UNIVERSITETET I OSLO
Institutt for informatikk

Crawlerarkitektur og Adaptive metoder

Masteroppgave

Vikramjeet Singh Jassal

1. august 2006

Sammendrag

Weben har siden dens opprinnelse på nittitallet gjennomgått en utrolig utvikling med tanke på vekst. Denne veksten har i den senere tid i vesentlig grad vært på dynamisk innhold. Aktuell informasjon som endrer seg daglig er mer og mer vanlig. Crawlere er fellesbetegnelse på applikasjoner som traverserer weben for å presentere ønskelig informasjon gjennom søkemotorer.

Med den sterke veksten i dynamisk innhold, er det viktig for crawleren å ha metoder for å behandle denne type innhold mest mulig effektivt. I et sterkt dynamisk miljø, er det avgjørende for kvalitet og ressursbruk å kunne forutsi hvilke elementer som burde og ikke burde oppdateres. Metoder for å estimere riktige tidspunkter og metoder for å synkronisere dynamisk innhold, blir med en fellesbetegnelse kalt for adaptive metoder.

I denne oppgaven blir generelle utfordringer for crawlere tatt opp, og løsninger på noen av disse presentert med tanke på støtte for adaptiv crawling. For å lette videre utvikling av adaptive tilnærminger presenteres en alternativ og mer konfigurerbar crawlerarkitektur. En konkret adaptiv algoritme blir implementert, testet og dokumentert i sammenheng med FAST-Crawleren.

Innhold

1 Introduksjon **6**
1.1 Problembeskrivelse 7
1.2 Fremgangsmetode 7
1.3 Begrensninger 7
1.4 Overblikk over oppgaven 8
1.5 Takk til 8

2 World Wide Web **10**
2.1 Bakgrunn 10
2.2 Utviklingen 11
2.3 Uforutsigbarhet 11
2.4 Søkemotorene 11
2.5 Crawlere 12
2.5.1 Periodisk vs. Kontinuerlig crawler 13
2.5.2 «Shadowing» vs. «In-Place» 14
2.5.3 Vurdering 15
2.6 Oppsummering 15

3 Grunnleggende Crawling **17**
3.1 Introduksjon 17
3.2 Generell nettverksproblematikk 17
3.2.1 Begrensede nettverksressurser 17
3.2.2 Variabel tjenestekvalitet 18
3.2.3 Crawlertaktikk 18
3.2.4 Inkonsistente nettverkskonfigurasjoner 18
3.3 Problemer knyttet til DNS 19
3.3.1 DNS og nedetid 19
3.3.2 Feil i DNS 19
3.4 Variasjon i HTTP-implementasjoner 20
3.4.1 HTTP-headere 20
3.4.2 Tjenersidebehandling av headeren 21

3.4.3 Allsidig dekning 21
3.4.4 Redefinisjon av feilkoder 22
3.4.5 Feildata i responsheaderen 22
3.4.6 Robots.txt spesifikasjonen 23
3.5 HTML koding 25
3.6 Serverapplikasjoner 25
3.6.1 Sesjonsidentifikatorer 25
3.6.2 Interne stier 25
3.6.3 Dynamiske sider 26
3.7 Duplikatdeteksjon 26
3.7.1 Duplikatdeteksjon ved hjelp av RSYNC 27
3.8 Utilgjengelig web - The Deep Web 28
3.9 Vurdering 29
3.10 Oppsummering 30

4 Adaptive Metoder 31
4.1 Introduksjon 31
4.2 Rammeverk 32
4.2.1 Poissonfordelingen 33
4.2.2 Utviklingsmodell for en base 33
4.3 Presentasjon av ulike synkroniseringsmetoder 34
4.4 Vurdering av ulike synkroniseringsrekkefølger 36
4.4.1 Fast rekkefølge 36
4.4.2 Variabel rekkefølge 36
4.4.3 Sterk variabel rekkefølge 37
4.4.4 Sammenligning av ulike synkroniseringsrekkefølger 37
4.5 Vurdering av ulike ressursallokeringsmetoder 37
4.5.1 Introduksjon 37
4.5.2 Uniform vs. Elementavhengig ressursallokering . . . 37
4.5.3 Praktisk tolkning av Uniform vs. Elementavhengig ressursallokering 38
4.6 Optimal ressursallokering 39
4.6.1 Introduksjon 39
4.6.2 Metode 40
4.6.3 Matematisk utfordring 41
4.6.4 Konklusjon 41
4.7 Estimering av forandringsrater 42
4.7.1 Introduksjon 42
4.7.2 Forberedende momenter 42
4.7.3 Forandringsrater i en Crawlerkontekst 44
4.8 Bayesiansk slutning 44

4.8.1 Introduksjon 44
4.8.2 Metode 44
4.9 Konklusjon 46

5 Implementasjon av adaptiv crawling for FAST-Crawleren 47
5.1 Introduksjon 47
5.2 Arkitektur og Design 47
5.2.1 FAST Crawlerarkitektur 47
5.2.2 Modul for Bayesiansk Slutning 50
5.3 Implementasjon 51
5.3.1 Introduksjon 51
5.3.2 Modul for Bayesiansk Slutning 51
5.3.3 Montering i Fast Crawleren 61
5.4 Eksperiment 62
5.4.1 Introduksjon 62
5.4.2 Konfigurering av FAST crawler 62
5.4.3 Gjennomføring 67
5.4.4 Bayesiansk Slutning. Teori vs. Praksis 72
5.5 Oppsummering 72

