo bazena in največjo vrednost med obema razlikama. Število prekinitvenih opravil je tako enako $min\left(d_p, max\left(dif_{minshare}, dif_{fairshare}\right)\right)$. Primanjkljaj poskuša razporejevalnik nadomestiti s prekinitvijo opravil v bazenih, ki presegajo svoje pravične deleže.

Prekinitve izvajanja se pričnejo pri opravilih, ki so v izvajanje prišli na koncu, saj se s tem ohrani delo predhodnih opravil in pa zagotovi, da ne pride do ponavljajočega prekinjanja istega opravila.



Slika 4.4: Dekompozicija posodabljanja parametrov razporejanja in prekinitve opravil.

Posodabljanje pravičnih deležev in ostalih parametrov razporejanja je prikazano v sliki 4.4. Osveževanje parametrov poteka v pol sekundnih intervalih, medtem ko so prekinitve izvajanja opravil aktivne vsakih petnajst sekund. Izračun pravičnih deležev, ki upošteva potrebe bazenov, minimalne deleže in njihove teže, je predstavljen v naslednjem poglavju.

4.3.5 Izračun pravičnih deležev

Opisali smo, da algoritem razporejanja za dosego pravičnega dodeljevanja med bazene ne uporablja zgodovine dodeljevanj in vnaprej izračunanih deležev (Poglavje 4.3.3). Dodeljevanje tako poteka glede na trenutna opravila v izvajanju posameznega bazena.

Pri prekinitvah želimo vedeti, kakšen je pravični delež bazenov v primerjavi z opravili v izvajanju in kdaj je bil nazadnje dosežen. V izračunu pravičnih deležev se upoštevajo tako potrebe in minimalni deleži bazenov kot tudi njihove teže. Izračun predstavlja, koliko računskih virov oziroma kakšen pravični delež bi prejel vsak bazen, če bi bili na voljo vsi viri gruče.

Za uteženo pravično razporejanje velja, da je delež dodeljenih virov in teže posameznega bazena enak pri vseh bazenih in njihov seštevek je manjši ali enak številu računskih virov v gruči. V primeru, ko ne upoštevamo potreb bazenov in njihovih minimalnih deležev, bi uteženo razporejanje posameznega bazena i predstavljalo sledeče število opravil

$$slotsAssigned_i = totalSlots \cdot \frac{w_i}{\sum_{p \in pools} w_p} .$$

Ker se pri izračunu pravičnih deležev poleg tež bazenov upoštevajo tudi njihovi minimalni deleži m_i in potrebe d_i, si pri izračunu deležev pomagamo s faktorjem r, ki preslika teže bazenov v računske vire (angl. *weight-to-slot ratio*). Pri izračunu pravične delitve je prav faktor r tisti, ki ga iščemo in za katerega veljajo naslednje lastnosti:

- bazenu, katerega potreba d_i je manjša od predvidenih dodeljenih virov rw_i, je dodeljeno d_i računskih virov,
- bazenu, katerega minimalni delež m_i je več kot rw_i, je dodeljeno $\min(m_i, d_i)$ virov,
- vsem ostalim bazenom je dodeljeno rw_i virov in
- vsota dodeljenih virov je enaka ali manjša številu vseh virov v gruči t.

Zgornje lastnosti oziroma število vseh dodeljenih virov za konkreten r povzema funkcija *f(r)*, ki je navzgor omejena s številom vseh virov v gruči t

$$t \geq f(r) = \sum_{i \in pools} \min(d_i, \max(rw_i, m_i)) . \tag{4.9}$$

Za izračun parametra r razporejevalnik uporablja binarno iskanje saj je funkcija naraščajoča za oba člena vsote. Za $r = 0$ so bazenom dodeljeni le minimalni deleži. Zgornja meja vseh dodeljenih virov za r je dosežena, ko njihovo število presega t ali pa ko so vse potrebe poslov zadoščene. Ob koncu iskanja primernega r se pravični deleži bazenov posodobijo glede na zgoraj opisane točke.

Število iteracij iskanja je odvisno od števila korakov k, ki določa natančnost dodeljevanja. Časovna zahtevnost algoritma je $\mathcal{O}(nk)$, kjer je n število bazenov prisotnih bazenov, za katere se ob vsakem iteraciji r-ja računa predvideno število dodeljenih računskih virov.

4.3.6 Algoritem zakasnjenega razporejanja

Že v predstavitvi pravičnega razporejevalnika smo omenili, kako pomembna je izbira vozlišču lokalnih opravil za doseganje boljše odzivnosti poslov. Bistvo zakasnjenega razporejanja je sprostitev strogo pravičnega dodeljevanja opravil z namenom povečanja lokalnosti poslov in posledično izboljšanja njihovih odzivnih časov. Pri pravičnem razporejanju, ki ne vključuje zakasnjenega razporejanja lahko zasledimo naslednji težavi:

Problem dodelitve opravila iz posla na začetku vrste (angl. head-of-line problem): Pri izbranem poslu z manjšim številom opravil je majhna verjetnost, da bo vozlišče, ki je zahtevalo novo opravilo, vsebovalo njegove podatke. Če je opravilo vedno izbrano iz posla na začetku vrste in ne glede na to, katero vozlišče sprašuje po njem, je posledično lokalnost takega posla slabša.

Pojav lepljivih računskih virov (angl. sticky slots): Pri pravičnem razporejanju se pogosto dogaja, da je poslu dodeljen isti računski vir, v katerem se je pravkar končalo njegovo opravilo. Ko se opravilo zaključi, se zmanjša tudi pravični delež posla, kar pomeni, da je posel ponovno izbran za iskanje novega opravila na istem vozlišču.

Zakasnjeno razporejanje rešuje oba problema. V primeru, da za izbrani posel ni na voljo opravil, ki bi se lahko izvajale nad podatki trenutnega vozlišča, je

izbira opravila prepuščena naslednjemu poslu. Izvajanje izbranega posla pa je zakasnjeno do naslednjega razporejanja. Zastoje poslov preprečujejo časovne omejitve, s katerimi se uravnava lokalnost poslov in posledično izvajanje tudi nelokalnih opravil.

Delež lokalnih opravil posla λ je možno spreminjati na račun zakasnitev njegovega izvajanja. Ocena lokalnosti posla z N opravili v gruči z M vozlišči, kjer ima vsako vozlišče L računskih virov in kjer je replikacijski faktor podatkov posameznega opravila enak R, se izračuna po naslednji enačbi [40]

$$\lambda = \frac{1}{N} \sum_{K=1}^{N} 1 - \epsilon^{-RDK/M} \ . \tag{4.10}$$

Parameter D v enačbi predstavlja največjo število zakasnitev za iskanje lokalnega opravila, nato se iz posla dopusti izbira tudi nelokalnega opravila. Na eni strani parameter D vpliva na delež lokalnosti, po drugi strani pa na čas in zakasnitev izvajanja posla.

Za izračun zakasnitve pri iskanju lokalnega opravila potrebujemo povprečni izvajalni čas opravil in število virov, ki so v sistemu na voljo. Če je povprečni izvajalni čas opravila enak T in S predstavlja število vseh računskih virov v sistemu, potem velja, da je hitrost dodeljevanja novih virov enaka T/S. Če upoštevamo, da dopuščamo iskanje lokalnega opravila do D zakasnitev oziroma dodeljevanj, sledi da je največji čas za iskanje lokalnega opravila enak

$$W = \frac{DT}{S} = \frac{DT}{LM} \ . \tag{4.11}$$

V algoritmu zakasnjenega razporejanja zakasnitveni parameter W predstavlja dopusten čas zakasnitve za iskanje lokalnih opravil poslov. Določimo ga glede na parametre okolja in želeni delež lokalnosti. Delež lokalnih opravil poslov ob poznavanju ostalih parametrov lahko ocenimo z enačbo 4.10. Algoritem uporablja omenjeni parameter za spreminjanje ravni lokalnosti posameznega posla, s čimer je določeno kdaj se lahko iz posla izvajajo tudi nelokalna opravila.

V pravičnem razporejevalniku ima vsak posel lahko 3 različne ravni lokalnosti[4], zato pri njihovem razporejanju algoritem uporablja dva zakasnitvena parametra W_1 in W_2. Na začetku ima vsak posel raven lokalnosti enak *node*, kar pomeni, da se lahko iz njegovih opravil izvajajo le tista, katera imajo prisotne podatke na trenutnem vozlišču. V primeru, da opravilo po preteku časa W_1 še ni bilo najdeno, posel preide na raven *rack*, kjer je dovoljeno izvajanje

[4]Lokalnost posla je opisana v 4.3.2.

vseh njegovih opravil, katerih podatki se nahajajo v množici bližje povezanih vozlišč. Če po preteku časa W_2 v poslu še vedno ni bilo mogoče najti primernega opravila, se njegovo iskanje razširi na vsa opravila, katerih podatki se nahajajo v širši okolici trenutnega vozlišča (angl. *off-rack*). V primeru, da je v poslu najdeno opravilo z nižjo ravnijo lokalnosti, se mu le-ta ustrezno zniža, kot je razvidno tudi v algoritmu 3. Stopnja lokalnosti posla je v algoritmu označena z $j.level$, medtem ko je zakasnitveni čas za iskanje vozlišču lokalnega opravila označen z oznako $j.wait$.

Kljub čakanju na lokalno opravilo avtorji v [40] navajajo, da je hitrost izvajanja lokalnih opravil v primerjavi z nelokalnimi in ob prisotnosti večjega števila računskih virov tudi do dvakrat večja. Kot smo omenili je čas za iskanje lokalnega opravila običajno določen glede na hitrost izvajanja opravil T/S, parametrov gruče in želene lokalnosti posla. Podrobnosti in obnašanje algoritma pri različnih parametrih lahko bralec najde v članku [40].

Algoritem 3 Delay scheduling

Input: TaskTracker n
Output: *tasks* for execution on n

```
1: heartbeat is received from node n
2: while another task can be assigned on n do
3:     sort pools in increasing order based on fairshare policy
4:     for p in pools do
5:         sort jobs in p depending on internal pool scheduling
6:         for j in jobs do
7:             if j has a node-local task t on n  then
8:                 set j.wait = 0 and j.level = node
9:                 add t to tasks
10:                break
11:            else if j has a rack-local task t on n and
12:            (j.level ≥ rack or j.wait ≥ W1) then
13:                set j.wait = 0 and j.level = rack
14:                add t to tasks
15:                break
16:            else if j.level = off-rack  or
17:            (j.level = node and j.wait ≥ W1 + W2) then
18:                set j.wait = 0 and j.level = off-rack
19:                add any unlanched task t to tasks
20:                break
21:            else
22:                set j.skipped = true
23:            end if
24:        end for
25:        if task is found then
26:            break          → Assign only one task per loop
27:        end if
28:    end for
29: end while
30: return tasks
```

Poglavje 5

Implementacija uteženega razporejanja s pravičnim oknom

V sledečem poglavju sta opisana model razporejanja in primer nadgradnje pravičnega razporejevalnika s komponento pravičnega okna. Temu podoben koncept dodeljevanja opravil lahko zasledimo v sistemih v realnem času pri strežnikih z odlogom (angl. *deferrable servers*). Cilj nadgradnje sta ohranitev lastnosti pravičnega razporejevalnika in vpeljava robustnega ter striktnejšega pravičnega dodeljevanja virov, s čimer želimo doseči enakomernejše in predvidljivejše izvajanje poslov. Poglavje se po opisu modela nadaljuje z evaluacijo razporejevalnika in primerjavo njegovih rezultatov z ostalimi razporejevalniki v tem okolju. Zaključi se z opisom ugotovitev in možnosti nadaljnjih izboljšav.

5.1 Uvod

Razporejanje v realnem času obsega sisteme, ki se lahko izvajajo strogo ali ohlapno v realnem času (angl. *hard or soft real-time systems*) [28]. Izvajanje opravil je v takšnih sistemih navadno omejeno z izvajalni roki, medtem ko je vrsta sistema v realnem času določena glede na striktnost upoštevanja njihovih rokov. Za ohlapne sisteme v realnem času običajno velja, da doseganje rokov ni edina prioriteta v sistemu, zato se njihova prekoračitev dopušča, in sicer z namenom izpolnjevanja ostalih parametrov razporejanja [28]. Podoben primer je algoritem zakasnjenega razporejanja s konca prejšnjega poglavja. Opravila v okolju *Hadoop* sicer ne vsebujejo rokov, do katerih morajo biti izvedena, kljub temu pa lahko v tovrstnem okolju opazimo ohlapnejše razporejanje posameznih opravil, kjer se z njihovo zakasnitvijo poskuša izboljšati odzivnost celotnega sistema.

V sistemih so običajno prisotne tri vrste opravil: periodična, sporadična in aperiodična. Za prvo skupino opravil velja, da se v sistemu pojavljajo periodično oziroma v določenih intervalih, njihov razpored pa se lahko opravi pred pričetkom izvajanja. Drugače velja za ostali dve skupini, pri katerih je prihod opravil pogojen z zunanjimi dogodki ter je tako naključen in vnaprej neznan. V letalstvu lahko primer zunanjega dogodka predstavlja preklop z avtomatskega na ročni način letenja, kjer so sporadična in aperiodična opravila za upravljanje preklopa vnaprej določena. Skupna lastnost tako sporadičnih kot aperiodičnih opravil je njihov naključni prihod, medtem ko se razlikujejo v rokih, do katerih morajo biti opravila izvedena. Prekoračitve rokov sporadičnih opravil lahko predstavljajo kritično točko v delovanju in stabilnosti sistema, medtem ko je prekoračitev rokov pri aperiodičnih opravilih ohlapnejša in nima ključnega vpliva na delovanje sistema.

Trenutna implementacija pravičnega razporejevalnika v okolju *Apache Hadoop* omogoča razporejanje zgolj aperiodičnih opravil. Zanje je značilno, da ne vsebujejo rokov in v sistem prihajajo naključno oziroma je njihov prihod odvisen od uporabnikov in njihove aktivnosti izstavljanja poslov. Iz omenjenih lastnosti sledi, da se opravila v ogrodju *Hadoop* izvajajo ohlapno v realnem času. Razporejevalnik, ki striktneje obravnava časovne omejitve opravil v oblačnih okoljih lahko najdemo v [33].

Razporejanje v ogrodju *Hadoop* umeščamo med dogodkovno in prioritetno gnana razporejanja, ki se izvajajo nad eno ali več prioritetnimi vrstami. Za prioritetno gnana razporejanja je značilno, da se prioritete izračunavajo ob prihodu dogodkov in da, v primeru da so v vrstah prisotna opravila, računskih virov nikoli namerno ne puščajo v neaktivnem stanju. Dogodkovno proženje izračunov prioritet je v okolju *Hadoop* zasnovano na pozivih delovnih vozlišč, ki sprašujejo po novem opravilu.

5.2 Model pravičnega okna

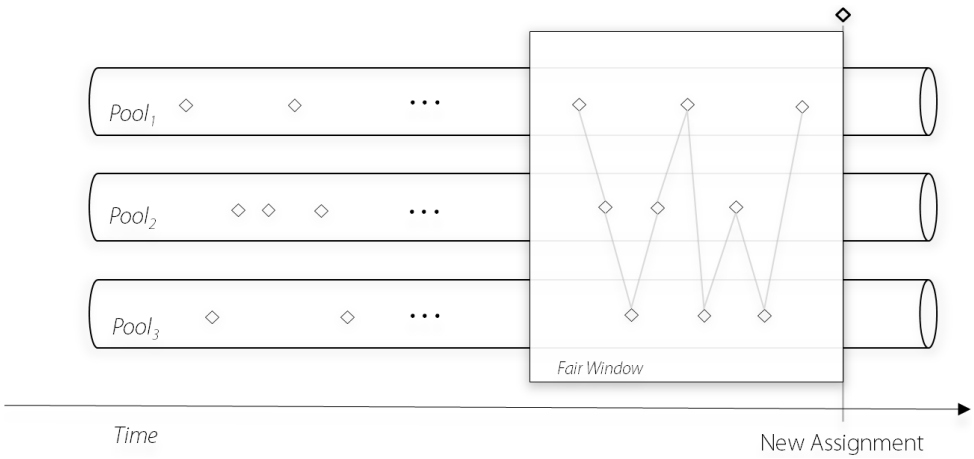
Podobno zamisel kot je razporejanje s pravičnim oknom lahko zasledimo v sistemih v realnem času pri strežnikih z odlogom. Za aperiodična opravila strežnikov je značilno, da se izvajajo na podlagi količine časa, ki jim je bila določena ob začetku izvajanja. Čas, ki je opravilom na voljo za izvajanje, imenujemo tudi proračun (angl. *budget*). Le-ta se dinamično spreminja glede na pravila polnjenja in porabe. Polnjenja proračuna potekajo v intervalih $k \cdot p$, ki predstavljajo večkratnik periode strežnika. Njegova poraba se vrši z izvajanjem aperiodičnih opravil. Pri pravilih polnjenja strežnik proračuna ne

akumulira, neporabljen proračun pa ohranja za nadaljnja opravila.

Pri implementaciji razporejevalnika s pravičnim oknom (angl. *FWS*) smo zamisel o ohranjanju proračuna želeli prenesti v koncept ohranjanja pravičnega izvajanja poslov. Kot osnovno strukturo smo izbrali pravično okno, ki za vsak bazen hrani število dodeljenih računskih virov. Slednji se upoštevajo pri določitvi bazena, ki mu je dodeljen naslednji prosti vir. Za vsako dodelitev vira se iz izbranega bazena izvede eno opravilo, zato si lahko število dodelitev predstavljamo kot število izvedenih opravil posameznega bazena.

Izbira bazena poteka za vsak prosti vir delovnega vozlišča in je odvisna od tež bazenov in števila računskih virov, ki so bili bazenu dodeljeni v trenutnem oknu. Rezultat zunanjega razporejanja prostih računskih virov je torej izbira bazena, ki mu je odobreno izvajanje enega izmed svojih opravil. Konkretna izbira opravila je prepuščena notranjemu razporejanju poslov v bazenu.

Koncept razporejanja z oknom je bil izbran predvsem zaradi hranjenja lokalne zgodovine dodeljenih virov, s čimer smo se izognili njihovemu spremljanju od začetka izvajanja sistema. Na sliki 5.1 vidimo primer okna in zaporedje bazenov, ki jim je bilo dodeljeno zadnjih devet računskih virov.



Slika 5.1: Dodeljevanje računskih virov z uporabo pravičnega okna.