6 Modulær crawlerarkitektur 74
6.1 Introduksjon 74
6.2 Krav til Crawlerarkitekturen 74
6.2.1 Adaptive crawlertilpasninger 75
6.2.2 Duplikatdeteksjon i en tilpasset crawler 76
6.2.3 Nedlasting i en tilpasset crawler 78
6.3 Forslag til Crawlerarktitektur 79
6.4 Oppsumering 81

7 Konklusjon og videre arbeid 83
7.1 Oppnådde resultater 83
7.2 Fremtidig arbeid 84

Figurer

4.1 Illustrerer hvordan lokalindekser oppdateres 31
4.2 Plot av optimal ressursallokeringsmetode 39

5.1 FAST's crawlerarkitektur 48
5.2 Klassediagram over modul for Bayesiansk Slutning 50
5.3 Organisering av objekter under kjøring 52

6.1 Klassediagram over rammeverk for forespørselsbehandling 81

Listings

3.1 Forespørrende header 20
3.2 Forespørrende header med range parameter 21
3.3 Responderende header 22
3.4 http://wap.vg.no/robots.txt 24
5.1 Kildekode for page modulen 54
5.2 Kildekode for pageHandler modulen 57
5.3 Kildekode for bayesian modulen 60
5.4 Oppstart av Crawler 63
5.5 Bayesiansk oversikt for startsiden.no 68

Kapittel 1

Introduksjon

Informasjonen som befinner seg på internett har hatt en omfattende vekst de 2 siste tiårene. Det totale omfanget av websider vokser med over 7 millioner sider hver dag [15]. Dette forsterker inntrykket av at weben ikke kun består av statisk faktainformasjon, men også inneholder en større mengde av dagsaktuell og dynamisk innhold.

Når antall elementer er så stort er det intuitivt en utfordring å finne den rette informasjonen man er ute etter. Har man ikke en konkret webadresse til ressursen, er det ikke sikkert man er istand til å lokalisere den rette siden ved å navigere gjennom andre sider. Det er for å forenkle oppgaven med å finne ressurser på internett, søkemotorene utfører oppdateringer og behandlinger av data. Man kan tenke seg et scenario:

- En person er ute etter værmelding for sitt lokalmiljø og for en gitt tid. Personen vet ikke om en konkret nettside eller adresse hvor denne informasjonen er tilgjengelig. Vedkommende velger å søke på stedsnavn og dato på en søkemotor som personen kjenner til. Men hvordan kan en søkemotor være istand til å besøke hele informasjonskatalogen på internett idet personen trykker på søkeknappen?

Ingen eksisterende maskinvare eller nettverkstopologi slik som de eksisterer idag, gir muligheten til å gjennomføre scenario ovenfor. Søkemotorene bruker crawlere til enhver tid å oppdatere lokale kolleksjoner, som skal fungere som mest mulig komplette speilbilder av relevant informasjon på nettet. Når en bruker gjør et søk så er det blant innholdet i søkemotorenes kolleksjoner og ikke på faktisk innhold som befinner seg på nettet. Det kan derfor ofte skje at de lokale kolleksjonene inneholder informasjon som ikke lenger eksisterer på internett, og dermed gir

resultater som egentlig ikke finnes. Det foreligger til enhver tid slike inkonsistenser mellom innhold på internett og innhold hos søkemotorene.

Det er høyt prioritert for søkemotorer å gjøre denne inkonsistenten minimal. Idag foreligger det flere algoritmer for å forutsi hvilke websider som er forandret eller ikke. Metodene blir med en fellesbetegnelse kalt for adaptive metoder som alle tar sikte på å gjøre crawleren istand til intelligent å kunne forutsi hvilke websider som er forandret og burde oppdateres i de lokale kolleksjonene. Denne oppgaven tar sikte på å beskrive eksisterende forskningsarbeider i forbindelse med adaptiv crawling. I samarbeid med Fast Search and Transfer AS og FASTs konfigurerbare crawler (*FAST-Crawleren*) blir det gjort et eksperiment for å teste og vurdere aktuelle algoritmer.

1.1 Problembeskrivelse

Målet med denne oppgaven er å sette seg inn i eksisterende forskning og arbeider gjort rundt adaptiv crawling og dokumentere, eventuelt implementere og teste disse i forhold til FAST-Crawleren. Et overordnet mål er derfor å praktisk teste eksisterende adaptiv teori, og bidra med resultater og erfaringer som kan dras nytte av i forhold til fremtidig arbeid og forskning innenfor feltet.

1.2 Fremgangsmetode

Teorien vil bli studert gjennom eksisterende utgivelser og artikler. På grunnlag av denne teorien vil det bli implementert en intuitivt nyttig metode i sammenheng med FAST-Crawleren. Det vil bli utført et eksperiment med den adaptive FAST-Crawleren hvis resultater vil bli vurdert og kommentert.

1.3 Begrensninger

Eksperimentet vil bli utført som et begrenset crawl som ikke kvalifiserer som et globalt crawl. Et globalt crawl stiller store krav til både tid og maskinressurser utover det som er tilgjengelig for denne oppgaven. Men da vi kun er interessert i effekten av en adaptiv crawlertilnærming, vil det være tilstrekkelig å forholde seg til en begrenset samling sider innenfor en bestemt tidsperiode for hvilke man kjenner til endringsmønsteret

grovt sett. Man vil kunne se effekten av implementasjonen uansett om eksperimentet er begrenset i tid og omfang.