Izvajanje smo ob predpostavki enakih tež bazenov označili kot **pravično**, če so v trenutnem oknu vsi bazeni prejeli enako število računskih virov. Pravično razporejanje bi v primeru k enakih bazenov in $k \cdot n$ prostih virov pomenilo, da je bilo vsakemu izmed k bazenov dodeljeno n virov. Kot smo omenili imajo lahko bazeni različne teže, kar pomeni, da je bazenu s težo dva dodeljeno dvakrat več virov kot bazenu s težo ena. Če torej upoštevamo teže bazenov je pravično

izvajanje doseženo takrat, ko so vsi deleži dodeljenih virov in tež bazenov enaki

$$\forall i \in pools : \ \frac{sa_i}{w_i} = k \ , \tag{5.1}$$

kjer sa_i v trenutnem oknu predstavlja število dodeljenih virov i-temu bazenu, w_i njegovo težo in k delež, ki je enak za vse bazene.

Primankljaj bazena (angl. *pool deficit*). Pravično okno ob vsakem razporejanju upošteva zgodovino dodelitev in tako kot strežniki z odlogom med neaktivnostjo bazena ohranja njegovo prioriteto. Vlogo proračuna v pravičnem oknu opravljajo primanjkljaji bazenov. Ob vsakem razporejanju primanjkljaji določajo prioriteto bazenov in odločajo kateremu izmed njih bo dodeljen naslednji prosti računski vir delovnega vozlišča. Primanjkljaj i-tega bazena je v podanem oknu ob času t izračunan kot razlika med pravičnim deležem bazena in številom njegovih trenutno dodeljenih računskih virov

$$dc_i\,(t) = sf_i - sa_i\,(t) \ . \tag{5.2}$$

V zgornji enačbi dc_i predstavlja primanjkljaj i-tega bazena, spremenljivka sf_i njegov pravični delež, medtem ko sa_i število dodeljenih virov v trenutnem oknu.

Pravični deleži (angl. *fairshares*) bazenov so v sistemu statični in opisujejo kolikšno število računskih virov mora prejeti posamezen bazen glede na velikost okna in teže bazenov, da bo dodeljevanje virov pravično. Zaradi potrebe po neenakomernem izvajanju med bazeni se pri izračunu pravičnih deležev bazena upoštevajo tudi njihove teže, ki posledično vplivajo na njihov primanjkljaj in s tem prioriteto razporejanja. Pravični delež posameznega bazena se izračuna po enačbi

$$sf_i = W_l \cdot \frac{w_i}{\sum_{j \in pools} w_j} \ , \tag{5.3}$$

kjer sf_i predstavlja pravično število virov i-tega bazena, w_i in w_j teži bazena i in j, W_l pa velikost pravičnega okna, ki se v našem primeru skozi izvajanje ne spreminja.

5.2.1 Algoritem razporejanja s pravičnim oknom

Razporejevalnik *FWS* temelji na pravičnem razporejevalniku iz prejšnjega poglavja, zato smo večino njegovih lastnosti vzeli v poštev tudi v naši implementaciji. Poleg pravičnega dodeljevanja smo v razporejevalniku upoštevali tudi

minimalne deleže bazenov. Dodeljevanje opravil s pravičnim oknom je opisano v algoritmu 4.

Algoritem 4 AssignTasks

Input: TaskTracker n
Output: $tasks$ for execution on n

```
1: heartbeat is received from node n
2: while another task can be assigned on n do
3:     sort pools considering their minshares and deficits
4:     for p in pools do
5:         sort jobs in p depending on internal pool scheduling
6:         for j in jobs do
7:             assign task from j to launch on n taking account task locality
8:         end for
9:         if task is found then
10:            add task to tasks
11:            update fairwindow and assigned slots for pool p
12:            break          → Assign only one task per loop
13:        end if
14:    end for
15: end while
16: return tasks
```

Razporejanje se prične ob pozivu delovnega vozlišča, ki oznani prisotnost prostih računskih virov in poda zahtevo za izvajanje novega opravila (vrstica 1). Temu sledi izbira bazena, ki je določen glede na minimalne deleže in primanjkljaje bazenov. Izbran je bazen, ki dosega najmanjši delež zagotovljenih virov

$$\min_{i \in pools} \left(\frac{runningTasks_i}{minShare_i} \right) , \tag{5.4}$$

pri čemer $runningTasks_i$ predstavlja število trenutno izvajajočih opravil bazena, $minShare_i$ pa njegov minimalni delež. Če vsi bazeni izpolnjujejo svoje minimalne deleže, se opravilo dodeli s pomočjo okna in primanjkljajev bazenov dc_i. V tem primeru je izbran bazen z največjim primanjkljajem

$$\max_{i \in pools} (dc_i) . \tag{5.5}$$

Zgornji enačbi sta v algoritmu prisotni v sortirnem primerjalniku (vrstica 3). Po določitvi bazena sledi izbira posla, ki se opravi glede na notranje razporejanje poslov v bazenu (vrstica 5). Za izbrani posel se nato poskuša najti opravilo,

ki ustreza njegovi lokalnosti (vrstica 7). Če opravilo ni bilo najdeno se njegovo iskanje nadaljuje v naslednjem poslu. V primeru, da je bilo za izbrani posel najdeno opravilo, se slednje doda na seznam opravil za obdelavo (vrstica 10). Po uspešni izbiri opravila sledi posodobitev parametrov pravičnega okna in števila dodeljenih računskih virov bazena (vrstica 11). Ker pri razporejanju želimo, da se bazenu naenkrat dodeli le eno opravilo, po uspešno najdenem opravilu obhod zanke prekinemo (vrstica 12). V primeru, da so na vozlišču še vedno prisotni dodatni prosti računski viri, se postopek dodeljevanja novega opravila nadaljuje s posodobljenimi parametri razporejanja. Ko vozlišče nima več prostih virov, se algoritem zaključi in izbrana opravila posreduje vozlišču, ki jih nato izvede (vrstica 16).

5.2.2 Opis delovanja pravičnega okna

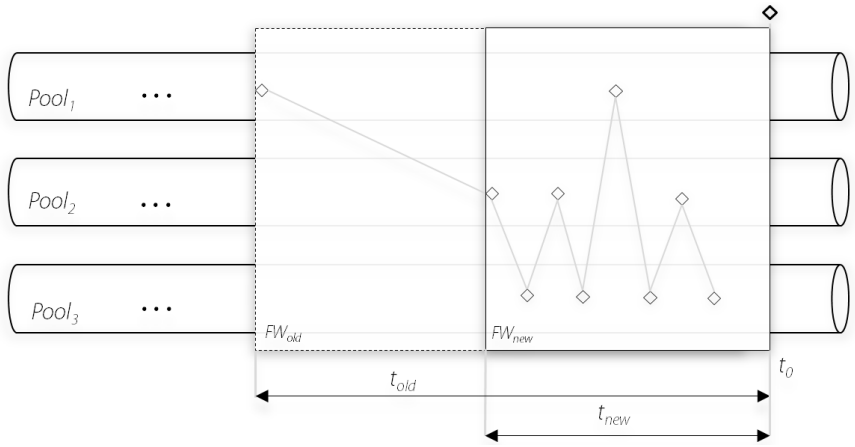
Kot smo že omenili, okno služi pravičnemu dodeljevanju prostih virov med bazene in poteka glede na zgodovino predhodnih dodeljevanj. Ker se velikost okna med izvajanjem ne spreminja, je potrebno ob prihodu novega računskega vira enega izmed starejših odstraniti. Postopek se zaključi z dodajanjem novega vira bazenu, ki je imel pred odstranitvijo največji primanjkljaj.

Primer izbire bazena in posodobitve okna iz algoritma 4 lahko zasledimo na sliki 5.2. V navedenem primeru je velikost okna enaka devet, kar pomeni, da pri razporejanju spremljamo zadnjih devet dodelitev. Bazeni imajo enake teže, medtem ko so njihovi pravični deleži enaki tri.

Ob prihodu novega računskega vira ob času t_0, je izbira bazena v oknu FW_{old} določena na podlagi primanjkljajev bazenov. Najmanjši primanjkljaj ima tretji bazen, ki svoj pravični delež presega za en računski vir, sledi mu drugi bazen brez primanjkljaja in prvi bazen pri katerem je primanjkljaj enak ena. Prosti vir se dodeli prvemu bazenu, ker pa okno nima dovolj prostora za hranjenje nove dodelitve, se pred tem iz okna odstrani najstarejša dodelitev. V našem primeru je to vir izpred devetih razporejanj, ki je bil dodeljen prvemu bazenu. Na sliki 5.2 okno FW_{new} predstavlja stanje dodelitve virov po izbrisu in pred vstavljanjem novega vira k prvemu bazenu. Po zaključku dodeljevanja je v oknu FW_{new} skupno prisotnih devet dodelitev od tega dve v prvem bazenu, tri v drugem in štiri v tretjem bazenu.

Na sliki 5.2 lahko opazimo, da se časovni interval v katerem so vključene dodelitve okna, lahko spreminja v odvisnosti od prihoda računskih virov delovnih vozlišč. V primeru okna FW_{old} lahko zasledimo, da se interval vsebovanih dodelitev t_{old} ob odstranitvi vira skrajša skorajda za polovico.

Implementacija okna z manjšimi spremembami sicer podpira tudi pravično



Slika 5.2: Dodelitev računskega vira bazenu $Pool_1$ z največjim primanjkljajem, pri čemer so teže bazenov enake, medtem ko okno spremlja zadnjih devet dodelitev.

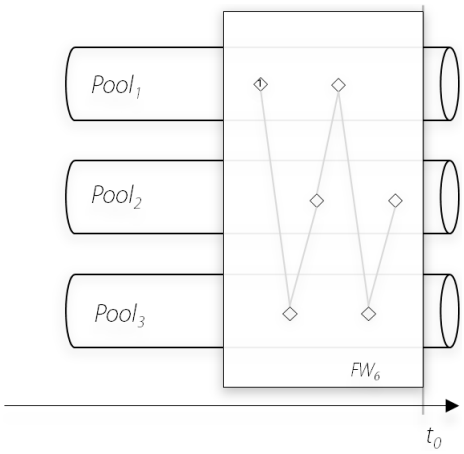
razporejanje glede na fiksen časovni interval, kjer je potrebno upoštevati spremenljivo velikost okna. V našem primeru smo se odločili, da testiranja razporejevalnika opravimo s fiksno velikostjo okna, ki lahko zaseda spremenljive časovne intervale. Bazenom se torej vire dodeljuje na podlagi zadnjih n razporejanj, kjer n predstavlja velikost pravičnega okna.

Ker v sistemu obstaja dvoje vrst opravil in računskih virov je dodeljevanje obeh ločeno. Okno podpira dodeljevanje obeh vrst opravil.

5.2.3 Vpliv velikosti okna na pravičnost razporejanja

Velikost okna določa število dodeljenih računskih virov, ki jih v oknu želimo spremljati. Iz testnih meritev smo spoznali, da niso vse velikosti oken primerne za ohranjanje pravičnosti med bazeni, saj vplivajo na neenakomerno porazdelitev virov. Pri oknih, katerih velikost je večkratnik vsote tež bazenom, se namreč pojavljajo trenutki, kjer je istemu bazenu lahko večkrat zapored dodeljen računski vir. Pojav smo opazili pri množici aktivnih bazenov, kjer imajo kandidati za dodelitev novega vira enake primanjkljaje. Razporejevalnik se v takih trenutkih odloča za isti bazen, kar ob daljšem izvajanju privede do nepravičnega razporejanja.

Podoben primer lahko zasledimo na sliki 5.3, kjer imajo bazeni enake teže in primanjkljaje ob času t_0 enake nič. Iz omenjenega naj bi sledilo, da je vseeno, kateremu izmed njih je dodeljen naslednji računski vir, kar v našem primeru ne



Slika 5.3: Vpliv velikosti okna na odločljivost razporejanja.

drži. Težava se pojavi v trenutku neodločljivosti t_0, kjer razporejevalnik daje prednost bazenu $pool_1$, iz katerega se ob novi dodelitvi odstrani vir z oznako ena (slika 5.3). Bazeni ob naslednjem razporejanju ponovno dosežejo svoje pravične deleže in ker razporejevalnik daje prednost bazenu $pool_1$, bo nov vir zopet dodeljen istemu bazenu. Vir je v našem primeru bazenu $pool_3$ dodeljen šele v tretjem razporejanju od trenutka neodločljivosti. Za razrešitev opisane težave predlagamo naslednje rešitve:

- Izbira bazena z upoštevanjem vira, ki se ob dodelitvi odstrani. Če naslednji element za odstranitev pripada bazenu izmed trenutnih kandidatov za dodelitev vira, se bazen pri trenutni izbiri izpusti.

- Izbira bazena na podlagi števila izvajajočih opravil, kjer ima izbran bazen trenutno najmanjše število opravil v izvajanju.

- Vpeljava naključne izbire bazena.

- Preprečitev opisanega problema z določitvijo velikosti okna glede na teže bazenov.

Osredotočimo se na zadnjo rešitev in predpostavimo, da je velikost okna l enaka večkratniku vsote tež bazenov:

$$l = k \cdot w \ . \tag{5.6}$$

Naj bo omenjen večkratnik zaenkrat $k = 1$, iz česar sledi, da je izvajanje pravično, če je bilo bazenu i v oknu FW_l dodeljeno natanko sf_i računskih virov.

Dodeljevanje virov stremi k pravičnemu izvajanju zato lahko pričakujemo, da bodo vsi bazeni ob določenem času dosegli svoje pravične vrednosti sf_i. Iz tega sledi, da bodo njihovi primanjkljaji enaki nič, kar pomeni da imamo za izbiro bazena na voljo več kandidatov, s čimer se pojavi problem neodločljivosti.

Zmanjšanje okna za en računski vir na velikost $l = (k \cdot w) - 1$ je bilo v meritvah učinkovito, saj se je izkazalo, da ima največji primanjkljaj ob vsakem razporejanju natanko en bazen, ki mu je dodeljenih $\lfloor sf \rfloor$ virov. Ostali bazeni presegajo svoje pravične deleže, število njihovih dodeljenih virov pa znaša natanko $\lceil sf \rceil$. Seštevek dodeljenih virov iz vseh bazenov je enak velikosti pravičnega okna.

Če povzamemo je torej učinkovita **velikost okna** z upoštevanjem večkratnika vsote tež bazenov k enaka

$$l = \left(k \cdot \sum_{i \in pools} w_i \right) - 1 \, . \tag{5.7}$$

Večino zgoraj predlaganih rešitev lahko v razporejevalnik vključimo z manjšimi spremembami pri izbiri bazena ali z implementacijo lastnega sortirnega primerjalnika. Učinkovitost predlaganih rešitev smo preizkusili v poglavju 5.3.3 pri meritvah vplivov neaktivnih bazenov na pravičnost dodeljevanja virov.

5.3 Evaluacija

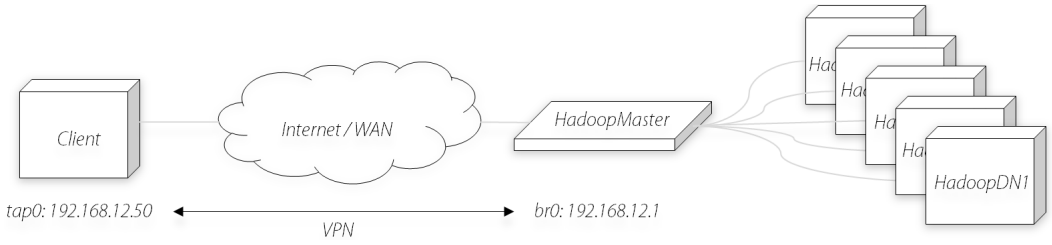
Razporejanje s pravičnim oknom smo primerjali s pravičnim razporejevalnikom in razporejevalnikom *FIFO*, ki spadata med bolj prepoznavne razporejevalnike v okolju *Hadoop*.[1] V meritvah smo primerjali kakšne izvajalne čase dosegajo pri različnih tipih poslov ter ocenjevali pravičnost njihovega dodeljevanja glede na različne parametre okolja.

5.3.1 Priprava testnega okolja

Postavitev manjše gruče šestih računalnikov je potekala v več delih. Strojno opremo okolja je sestavljal zmogljivejši računalnik, ki je upravljal dostope do gruče, storitve lokalnega omrežja in koordinacijo poslov okolja *Hadoop*. Gručo

[1] Razporejevalnik smo opisali v 4.2.1

je sestavljalo pet dodatnih delovnih vozlišč, ki so bila namenjena hranjenju podatkov in izvajanju opravil. Vozlišča so bila med seboj povezana z gigabitnim stikalom. Diagram okolja opisuje slika 5.4.



Slika 5.4: Diagram testnega okolja.

Vozlišča so bila opremljena z operacijskim sistemom *Ubuntu Server 10.10*, za upravljanje omrežnih naslovov in preslikovanje domen je skrbelo orodje *DnsMasq*. Povezljivost in dostopnost do lokalnih virov je z vzpostavitvijo infrastrukture javnih ključev v mostovnem načinu (angl. *bridge mode*) omogočalo orodje *OpenVPN*.

Na vozliščih je bila nameščena različica *Apache Hadoop 0.20.205*, ki je v trenutku postavitve okolja predstavljala najstabilnejšo različico ogrodja. Izvajanje in statistiko opravil smo spremljali preko namenskih servletov ogrodja in orodja za monitoring gruč *Ganglia*. Primer spremljanja virov vozlišč ob izvajanju opravil je podan v dodatku B.

Dodatne informacije razporejanja smo pri pravičnem razporejevalniku in razporejevalniku *FWS* pridobili z manjšimi spremembami v odseku kode, kjer poteka izbira bazenov za dodelitev računskih virov.

V gruči je bilo skupno na voljo 700GB prostora, pri čemer je vsak podatek vseboval svojo kopijo na ločenem vozlišču. Podatki so bili porazdeljeni na petih običajnih računalnikih z naslednjo konfiguracijo:

- procesor: AMD Athlon 64 3500+, 2,2GHz, 1 procesorsko jedro
- delovni spomin: 2GB
- lokalni disk: 140GB

Za samodejno postavitev in dodajanje novih vozlišč smo pripravili različne skripte, kar nam je omogočilo postavitev gruče v relativno kratkem času. Pravtako smo za testno okolje pripravili skripte za samodejni prenos izgradenj razporejevalnika, poslov *MapReduce* in zajem statistike opravil.

Začetna določitev števila računskih virov posameznega vozlišča je obsegala dva računska vira *map* in enega *reduce*, kar je vodilo do preobremenitve procesorjev, zato smo njihovo število postavili v razmerje ena proti ena. Ker je bilo v gruči prisotnih pet delovnih vozlišč, je bilo skupno število računskih virov *map* enako pet, enako število je bilo virov *reduce*.