1.4 Overblikk over oppgaven

Kapittel 2 omfatter en innledende beskrivelse av "World Wide Web"og fundamentale konsepter i forbindelse med crawling av web.

Kapittel 3 beskriver praktiske problemer og grunnleggende utfordringer knyttet til crawling. Innholdet baserer seg på tidligere undersøkelser og erfaringer.

Kapittel 4 presenterer eksisterende teori i forhold til adaptiv crawling. Metodene som beskrives kan brukes til å løse flere av de nevnte utfordringene fra kapittel 3.

Kapittel 5 dokumenterer den utførte implementasjonen av en adaptiv metode presentert i kapittel 4. Metoden blir implementert i kombinasjon med FAST-Crawleren. I tillegg dokumenteres et eksperiment for å vurdere resultatene opp mot teorien fra kapittel 4.

Kapittel 6 tar opp generelle spørsmål rundt crawlerarkitektur med hovedvekt på et modulært og pluggbart design. Arkitekturen tar sikte på å forenkle implementasjoner av løsninger på problemstillinger fra kapittel 3, og lette implementasjoner av adaptive løsninger.

Kapittel 7 oppsummerer oppgaven og beskriver resultatene som kan være utgangspunkt for videre arbeid innenfor feltet.

Arbeidet har vært komplisert med tanke på litteratur, stor kodebase, distribuert arkitektur og asynkron eksekvering. Dette er årsaker til at den praktiske delen med implementasjon ikke ble så stor som jeg i utgangspunktet hadde ønsket. Med det er uansett et godt grunnlag for videre arbeid innenfor feltet.

1.5 Takk til

Jeg ønsker med det første å takke veileder Knut Omang for kontinuerlig støtte, verdifulle innspill og ikke minst tålmodighet gjennom hele

perioden. Jeg setter stor pris på fritid og ferie du har brukt til å bistå. En viktig forutsetning for mye av arbeidet i oppgaven er arbeidsforholdet til Fast Search and Transfer og mulighetene det har gitt.

I tillegg vil jeg utrykke stor takknemlighet overfor min nåværende arbeidsgiver FAST Search and Transfer, som har gjort det mulig å gjennomføre denne utredningen ved å tilby fleksibiliteten til å følge eget tidskjema. Diverse maskinressurser til disposisjon og tilgang til applikasjoner, har vært en viktig forutsetning for gjennomføringen. Takk til kollegaer for gode diskusjoner, innspill samt forståelse for mine prioriteringer i forhold til jobb og oppgave.

Takk til Universitetet i Oslo, Instititutt for Informatikk, administrasjonen for rask oppfølging med hjelp til enhver tid.

Til slutt ønsker jeg å nevne familie og venner for forståelse og motivering underveis.

Kapittel 2

World Wide Web

I dette kapittelet gis det en kort omtale av World Wide Web og problemer knyttet til størrelsen og funksjonen. Samtidig gis det et kort innblikk i hvordan dagens søkemotorer prøver å løse dette for å gi en mest mulig riktig presentasjon.

2.1 Bakgrunn

Det som i utgangspunktet var ment som et internt prosjekt ved europeisk organisasjon for atomforskning, har idag utviklet seg til å bli et av de største informasjons- og kommunikasjons- medier i verden [4]. The World Wide Web eller internett som det også blir kalt på norsk, bygger suksessen på blant annet den enorme utbredelsen og størrelsen [8]. I tillegg til dette er også fraværet av en sentralisert innholdskontrollerende enhet en viktig faktor for dets utbredelse. Samtidig som disse er noen av årsakene til eksplosjonen av bruk av web, utgjør de også opphavet til mange av svakhetene. De gjør blant annet den viktigste funksjonaliteten, nemlig det å finne og distribuere informasjon, til en avansert problemstilling. Søkemotorene som er tilgjengelig på web idag har alle som hovedmål å være den beste informasjonsformidleren for brukerene. Det er idag stor konkurranse om å være den mest effektive og oppdaterte søkemotoren. Det betyr å være først oppdatert på eventuelle endringer som er gjort på web og gi brukerene de mest oppdaterte, og relevante søkeresultatene.

2.2 Utviklingen

World Wide Web har siden nittitallet gjennomgått en utrolig ekspansjon [14]. De første studiene fra 1993 forteller at antall webservere ble nesten femdoblet fra 130 til 625 i løpet av en seks måneders periode. Estimatorer for å beregne den totale størrelsen av weben baserer seg på vidt forskjellige metoder som igjen gir vidt forskjellige resultater. Men ifølge [14] tilsier troverdige kilder at weben tilsammen utgjorde mer enn fire milliarder sider. Ved tidspunkt for skriving av denne oppgaven, er den siste estimasjonen på størrelsen av WWW gjennomført i 2005. Ifølge denne [5] som er gjennomført omtrent ett år etter [14] utgjør den totale indekserbare weben omtrent 11.5 billioner sider. Da er det verdt å merke seg at dette kun er snakk om indekserbar web basert på tall fra utbredte søkemotorer på web.