Izvajalno okolje smo skušali v vseh razporejevalnikih poenotiti, zato smo se pri meritvah odločili, da kljub podpori zagotavljanja minimalnih deležev pri dveh izmed razporejevalnikov, slednje v meritvah ne upoštevamo. Prav tako so bile v vseh izvajanjih onemogočene prekinitve izvajanja opravil in mehanizmi zakasnjenega razporejanja.

5.3.2 Ocenjevanje pravičnosti dodelitev

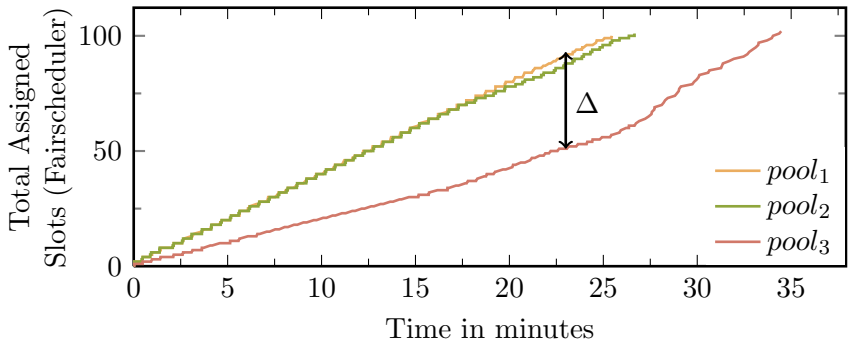
Pravičnost dodelitev računskih virov smo lahko spremljali le pri pravičnem in *FWS* razporejevalniku, saj razporejevalnik *FIFO* ne omogoča izstavljanja poslov različnim bazenom.

Razporejanje oziroma dodeljevanje virov smo označili kot pravično, če vsi bazeni od začetka meritve prejmejo enak delež računskih virov. Z upoštevanjem tež bazenov je izvajanje pravično, če so ob poljubnem času t deleži dodeljenih virov in tež enaki za vse aktivne bazene. Oceno za spremljanje pravičnosti dodelitev smo določili s funkcijo $\Delta(t)$, ki ob vsakem razporejanju meri največjo razliko v dodeljenih virih trenutno aktivnih bazenov od začetka izvajanja meritve

$$\Delta(t) = \max_{i,j \in pools} \left| \frac{sa_i(t)}{w_i} - \frac{sa_j(t)}{w_j} \right|, \; i \neq j \; , \tag{5.8}$$

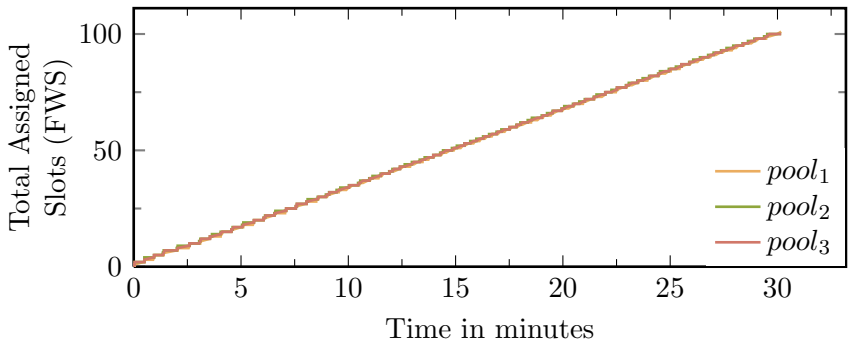
kjer sa_i in sa_j predstavljata skupno število dodeljenih virov bazenu i in j od začetka meritve do časa t, medtem ko w_i in w_j njihovi teži. Nizke vrednosti funkcije $\Delta(t)$ nakazujejo, da je razporejanje pravično, saj je dodeljevanje virov med bazene enakomernejše.

Obnašanje funkcije $\Delta(t)$ v pravičnem razporejevalniku opisuje slika 5.5, na kateri so prisotni trije grafi, ki predstavljajo skupno število dodeljenih virov posameznemu bazenu do časa t.



Slika 5.5: Število dodeljenih virov posameznega bazena od začetka izvajanja meritve in spremljanje pravičnosti s funkcijo $\Delta(t)$ za pravični razporejevalnik.

Funkcija $\Delta(t)$ ob vsakem razporejanju meri največje odstopanje v številu dodeljenih virov med bazeni. Iz zgornje slike podanega primera je razvidno, da dodeljevanje ni povsem pravično, saj je tretjemu bazenu v istem časovnem intervalu dodeljeno manj opravil kot ostalima dvema bazenoma, čeprav imajo vsi enake teže. Primer pravičnejšega razporejanja in enakomernejšega dodeljevanja virov ponazarja slika 5.6 za razporejevalnik *FWS*, kjer razpon funkcije $\Delta(t)$ niha med enim in dvema računskima viroma. V obeh primerih imajo bazeni enake teže.



Slika 5.6: Število dodeljenih virov posameznega bazena od začetka izvajanja meritve razporejevalnika *FWS*.

5.3.3 Meritve in diskusija rezultatov

Podatke meritev so sestavljali arhivi spletnih objav strani *Wikipedia*, ki so bili v gruči hranjeni z replikacijskim faktorjem dva. Velikost posamičnega datotečnega bloka je bila 64MB. Pred testiranji so bili podatki uravnoteženo porazdeljeni med vozlišča gruče.

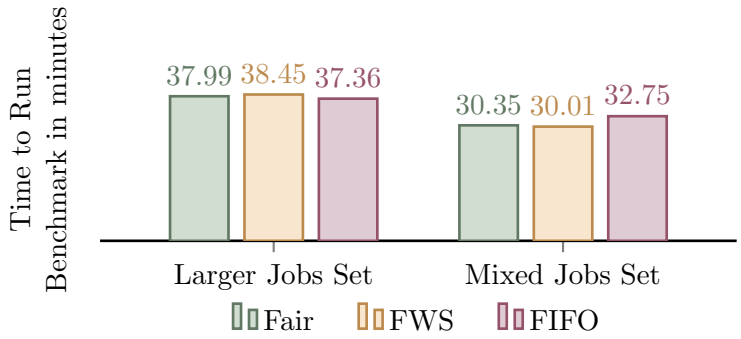
V testiranjih so bili uporabljeni trije tipi podatkovno intenzivnih poslov z različnimi števili opravil *map*. Najmanjši posel j_s je vseboval 3 opravila in se je izvajal nad podatki velikosti 192MB, srednje velik posel j_m z 10 opravili je obsegal 640MB podatkov, medtem ko je največji posel j_b zajemal 100 opravil in se izvajal nad podatki velikosti 6,25GB. Vsak posel je poleg opravil *map* vseboval tri opravila *reduce*. Kombinacije zgoraj naštetih poslov so bile sestavni del meritev, opisanih v nadaljevanju.

Izvajalni časi

V meritvah smo želeli preveriti vpliv dodeljevanja virov posameznega razporejevalnika na izvajalne čase dveh različnih množic poslov. V ta namen smo sestavili dve testni množici, ki sta vsebovali različne tipe in število poslov. Izvajalni časi so bili merjeni od izstavitve prvega posla množice vse do zaključka zadnjega posla. Povprečni izvajalni časi množic poslov so povzeti na sliki 5.7.

Prvo testno množico so sestavljali trije večji posli j_b, za katere je veljalo, da so v času izvajanja v delovnih vozliščih večinoma zasedali računske vire *map*, medtem ko sta bila oba tipa virov zasedena le pri koncu izvajanj testne množice. Za vsak razporejevalnik je bilo opravljenih pet meritev, na podlagi katerih se je izkazalo, da se povprečni časi izvajanj testnih množic pri vseh treh razporejevalnikih bistveno ne razlikujejo. Pri meritvah razporejevalnika *FWS* se je izkazalo, da je čas izvajanja večjih poslov malenkost večji kakor pri ostalih dveh razporejevalnikih (slika 5.7, *Larger Jobs Set*).

Množica mešanih poslov (slika 5.7, *Mixed Jobs Set*) je bila sestavljena z namenom preučevanja vpliva skupnega prepletanja izvajanj opravil *map* in *reduce* na izvajalne čase pri različnih razporejevalnikih. Množica je predstavljala približek izvajanja poslov v realnih okoljih. Sestavljena je bila iz 23 manjših poslov j_s in 10 srednjih poslov j_m, ki so bili naključno porazdeljeni v tri bazene. Meritve z isto množico poslov so bile opravljene v petih ponovitvah za vsakega izmed razporejevalnikov. Rezultati meritev so podobno kot v prejšnjem primeru pokazali, da se izvajalni časi razporejevalnikov v našem primeru bistveno ne razlikujejo. Najboljši izvajalni čas množice mešanih poslov je dosegla naša implementacija razporejevalnika *FWS*, medtem ko se je najslabše odrezal razporejevalnik *FIFO* s skoraj tri minutno zakasnitvijo. Le-



Slika 5.7: Povprečni izvajalni časi za testno množico treh večjih poslov j_b in mešano množico sestavljeno iz poslov j_s in j_m. Oznake $Fair$, FWS in $FIFO$ predstavljajo razporejevalnike za katere smo opravljali meritve.

to pripisujemo predvsem različnemu zaporedju izvajanj poslov, kar je vodilo do drugačnega razporeda izvajanj opravil *map* in *reduce*. V primerjavi z ostalima razporejevalnikoma predvidevamo, da so zakasnitve nastale predvsem v trenutkih preobremenjenosti vozlišč.

Izmerjeni časi so v obeh testnih primerih pričakovani, saj so podobne rezultate brez uporabe zakasnjenega razporejanja za pravični in *FIFO* razporejevalnik dosegli tudi v članku [39] (slika 12). Na podlagi tega članka lahko sklepamo, da bi bila razlika med *FIFO* razporejevalnikom in ostalima dvema z uporabo zakasnjenega razporejanja še bolj očitna. Meritve, ki vključujejo omenjeni mehanizem v testih nismo upoštevali, saj so posli v gruči že v osnovi dosegali dokaj visoko lokalnost. Vsako opravilo je bilo namreč prisotno na dveh od petih vozlišč.

Pravičnost dodeljevanja računskih virov

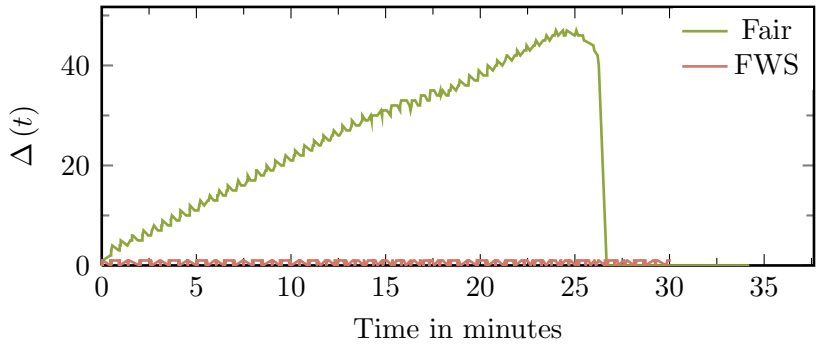
Za spremljanje pravičnega razporejanja smo v meritvah uporabili funkcijo $\Delta(t)$, ki smo jo predstavili v 5.3.2 in katera opisuje največjo razliko v dodeljenih virih med aktivnimi bazeni. Na sliki 5.8 je funkcija $\Delta(t)$ opisana za pravični in *FWS* razporejevalnik, medtem ko smo razporejevalnik *FIFO* izpustili, saj ne omogoča deljenja virov med več uporabnikov oziroma vrst.

Pravičnost smo v posamezni meritvi spremljali nad izvajanjem testne množice, ki je za vsakega izmed treh bazenov vsebovala posel j_b. Skupno je bilo v vsaki meritvi tako prisotnih 300 opravil. Posli so bili v izvajanje izstavljeni na začetku meritve, kar pomeni, da je bil vsak izmed bazenov aktiven vse do

zaključka zadnjega opravila v bazenu.

V meritvi pravičnega razporejevalnika s slike 5.8 (Fair) lahko opazimo, da se funkcija $\Delta(t)$ povečuje vse do 25. minute izvajanja, kjer doseže razliko 46 računskih virov, kar v celotni meritvi predstavlja $15,3$ % vseh dodeljenih virov. Podatki meritev so pokazali, da je vzrok za naraščajoči trend funkcije $\Delta(t)$ zapostavljanje dodeljevanja virov bazenu $pool_3$, kar predstavlja problem, ki izhaja iz dodeljevanja virov na podlagi števila opravil, ki so trenutno v izvajanju. V našem primeru predvidevamo, da se razlika povečuje predvsem v trenutkih, ko imata vsaj dva bazena isto število opravil v izvajanju in tako predstavljata kandidata za dodelitev naslednjega vira. V implementaciji pravičnega razporejevalnika je mogoče zaslediti, da se v omenjenih trenutkih računski vir dodeli bazenu glede na njegovo ime[2], kar pa ob večjem številu takšnih trenutkov lahko privede do zapostavljanja določenega bazena. Kljub temu da v našem primeru sicer nismo merili trenutnih opravil v izvajanju, lahko na sliki 5.5 vidimo potek dodeljevanja in naraščajočo razliko med tremi bazeni, katerih podatki izhajajo iz iste meritve. Po preteku 26. minute zasledimo strm padec funkcije na vrednost 0, ki nakazuje zaključek izvajanj opravil v bazenih $pool_1$ in $pool_2$.

Drugi graf na sliki 5.8 predstavlja funkcijo $\Delta(t)$ za implementacijo razporejevalnika *FWS* z velikostjo okna 8 dodelitev. Funkcija skozi celoten čas izvajanja ohranja vrednosti 0 in 1, kar pomeni, da je razlika v dodeljenih virih med prisotnimi bazeni enaka največ 1 računski vir.



Slika 5.8: Pravičnost dodeljevanja virov v izvajanju pravičnega razporejevalnika in razporejevalnika *FWS*, pri čemer je pravičnost dodeljevanja določena s funkcijo $\Delta(t)$.

[2]Bazen je sicer izbran glede na čas izstavitve bazena, ki pa v implementaciji v vsakem primeru vrača vrednost nič zato je odločitev sprejeta glede na ime bazena.

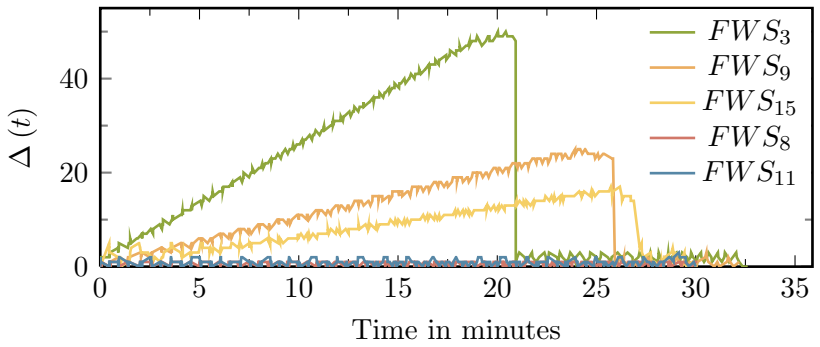
Iz podanih meritev lahko razberemo, da smo z implementacijo razporejevalnika *FWS* dosegli pravičnejše dodeljevanje virov kakor je to prisotno pri pravičnem razporejevalniku. Pri tem naj omenimo, da v obeh meritvah nismo zajeli primera neaktivnosti bazenov in naključnih prihodov poslov, ki lahko dodatno vplivajo na potek razporejanja.

Vpliv velikosti okna na pravičnost dodeljevanja virov v FWS

Vpliv velikosti pravičnega okna nad dodeljevanjem razporejevalnika smo merili nad tremi bazeni z enakimi težami, kjer smo na začetku meritve v vsakega izstavili v izvajanje posel j_b. Pravičnost dodeljevanj smo opravili s funkcijo $\Delta(t)$, katere rezultati so prikazani na sliki 5.9.

Za velikost okna 3 lahko iz grafa opazimo konstantno večanje funkcije $\Delta(t)$ vse do vrednosti 50 dodelitev, kar je posledica premajhne velikosti okna in delno tudi trenutkov neodločljivosti, katerih pojav smo opisali v poglavju 5.2.3. Graf izvajanja okna z velikostjo 9 dodelitev sicer prikazuje manjšo razliko v dodeljenih virih med bazeni, kljub temu pa doseže skorajda razliko 25 virov, kar v meritvi predstavlja $8,3\,\%$ vseh dodeljenih virov.

Iz grafov lahko razberemo, da je bilo najpravičnejše dodeljevanje doseženo pri uporabi okna velikosti 8 in 11 dodelitev (FWS_8, FWS_{11}), kjer je bila največja razlika med bazeni 2 računska vira. Podobne rezultate smo dosegli tudi pri velikosti okna 20 in 23, ki pa ju zaradi preglednosti nismo umestili v graf. Bazeni so v zgornjih primerih imeli teže enake ena.



Slika 5.9: Vpliv velikosti okna na pravičnost dodeljevanja opravil, pri čemer je vsaka meritev izvedena nad tremi aktivnimi bazeni z enakimi težami.

V enačbi 5.7 smo omenili, da je učinkovita velikost okna odvisno od faktorja k in vsote tež bazenov. V primeru FWS_8 se je pokazalo, da je pravičnost

dodeljevanja dosežena že pri $k = 3$, zato smo dodatno izmerili potek dodeljevanja v primeru različnih tež bazenov. Tako kot pri enakih težah bazenov se je pokazalo, da se z upoštevanjem enačbe okna pravičnost dobro ohranja in dosega majhne vrednosti funkcije $\Delta(t)$.

Iz opravljenih meritev je razvidno, da se ob aktivnih bazenih najbolje obnesejo velikosti okna, ki upoštevajo teže bazenov in so določena z enačbo 5.7. Izvajanje je bilo pravično za vse velikosti okna, katere faktor $k > 2$.

Vpliv neaktivnih bazenov na pravičnost dodeljevanja virov v FWS

Bazeni so lahko med izvajanjem tudi v neaktivnem stanju, kar pomeni, bodisi da so se vsi njihovi posli že zaključili bodisi da v bazen ni bilo izstavljeno v izvajanje nobenega posla. V meritvi se je izkazalo, da se trenutki neodločljivosti pojavljajo kljub upoštevanju enačbe 5.7 za izračun velikosti okna.