2.3 Uforutsigbarhet

Innholdet på web er knyttet sammen ved hjelp av *Uniform Resource Locators* (URL) eller linker på norsk. Disse linkene er pekere til addresser som unikt beskriver lokasjonen til et element. Det eksisterer etterhvert et omfattende nettverk av slike linker som gjør det mulig for en bruker å manøvrere mellom sider som linker til hverandre. Det finnes intet regelverk eller krav til hva som kan linke til hva. Det er ikke bare størrelsen, den stadige ekspansjonen og den komplekse linkingen som gjør det problematisk å manøvrere på web. Fraværet av standarder og krav til å følge standardene som til gjengjeld eksisterer, gir opphav til utallige utfordringer. Verdt å nevne er forskjellige nettlesere og deres ulike tolkninger og implementasjoner av standardene. Utviklere bruker oftest nettlesere som referanser under utvikling av nye elementer for web. Men nettlesernes tolkning av standardene er ofte av varierende kvalitet, som resulterer i at forskjellige elementer gir forskjellige resultater avhengig av nettleser. I tillegg til nettleserene er det viktig at informasjonskildene gir konsistente svar. Feilkonfigurasjoner på webservere gjør dem ofte uforutsigbare med tanke på kvalitet på tjeneste og resultat.

2.4 Søkemotorene

Det finnes utallige søkemotorer på web. Noen er fritt tilgjengelige mens andre er kommersielle og kan også spesialtilpasses etter behov. En

viktig felles egenskap med søkemotorene på nettet er at de til tross for komplikasjonene på web, prøver å forbedre tilgjengeligheten av informasjonen. For en gjennomsnittlig bruker er det ikke trivielt å finne frem til aktuell informasjon tatt i betraktning den stadige ekspansjonen i et mangfold av standarder og formater. Det er her intelligente søkemotorer prøver å lette brukerenes hverdag ved å finne frem til den riktige informasjonen det ellers er vanskelig å finne, samt spesialbehandle forskjellige tjenerinnstallasjoner for en mest mulig fullstendig presentasjon. Dette stiller høye krav til innsamlingsrutinene til søkemotorene. Eller rettere sagt høye krav til søkemotorenes crawlere.

2.5 Crawlere

Viktige egenskaper når brukere velger hvilken søkemotor som skal benyttes, er omfanget av søkbart innhold hos de forskjellige leverandørene og hvor de føler de får best resultat. Begge disse egenskapene avhenger av hverandre og er tett knyttet sammen. Det er høyere sannsynlighet for at en søketjeneste har den riktige og mest ønskelige informasjonen desto flere dokumenter den omfatter. En index som avspeiler et mest mulig riktig bilde av den virkelige weben vil kunne gi det bredeste resultatet. Det er derfor viktig å ha en mest mulig effektiv og omfattende rutine for innsamling av data. Applikasjoner som traverserer weben og samler inn innhold blir med en fellesbetegnelse kalt crawlere eller spidere. Det finnes mange forskjellige typer og metoder for en crawler å fungere på. Fellesnevneren for enhver crawler er at den robust skal kunne behandle serverkonfigurasjoner og forskjellige formater for å kunne laste ned det aktuelle innholdet. Veldig forenklet kan crawleprosessen beskrives slik:

1. Crawleren starter med en samling adresser til ulike sider som utgangspunkt.
2. Linkene trekkes ut fra disse sidene og legges til listen med addresser som skal crawles.
3. 1. og 2. blir så gjentatt for enhver side som blir addressert/linket til.

Crawlere kan lett konfigureres til å besøke de samme sidene hyppig nok i forhold til endringsraten. Men en naiv løsning vil gi mye unødig trafikk til alle disse sidene og dermed unødig belastning av nettverk og servere. Dessuten er det typisk slik at noen sider endrer seg mye oftere enn andre. I tillegg kommer problemet med dynamiske sider [1]

[1] sider som genereres som et resultat av forespørsel

eller feilkonfigurerte webservere hvor det kan se ut som om sidene endres kontinuerlig. Man skiller hovedsakelig mellom to hovedtyper crawlere: periodisk og kontinuerlig [10]. Selve prosessen å crawle er i seg selv temmelig lik for begge typene. Kjernen i problemstillingen er å definere hvilke addresser som er nødvendig å crawle samt holde denne samlingen mest mulig oppdatert. Den «perfekte» crawleren sørger for at alle interessante websider blir crawlet samt at alle disse er oppdatert til enhver tid. Denne finnes ikke og hvorvidt det er mulig å lage en slik er også usikkert, men enhver forbedring av eksisterende crawlere gjør avstanden mindre.

2.5.1 Periodisk vs. Kontinuerlig crawler

Fra 17 februar til 24 juni 1999 ble det ved Stanford Universitet gjennomført en global crawling på web for å kartlegge dynamikken på web [10]. Dette var en del av et større prosjekt for å vurdere fordeler og ulemper ved ulike crawler arkitekturer. Ved hjelp av forsøket ble det trukket en parallell til Poisson fordelingen som igjen ble brukt til å bevise ulemper eller fordeler ved diverse crawler arkitekturer. Den primære oppgaven til en crawler er å sørge for at de lokale sidene er mest mulig oppdatert. Man kan innføre et mål *ferskhet* på hvor oppdatert en lokal samling med websider er. Typisk kan man si at hvis ferskheten er 1 er alle websidene i den lokale crawler kolleksjonen oppdatert, mens hvis ferskheten er 0,5 er kun halvparten av websidene i den lokale kolleksjonen oppdatert. Begrepet vil bli benyttet videre i teksten.

Crawlere kan jobbe på to forskjellige måter: periodisk (*periodic*) eller kontinuerlig (*incremental*). En periodisk crawler kjører i perioder og ligger i dvale resten av tiden. Når den først kjører oppdaterer den alle elementene i den lokale samlingen. I perioden når crawleren kjører stiger ferskheten for kolleksjonen mens i periodene hvor crawleren ligger i dvale synker ferskheten. Til tross for at det umiddelbart skulle virke slik så blir aldri ferskheten lik 1 for en periodisk crawler. Selv ved slutten av perioden når crawleren er aktiv, er ferskheten mindre enn 1. Dette fordi det hele tiden er noen sider som blir forandret og mange sider forandrer seg i løpet av den perioden crawleren kjører. Derfor vil det ved slutten den aktive perioden allerede være sider som er forandret og ferskheten vil da være mindre enn 1. En kontinuerlig crawler kjører hele tiden. Den ligger altså aldri i dvale. På denne måten får man ikke de omfattende variasjonene i ferskheten som for en periodisk crawler. Raten holder seg stabil over tid. Fordeler ved en slik aktivitet er at crawleren ikke trenger å besøke sider så raskt som en

periodisk crawler.

Hvis begge typer crawlere skal besøke et visst antall sider i løpet av en periode (en periode = en aktiv + en passiv, for en periodisk crawler), må den periodiske crawleren raskt gjøre seg ferdig med hele antallet for så å ligge i dvale resten av perioden, mens den kontinuerlige crawleren kan jobbe med en lavere hastighet gjennom hele perioden. Den høye hastigheten som er en forutsetning for en periodisk crawler kan i noen tilfeller føre til overbelastning av lokal maskin samt nettverk, som videre kan bli årsaken til alvorlige stabilitetsproblemer.

I tillegg, liker ikke webansvarlige aggressive crawlere og kan i så fall velge å blokkere forespørsler fra slike. Det også bevist ved hjelp av Poisson fordelingen, forutsatt at den gjennomsnittlige besøkshastigheten er den samme, at både en periodisk og kontinuerlig crawler har samme gjennomsnittlig ferskhet. Men den lave stabile ressursbelastningen hos en kontinuerlig crawler i motsetning til høy periodisk ressursbelastning hos en periodisk crawler, gjør arkitekturen til en kontinuerlig crawler mer praktisk.

2.5.2 «Shadowing» vs. «In-Place»

Som omtalt i [10] finnes også variasjoner for hvordan crawleren oppdaterer sin lokal samling av websider. Man skiller hovedsakelig mellom to metoder: *shadowing* og *in-place*. For en in-place oppdatering eksisterer det kun en kopi av kolleksjonen. Etterhvert som sidene blir crawlet, blir disse fortløpende gjort tilgjengelig for brukerene. Ved bruk av shadowing eksisterer det derimot to kolleksjoner. En kolleksjon som er tilgjengelig for brukersøk, og en kolleksjon som crawleren jobber med. Etter hvert crawl byttes så disse kolleksjonene slik at den oppdaterte kolleksjonen blir gjort tilgjengelig for brukersøk, mens den ikkeoppdaterte kolleksjonen blir crawlet. Fordelen ved shadowing er separeringen av crawleprosessen og søkingen. Den søkbare kolleksjonen som er tilgjengelig for brukerene er fullstendig skjermet for crawleprosessen. Dette forbedrer tilgjengeligheten for brukerene. I tilfeller hvor nye kolleksjoner er nødt til å bli preprosessert er fortsatt den søkbare kolleksjonen tigjengelig for brukere og totalt uberørt av preprosesseringen. Nok et viktig poeng er at shadowing rent praktisk sett er enklere å implementere enn en in-place taktikk.

Det er uansett en stor fordel med in-place oppdatering, nemlig ferskheten. En viktig faktor som gir stort utslag på ferskheten er når in-place gjør oppdaterte sider umiddelbart tilgjengelig for søk. Dette gir en direkte påvirkning på ferskhetsraten umiddelbart etter et element er

oppdatert. Ved shadowing blir ikke de oppdaterte sidene gjort tilgjengelig før kolleksjonene byttes. Ferskheten vil da ikke bli påvirket av crawlingen fordi den tilgjengelige kolleksjonen er totalt skjermet. Ved bruk av shadowing vil derfor ferskhetsraten fortsette å synke uavhengig av crawlingen.

2.5.3 Vurdering

Men hvordan påvirker de forskjellige crawler taktikkene ulike typer crawlere? I forsøket ved Stanford [10] ble ferskheten målt for ulike kombinasjoner av en periodisk vs. kontinuerlig crawler, og in-place vs. shadowing taktikkene. For en kontinerlig crawler ble det målt stor nedgang i ferskheten ved overgang fra in-place til shadowing. Dette er forståelig når crawleren går uavbrutt og alle sidene som etterhvert blir oppdatert fortløpende blir lagt ut tilgjengelig for søk, som videre gir direkte positiv effekt på ferskhetsraten. Raten for en periodisk crawler er ikke mye påvirket av overgangen fra in-place til shadowing. Raten ble noe mindre, men ikke så markant som for en kontinuerlig crawler. Derfor kan det anbefales å implementere en shadowing taktikk for en periodisk crawler. Den er lettere å implementere samtidig som forskjellen i ferskheten ikke er så stor. Perioden hvor crawleren er aktiv er årsaken til at ferskheten blir forskjellig. Dette fordi ved en in-place ville alle de oppdaterte sidene bli lagt ut fortløpende, mens ved shadowing forbedres ikke ferskheten av de nylig oppdaterte sidene før kolleksjonene byttes. Perioden hvor crawleren var aktiv var forholdsvis sjelden som videre ikke gav så markante påvirkninger på raten. Hadde crawleren vært brukt på en mer dynamisk del av weben hvor den på grunn av dette ble satt opp til å kjøre oftere, ville den negative påvirkningen ved en overgang fra shadowing til in-place vært større. Det ville derfor i et slik tilfelle vært lurere å implementere periodisk crawling med in-place taktikk.