V meritvi smo želeli preveriti obnašanje okna v prisotnosti neaktivnih bazenov in trenutkov neodločljivosti. Testna konfiguracija, ki ustreza opisanim pogojev je bila sestavljena iz štirih bazenov z enakimi težami, kjer je bila velikost okna določena z enačbo 5.7 in je znašala 15 dodelitev. Trenutki neodločljivosti se v podanem primeru pojavijo, ko je v oknu trem bazenom dodeljeno po pet virov, zato smo četrti bazen pustili neaktiven.

Za reševanje trenutkov neodločljivosti smo v meritvi upoštevali vse predlagane rešitve iz poglavja 5.2.3 in izmerili njihovo učinkovitost. Kot najučinkovitejša se je izkazala rešitev, pri kateri smo bazen izmed kandidatov za dodelitev naslednjega vira izbrali naključno.

Učinkovitost uporabljene rešitve se je pokazala v povečani pravičnosti, kjer je funkcija $\Delta(t)$ nihala med vrednostima 0 in $1,3$ % vseh dodeljenih virov, kar je bistveno nižje kot pri ostalih rešitvah, ki smo jih predlagali. Rešitev s preprečevanjem dodelitve vira bazenu, katerega vir je pravkar zapustil okno, je s $5,3$ % razliko v dodelitvah dosegla slabše rezultate, medtem ko se je reševanje trenutkov neodločljivosti glede na število opravil v izvajanju odrezalo bolje in doseglo največjo razliko izmed vseh dodeljenih virov v višini $3,6$ %.

Iz opisanega sledi, da smo z upoštevanjem enačbe za izračun velikosti okna in mehanizmom naključne izbire bazena ob trenutkih neodločljivosti dosegli bistveno izboljšavo v pravičnejšem dodeljevanju virov. Največja razlika v dodeljenih virih med bazeni je znašala $1,3$ % kar znaša $13,4$ % izboljšavo v primerjavi s pravičnim razporejevalnikom. Slednji je v isti meritvi dosegel razliko med bazeni v višini $14,7$ % vseh dodeljenih virov.

5.4 Sklepne ugotovitve

Nadgradnja pravičnega razporejevalnika s časovno komponento okna se je izkazala za učinkovito rešitev pri zagotavljanju pravičnejšega in enakomernejšega dodeljevanja računskih virov. Z novim razporejevalnikom odzivnosti poslov sicer nismo uspeli izboljšati, a smo z razporejanjem, ki temelji na lokalni zgodovini zadnjih dodelitev, kljub temu dosegli pravičnejše dodeljevanje virov med prisotne bazene.

V meritvah pravičnega razporejevalnika je možno opaziti, da je v istem intervalu nekaterim bazenom kljub enakim težam dodeljeno manj virov kot ostalim (slika 5.5). Z ohranjanjem pravičnega in bolj striktnega dodeljevanja smo v novem razporejevalniku dosegli, da se viri med bazeni dodeljujejo bolj enakomerno, kar posledično pomeni enakomernejše in predvidljivejše izvajanje njihovih poslov (slika 5.6).

Meritve so pokazale, da zamisel in vpeljava strukture okna v razporejanje vsebuje pomanjkljivosti, ki jih ob dodelitvi novega vira predstavljajo trenutki neodločljivosti. V primeru aktivnosti vseh bazenov smo težavo rešili s predhodnim določanjem velikosti okna glede na teže bazenov. Rešitev se sicer zelo dobro obnese ob aktivnih bazenih, manj pa v primeru, ko nekateri izmed njih niso v uporabi oziroma preidejo v neaktivno stanje, kar ponovno povzroči njihovo pojavitev. Problem smo poleg upoštevanja enačbe 5.7 za velikost pravičnega okna razrešili z naključno izbiro bazena izmed kandidatov za dodelitev naslednjega vira.

Kombinaciji zgornjih rešitev sta v vseh opravljenih meritvah dosegali pravičnejše dodeljevanje virov, kot je to prisotno pri obstoječem pravičnem razporejevalniku. Navkljub dobrim rezultatom bi bilo v nadaljnjih meritvah smiselno preučiti potek dodeljevanja v odvisnosti od velikosti okna in naključnega prihoda večjih poslov. Pri preveliki velikosti okna in prihodu večjega posla v dalj časa neaktiven bazen, se namreč lahko zgodi, da so vsi razpoložljivi viri dodeljeni slednjemu, kar lahko privede do zastojev poslov v ostalih bazenih.

Meritve smo opravljali nad manjšo gručo običajnih računalnikov z omejenimi viri. Dodaten vpogled v delovanje in učinkovitost razporejevalnika *FWS* bi lahko dosegli z razporejanjem v zmogljivejši gruči računalnikov in simulacijo produkcijskih sledi aplikacij iz realnih izvajalnih okolij.

Poleg omenjenega bi bilo v nadaljnjih verzijah razporejevalnika smiselno preučiti ustreznost in uporabnost vpeljave razporejanja na podlagi časov zaključka poslov. Slednje bi služilo predvsem aplikacijam, ki zahtevajo strožje časovne omejitve in pri katerih je doseganje časovnih rokov ključnega pomena za delovanje sistema.

Trenutna različica razporejevalnika *FWS* deluje v vseh ogrodjih *Apache Hadoop*, kjer obstaja možnost razporejanja z obstoječim pravičnim razporejevalnikom. Večja novost novejše *beta* različice ogrodja *Apache Hadoop MapReduce v2* (*YARN*) je bila sprememba celotne arhitekture sistema z uvedbo novega modela upravljanja virov. Razporejevalnik *FWS* v omenjeni različici še nismo testirali, vendar verjamemo, da bi lahko bilo razporejanje v tem okolju z natančnejšimi informacijami o stanju virov še učinkovitejše.

Poglavje 6

Zaključek

V diplomskem delu smo se podrobneje seznanili z modelom *MapReduce*, ki predstavlja enega izmed bolj prepoznavnih modelov za podatkovno intenzivno in porazdeljeno računanje. Prepoznavnost modela se je pokazala v številnih implementacijah, zato smo v drugem poglavju povzeli le trenutno najbolj razširjena ogrodja. Največ pozornosti smo posvetili ogrodju *Apache Hadoop* in njegovemu razporejanju, katerega skupine razporejevalnikov smo opredelili v četrtem poglavju in jih nekaj tudi opisali. V zaključnem delu smo izvedli nadgradnjo pravičnega razporejevalnika, s katero sicer nismo dosegli krajše odzivnosti poslov, a smo kljub temu v primerjavi s pravičnim razporejevalnikom dosegli bistveno izboljšavo v pravičnosti dodeljevanja virov med uporabnike sistema in posledično enakomernejše izvajanje njihovih poslov. Izboljšavo smo dosegli z vpeljavo zamisli razporejanja s pravičnim oknom in spremljanjem zadnjih dodelitev virov. Pomanjkljivost vpeljane zamisli se je pokazala v trenutkih enakih prioritet bazenov oziroma v trenutkih neodločljivosti, ki so bili vzrok za neenakomerno porazdelitev virov med bazene. Izmed predlaganih možnosti za rešitev težave se je kot najučinkovitejša izkazala določitev velikosti okna glede na teže bazenov in izbira naključnega bazena iz množice kandidatov za dodelitev novega vira.

Modelom in sistemom za podatkovno intenzivno računanje bo v prihodnjih letih glede na naraščajoče količine podatkov skoraj zagotovo posvečeno še veliko pozornosti. Po ocenah vodilnih analitskih podjetij kot so IDC, McKinsey, Gartner in Forrester naj bi količina podatkov v naslednjih letih strmo naraščala. Podjetje McKinsey ocenjuje, da se bo njihov obseg do leta 2021 v primerjavi z letom 2009 povečal za 44-krat z letno rastjo 40 %. Trg omenjenih tehnologij naj bi se iz trenutno 9,1 milijarde dolarjev povzpel na 86,4 milijarde dolarjev, kar naj bi predstavljalo 11 % vseh vloženih sredstev na

področju informacijske tehnologije [29]. Na razvoj omenjenih sistemov bo po našem mnenju dodatno vplivalo zanimanje majhnih in srednje velikih podjetij, ki bodo v svojih podatkih prepoznala potencialni vir koristnih informacij za pomoč pri odločanju in pregledu poslovanja. S tem mislimo predvsem na uporabo preprostih in odzivnih aplikacij na področju poslovne analitike, katerih učinkovitost je v primeru velikih zbirk podatkov tesno povezana z razvojem sistemov in ogrodij za podatkovno intenzivno računanje.

Dodatek A

Izvorna koda razporejevalnika FWS

Za dostop do hrambe kontaktirajte mene oziroma dr. Andreja Brodnika.
`https://lusy.fri.uni-lj.si/svn/dipl-mu/`

Dodatek B

Spremljanje izvajanja opravil z orodjem Ganglia

Ganglia ponuja širok izbor metrik za spremljanje sistemskih parametrov vozlišč. Orodje omogoča integracijo z ogrodjem *Hadoop* in spremljanje njegovega izvajanja kot npr. zasedenost računskih virov, parametre porazdeljenega datotečnega sistema *HDFS* in ostalo.

Slika B.1 prikazuje kumulativni metriki vozlišč, ki opisujeta stanje omrežja in procesorsko obremenjenost gruče. Kot primer smo podali izvajanje treh poslov s skupno tristo opravili *map* in šestimi opravili *reduce*. Iz grafa je lepo razviden prehod v stopnjo *reduce*, ki ga v omrežnem grafu zaznamuje povečana pasovna širina omrežja, medtem ko se v grafu procesorske aktivnosti odraža v nižji obremenjenosti vozlišč. Prehod je označen z zeleno črto.