2.6 Oppsummering

En eksponensiell vekst, kompleks struktur, fraværet av standarder samt slappe tolkninger av eksisterende standarder er alle viktige årsaker til en ikke-triviell lokalisering av informasjon på internett. En oppgave som dagens søkemotorer konkurrerer om å løse best mulig. Hva best mulig innebærer er selvsagt brukerstyrt, men intuitivt kan ikke best mulig være langt fra den faktiske tilstanden på web. En søkemotor med en indeks som er mest identisk med originalversjonene på internett vil ha best

mulig utgangspunkt for å gi den riktigste presentasjonen. Rutinene som er knyttet til innsamling og replikering av søkemotorenes lokale indekser, er alle samlet i moduler kalt crawlere. Crawlere kan konfigureres på vidt forskjellige måter og disse gir igjen forskjellig resultater avhengig av forskjellige kontekster. Hvilken konfigurasjon som skal velges er en vurdering som må tas utifra de reelle behovene og i hvilken kontekst crawleren skal operere.

Kapittel 3

Grunnleggende Crawling

3.1 Introduksjon

Besøksmønsteret for forskjellige sider på weben er veldig varierende [10],[7]. Flesteparten av brukerene besøker en begrenset del av weben som gjennomgående er sider av høy kvalitet. Men til tross for dette representerer ikke webstedene i denne begrensede delen weben generelt. Fordi både kostnads- og kunnskaps- terskelen for å publisere informasjon på weben er svært lave, finnes det enormt mange websteder av tvilsom kvalitet. Webcrawlere må i tillegg til å takle høykvalitetselementer også stabilt kunne behandle lavkvalitetselementer. Her presenteres en gjennomgang av hvilke praktiske aspekter som er viktig for en crawler å ta hensyn til, samt aktuelle problemstillinger forbundet med global crawling av web.

3.2 Generell nettverksproblematikk

3.2.1 Begrensede nettverksressurser

Gjennom tidligere undersøkelser [7] er det estimert at et globalt crawl av hele «World Wide Web» tilsvarer et forbruk av ressurser verdt 1.5 millioner dollar. Disse kostnadene er kun knyttet til bruk av nettverksressurser for å utføre crawlet. Det er derfor viktig å benytte disse ressursene på en mest mulig effektiv måte. En stor utfordring knyttet til effektiv bruk av nettverksressurser, er en mest mulig jevn utnyttelse av ressursene. Det er mer ønskelig med et moderat men jevnt nivå på nedlastingshastigheten, enn i noen situasjoner ekstreme hastigheter og andre situasjoner lave hastigheter. Dette for å forsikre stabilitet og unngå å bli blokkert fra

websteder på grunn av til tider aggressiv nedlasting.

3.2.2 Variabel tjenestekvalitet

Men i praksis er det ikke triviellt å sørge for at crawleren hele tiden skal ha en stabil bruk av nettverksressursene. Forskjellige websteder har forskjellig tjenestekvalitet hvor noen kan ha en minimal responstid, mens andre kan ha lengre responstid. Disse tidene kan også variere for et og samme websted. Det kan for eksempel være websteder som har vært nede i en lengre periode som plutselig dukker opp igjen, eller websteder som forsvinner. Man kan derfor ikke forutsi hva slags tjenestekvalitet man vil få fra forskjellige websteder. Dette betyr videre at det ikke er en enkelt oppgave for en crawler å operere med et jevnt bruk av nettverksressursene.

3.2.3 Crawlertaktikk

I utgangspunktet var ikke webcrawlere spesielt godt likt av webansvarlige på grunn av begrenset båndbredde. De tar opp ressurser som man heller ville skulle være direkte tilgjengelig for brukerene, og til tross for den sterke teknologiske utviklingen på nettverksfronten, er problemene knyttet til nettvergsbegrensinger fortsatt tilstede idag. Gjentatte forespørsler over kort tid på grunn av den variable hastigheten på forespørslene, kan ofte resultere i advarsler og alarmer på webserverene. Dette kan videre resultere i klager i form av epost og telefon eller hvis denne informasjonen ikke er tilgjengelig, total utestengelse fra tjenesten. Det er derfor viktig for crawlere å identifisere seg selv når de er aktive. Men viktigere enn dette er å sørge for at forspørslene til en og samme webserver ikke blir for hyppige. I mange tilfeller viser det seg at flere forskjellige webadresser ofte peker til en og samme ipadresse. Det kan for eksempel være flere virtuelle servere som er satt opp på en og samme maskin. I slike situasjoner er det viktig at crawleren kan identifisere serverene på grunnlag av ipadresse og ikke bare URLen.

3.2.4 Inkonsistente nettverkskonfigurasjoner

Variasjonen i tjenestekvaliteten er ikke alltid på grunn av de fysiske nettverksbegrensningene. Ofte kan dette skyldes feilkonfigurasjoner i webserver, brannmurer, switcher og andre nettverkskomponenter. I flere situasjoner er det avdekket at feilkonfigurasjoner i brannmurer kan gi

uforutsette resultater. Som for eksempel kan det være åpent i brannmuren på port 80 (standard port for http protokollen), men konfigurasjonen slik at alle pakker som kommer inn på port 80 blir droppet. Dette vil praktisk bety at «connect()» på port 80 vil fungere, «write()» på koblingen vil også være vellykket (forespørselen blir skrevet), men det vil aldri bli sendt noen respons tilbake fordi alt som ble skrevet i forespørselen ble droppet. En crawlertråd som ikke tar hensyn til lignende avbrytelser vil mest sannsynlig i slike tilfeller resultere i ustabiliteter. En robust crawler må til enhver tid operere med såkalte «timeouts» hver gang den venter på data for i alle situasjoner å kunne avbryte en nedlasting.

3.3 Problemer knyttet til DNS

3.3.1 DNS og nedetid

Tjenesten som blir benyttet for å oversette en url til en ip kalles DNS eller «Domain Name System». Hver gang det gjøres en forespørsel til en url blir først den lokale dnsserveren spurt om en ipadresse for den respektive url'en. Det blir deretter returnert en ipadresse til klienten som kobler opp mot den returnerte adressen. Som for alle andre typer tjenester, har heller ikke DNS en oppetid på 100%. Det kan derfor til tider hende at tjenesten ikke er tilgjengelig. For en webklient betyr kanskje ikke dette så mye på grunn av standard «caching» som oftest er en uke. I opptil en uke vil da den «cachede» adressen bli benyttet for så når DNS tjenesten er oppe igjen vil alt fungere som vanlig. Men for en crawler vil ethvert oppslag på urler feile, og et helt crawl vil i tilfellet utilgjengelig DNS være bortkastet. For å unngå lignende situasjoner er det viktig at DNS tjenesten blir testet under mye trafikk for å forsikre stabilitet. I tillegg burde det være mulig for en crawler å detektere feil i DNS tjenesten hvis for eksempel en gitt andel av DNS oppslagene feiler. Ved en detektert feil burde crawleren automatisk gi beskjed og repetere crawlet etter en gitt tidsperiode for at crawlet ikke skal bli fullstendig mislykket.

3.3.2 Feil i DNS

DNS tabellen er ikke feilfri og det forekommer uriktige poster. Vi kan se for oss et scenario med to forskjellige websteder, A og B. Ved et første crawl resulterer feillagring i DNS i at oppslag på url'en til A returnerer ipadressen til B. Det betyr videre at alt innhold som eksisterer på websted B blir lastet ned og indeksert, referrert til av url'en til A. Når så B

skal crawles blir det initiert oppslag på B i DNS som returnerer samme ipadressen som for A. Innholdet blir detektert til å være identisk, som det i prinsippet også er fordi det er akkurat det samme innholdet, og crawleren unngår å laste dette ned. I denne situasjonen er altså både webstedet A og B indeksert slik at innholdet av B er referrert til av url'ene til både A og B. I utgangspunktet er dette forsåvidt greit da søketreff (for innholdet i B) vil returnere url'ene til både A og B, og på grunn av feilkonfigurert DNS vil oppslag i DNS (ved bruk av søkeresultatet, nemlig url'ene A og B) returnere ipadresen til B for begge url'ene. Dette er forsåvidt helt rett da innholdet som det blir søkt i også stammer fra B for begge url'ene. Men med en gang feilposten i DNS blir korrigert blir resultatet inkonsistent. I situasjonen hvor DNS posten for A er blitt korrigert, men crawleren ikke har crawlet og indeksert nytt innhold enda (gjort nytt oppslag på url'en til A) blir det problemer. Søk i innhold fra B vil fortsatt returnere url'ene til både A og B. Men når så klienten klikker på url'en til A i resultatsettet, vil lokaloppslaget på url'en til A returnere den korrigerte ipadressen til A. Dermed vil altså søketreff i innhold fra B kunne referere til resultatsider fra A, altså et inkonsistent resultat.

3.4 Variasjon i HTTP-implementasjoner

3.4.1 HTTP-headere

En generert HTTP forespørsel inneholder flere felt. Den inneholder blant annet informasjon for å identifisere seg selv, det vil si klienten som spør, hvilket type innhold klienten er ute etter, hva klienten er ute etter med mer. En forespørsel fra nettleseren firefox på en linux platform kan for eksempel se ut som Listing 3.1.

Listing 3.1: Forespørrende header

```
1 GET / HTTP/1.1
2 Host: www.fast.no
3 Connection: close
4 Accept-Encoding: gzip
5 Accept: application/xhtml+xml,text/html,text/plain
6 Accept-Language: en-us,en
7 Accept-Charset: ISO-8859-1,utf-8
8 User-Agent: Mozilla/5.0
```

Linje 1 beskriver hvilken protokoll som skal brukes (HTTP), navnet på ressursen som blir forespurt (/) og hvilken type forespørsel som sendes (GET). Videre forteller linje 2 hvilken tjener som skal kontaktes. Dette er

typisk adressen som blir sendt for oppslag til DNS som dernest returnerer korrekt ipadresse til riktig tjener.