Slika B.1: Prikaz metrik pasovne širine omrežja in procesorske obremenjenosti v izvajanju *MapReduce* opravil.

Slike

2.1 Primer uporabe višjenivojske funkcije *map* in *fold*. *Map* kot argument sprejme funkcijo f in jo aplicira na vse elemente seznama, medtem ko *fold* s funkcijo g rekurzivno združuje rezultate funkcije f in predhodne vmesne rezultate. 7
2.2 Prikaz izvajanja opravil *map* in *reduce*. 8
2.3 Trend naraščanja podatkov in števila obdelav pri podjetju *Google*. 11
2.4 Seznam podjetij v povezavi s projektom *Hadoop* [12]. 12

3.1 Običajen primer gruče s prisotnostjo glavnega in delovno podatkovnih vozlišč z delitvijo na računski in podatkovni sloj. . . . 22
3.2 Prikaz bralno-pisalnih dostopov v *HDFS*. 23
3.3 Izvajalno okolje modela *MapReduce* in osnovni gradniki ogrodja *Hadoop*, ki smo jih spoznali v poglavju 4.3.2. 25

4.1 Primer hierarhije razporejanja in pravičnega dodeljevanja virov. 30
4.2 Primer funkcije koristnosti za različne parametre β. 31
4.3 Dodelitev opravila vozlišču $TaskTracker$ 1 z upoštevanjem ravni lokalnosti *Rack-local*. 38
4.4 Dekompozicija posodabljanja parametrov razporejanja in prekinitve opravil. 41

5.1 Dodeljevanje računskih virov z uporabo pravičnega okna. 49
5.2 Dodelitev računskega vira bazenu $Pool_1$ z največjim primanjkljajem, pri čemer so teže bazenov enake, medtem ko okno spremlja zadnjih devet dodelitev. 53
5.3 Vpliv velikosti okna na odločljivost razporejanja. 54
5.4 Diagram testnega okolja. 56
5.5 Število dodeljenih virov posameznega bazena od začetka izvajanja meritve in spremljanje pravičnosti s funkcijo $\Delta(t)$ za pravični razporejevalnik. 58

5.6 Število dodeljenih virov posameznega bazena od začetka izvajanja meritve razporejevalnika *FWS*. 58
5.7 Povprečni izvajalni časi za testno množico treh večjih poslov j_b in mešano množico sestavljeno iz poslov j_s in j_m. Oznake $Fair$, FWS in $FIFO$ predstavljajo razporejevalnike za katere smo opravljali meritve. 60
5.8 Pravičnost dodeljevanja virov v izvajanju pravičnega razporejevalnika in razporejevalnika *FWS*, pri čemer je pravičnost dodeljevanja določena s funkcijo $\Delta(t)$. 61
5.9 Vpliv velikosti okna na pravičnost dodeljevanja opravil, pri čemer je vsaka meritev izvedena nad tremi aktivnimi bazeni z enakimi težami. 62

B.1 Prikaz metrik pasovne širine omrežja in procesorske obremenjenosti v izvajanju *MapReduce* opravil. 70

Literatura

[1] *Hadoop Fair Scheduler Design Document*, 2010. Dostopno na: https://issues.apache.org/jira/secure/attachment/12457515/fair_scheduler_design_doc.pdf.

[2] *HDFS User Guide*, 2010. Dostopno na: http://hadoop.apache.org/common/docs/r0.20.2/hdfs_user_guide.html.

[3] *CapacityScheduler Guide*, 2011. Dostopno na: http://hadoop.apache.org/docs/r0.20.205.0/capacity_scheduler.html.

[4] *Gluster Filesystem 3.3*, 2011. Dostopno na: http://download.gluster.com/pub/gluster/glusterfs/qa-releases/3.3-beta-2/Gluster_Hadoop_Compatible_Storage.pdf.

[5] Apache Hadoop, 2012. Dostopno na: http://hadoop.apache.com/.

[6] Disco - massive data, minimal code, 2012. Dostopno na: http://discoproject.org/about.

[7] *Erlang*, 2012. Dostopno na: http://www.erlang.org/faq/introduction.html.

[8] *LexisNexis Risk Solutions - HPCC Systems*, 2012. Dostopno na: http://hpccsystems.com/community/white-papers.

[9] NERSC R&D - Hadoop, 2012. Dostopno na: http://www.nersc.gov/research-and-development/testbeds/hadoop/.

[10] *Oracle: Big Data for the Enterprise*, 2012. Dostopno na: `http://www.oracle.com/us/products/database/big-data-for-enterprise-519135.pdf`.

[11] A. Alexandrov, S. Ewen, M. Heimel, F. Hueske, O. Kao, V. Markl, E. Nijkamp, D. Warneke. MapReduce and PACT - Comparing Data Parallel Programming Models. *BTW*, str. 25–44, 2011.

[12] M. Aslett. The Hadoop Ecosystem, 2012. Dostopno na: `http://www.informatica.com/hadooptuesdays`.

[13] E. Baldeschwieler. Hadoop applications at Yahoo!, 2009. Dostopno na: `http://www.cloudera.com/blog/2009/12/hadoop-world-hadoop-applications-at-yahoo/`.

[14] Y. Bu, B. Howe, M. Balazinska, M. D. Ernst. HaLoop: Efficient Iterative Data Processing on Large Clusters. *PVLDB*, Vol. 3, No. 1, str. 285–296, 2010.

[15] Luca Cardelli, Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, Vol. 17, No. 4, str. 471–522, 1985.

[16] J. Dean. *Building Software Systems at Google and Lessons Learned*, 2010. Dostopno na: `http://www.stanford.edu/class/ee380/Abstracts/101110-slides.pdf`.

[17] J. Dean, S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *OSDI 2004*, str. 137–150, 2004.

[18] J. D. Dhok. *Learning Based Admission Control and Task Assignment in MapReduce*. Magistrsko delo, 2010.

[19] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S. H. Bae, J. Qiu, G. Fox. Twister: a runtime for iterative MapReduce. *HPDC*, str. 810–818, 2010.

[20] E. Friedman, P. M. Pawlowski, J. Cieslewicz. SQL/MapReduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *PVLDB*, Vol. 2, No. 2, str. 1402–1413, 2009.

[21] Gu, Y. and Lu, L. and Grossman, R. and Yoo, A. Processing massive sized graphs using Sector/Sphere. *Many-Task Computing on Grids and Supercomputers (MTAGS), 2010 IEEE Workshop on*, str. 1 –10, 2010.

[22] J. Hammerbacher. 10 Common Hadoop-able Problems, 2010. Dostopno na:
`http://www.cloudera.com/resource/10_common_hadoop-able_problems_webinar_2010_hammerbacher/`.

[23] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, Vol. 21, No. 3, str. 359–411, 1989.

[24] M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *EuroSys*, str. 59–72, 2007.

[25] H. Karloff, S. Suri, S. Vassilvitskii. A model of computation for MapReduce. *SODA 10*, str. 938–948, 2010.

[26] R. Lämmel. Google's MapReduce programming model - Revisited. *Science of Computer Programming*, Vol. 70, No. 1, str. 1–30, 2008.

[27] J. Lin, C. Dyer. *Data-Intensive Text Processing with MapReduce*. Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers, 2010.

[28] J. W. S. Liu. *Real-time systems*. Prentice Hall, 2000.

[29] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, A. H. Byers. Big data: The next frontier for innovation, competition, and productivity, 2011. Dostopno na:
`http://www.mckinsey.com/insights/mgi/research/technology_and_innovation/big_data_the_next_frontier_for_innovation`.

[30] A. C. Murthy. Hadoop at Yahoo!, 2010. Dostopno na:
`http://www.cloudera.com/resource/hw10_video_hadoop_at_yahoo_ready_for_business/`.

[31] M. Olson. The Community Effect, 2011. Dostopno na:
`http://www.cloudera.com/blog/2011/10/the-community-effect/`.

[32] L. T. X. Phan, Z. Zhang, B. T. Loo, I. Lee. Real-time MapReduce Scheduling. Tehnično poročilo MS-CIS-10-32, University of Pennsylvania, 2010.

[33] L. T. X. Phan, Z. Zhang, Q. Zheng, B. T. Loo, L. Lee. An empirical analysis of scheduling techniques for real-time cloud-based data processing. *SOCA 10*, str. 1–8, 2011.

[34] T. Sandholm, K Lai. Dynamic proportional share scheduling in hadoop. *JSSPP*, str. 110–131, 2010.

[35] X. Su, G. Swart. Oracle in-database Hadoop: when MapReduce meets RDBMS. *SIGMOD Conference*, str. 779–790, 2012.

[36] R. Taylor. An overview of the Hadoop/MapReduce/HBase framework and its current applications in bioinformatics. *BMC Bioinformatics*, Vol. 11, , 2010.

[37] D. Thain, T. Tannenbaum, M. Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, Vol. 17, No. 2-4, str. 323–356, 2005.

[38] T. White. *Hadoop - The Definitive Guide: Storage and Analysis at Internet Scale (2. ed.)*. O'Reilly, 2011.

[39] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, I. Stoica. Job scheduling for multi user mapreduce clusters. Tehnično poročilo, EECS Department, University of California, Berkeley, 2009.

[40] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. *EuroSys*, 2010.

[41] M. Zaharia, A. Konwinski, A. Joseph, R. Katz, I. Stoica. Improving mapreduce performance in heterogeneous environments. *OSDI 08*, 2008.