3.4.2 Tjenersidebehandling av headeren

De viktigste feltene i HTTP forespørselen er «Accept» og «User-Agent». Disse er typiske felt som blir hentet ut av tjeneren og testet på før innhold returneres. Problemet her er fraværet av en standardisert resultatmatrise som definerer innholdstyper for forskjellige «Accept» typer og «User-Agent» klientdefinisjoner. Forskjellige nettleser har begrenset hvilket type innhold de er istand til å vise. Begrensningene for forskjellige nettlesere er som regel gjenspeilet i forespørselen som blir sendt. Problemet ligger så på tjenersiden. Det forekommer her ingen konsistent metode for å sjekke forespørselen og returnere innhold. To forskjellige tjenere kan returnere forskjellig type innhold selv om forespørselen indikerer identiske typer. Mens andre tjenere kan velge å se totalt bort i fra typedefinisjonene i forespørselen. I disse tilfellene vil klienten ofte bruke tid og nettressurser på å laste ned innhold som den ikke er istand til å vise.

3.4.3 Allsidig dekning

For å forsikre en mest mulig allsidig dekning av innholdet på weben, er det viktig å begrense totalmengden av innhold som blir lastet ned fra hver enkel tjener. Dette kan enkelt gjøres ved å begrense totalt antall elementer som skal lastes ned fra en og samme tjener, samtidig som å definere et maksimum for hver enkel nedlasting. Det anbefales [7] et maksimum på 300 - 400 Kb for individuelle nedlastinger. Denne mengden data skal inneholde mange nok ord for å gi en god resultatindeks. Maksimumsverdien må kunne overføres til tjeneren slik at den vet hvor mye data som er ønsket. Dette kan enkelt gjøres ved å legge til en «Range» parameter i forespørselen. Headeren i forrige eksempel kan for eksempel se ut som Listing 3.2.

Listing 3.2: Forespørrende header med range parameter

```
1 GET / HTTP/1.1, Range: 0-400000
2 Host: www.fast.no
3 Connection: close
4 Accept-Encoding: gzip
5 Accept: application/xhtml+xml,text/html,text/plain
6 Accept-Language: en-us,en
7 Accept-Charset: ISO-8859-1,utf-8
8 User-Agent: Mozilla/5.0
```

I linje 1 ser man at det er lagt til en «Range» parameter for å indikere gyldig filstørrelsesintervall i antall byte. Men også i dette tilfellet blir parameteren tolket på forskjellige måter av forskjellige tjenere. Ønsket funksjonalitet er å kunne laste ned 400Kb av filer som er større enn 400Kb, og laste ned hele filen for de som er mindre enn 400Kb. Men i mange tilfeller oppnås ikke dette. Noen tjenere velger å returnere en såkalt 416 feilkode (feilkode for «Range» feil) når filstørrelsen ikke er innenfor det gitte intervallet. Andre tjenere kan velge å totalt ignorere «Range» verdien. Ifølge HTTP standarden [1] er dette ikke korrekt.

3.4.4 Redefinisjon av feilkoder

Når et websted vokser og blir oppdatert av flere forskjellige utviklere øker risikoen for døde linker på webstedet. Mange utviklere liker ikke at brukere skal bli presentert med nettleserenes interne sider for de ulike feilkodene som genereres og returneres. Det er ofte webansvarlige velger å omdirigere brukere til egendefinerte sider når døde linker forekommer, slik at klientene ikke viser sine interne feilkodesider. Men dette vil i noen tilfeller, hvis feilkoden ikke ivaretas, resultere i at de egendefinerte feilkodesidene som returneres blir sendt som ordinære sider. De egendefinerte feilkodesidene blir ofte sendt tilbake som vanlige websider [1]. Det vil i slike tilfeller ikke være enkelt for hverken nettlesere eller crawlere å identifisere disse sidene korrekt, nemlig som døde linker. I crawlertilfellet vil disse sidene bli indeksert som vanlig innhold og faktisk være søkbare fra indeksen.

3.4.5 Feildata i responsheaderen

Responsheaderen inneholder informasjon om innholdet som blir sendt tilbake til tjeneren. En typisk responsheader sendt tilbake fra etterspørselen tidligere kan se ut som Listing 3.3.

Listing 3.3: Responderende header

```
1 HTTP Status Code: HTTP/1.1 200 OK
2 Connection: close
3 Content-Length: 26423
4 Date: Thu, 13 Apr 2006 14:47:47 GMT
5 Content-Type: text/html; charset=utf-8
6 Server: Microsoft-IIS/6.0
7 X-Powered-By: ASP.NET
```

[1] med HTTP statuskoden 200

```
X-AspNet-Version: 1.1.4322
Set-Cookie: ASP.NET_SessionId=i011mv55ktc3m5fx1rqqjrim; path=/
Cache-Control: private
```

Ved å tolke informasjonen sendt tilbake fra tjeneren, kan klienten bestemme hvorvidt den vil hente ned det forespurte innholdet eller ikke. Feilkonfigurerte tjenere kan resultere i feilinformasjon i responsheaderen som blir sendt tilbake til klienten. Ofte kan dette være banale årsaker som feil tidsinnstilling på tjeneren. Som i linje 4 ser vi hvilken tidsinnsilling tjeneren har. I tillegg til denne er noen tjenere konfigurerte til å sende tilbake en sist modifisert tid eller «last modified» tid. Denne kan strategisk benyttes, som omtalt senere, til å estimere sannsynlighet for at et element er forandret eller ikke.

I forsøk på å utvide prøveperioden for tidsbegrensede demonstrasjonsversjoner av programvare, er det ofte at tidsinnstillingen blir satt til et tidligere tidspunkt enn hva som er reelt. Slike konfigurasjoner vil returnere en gal «last modified» tid, og følgelig vil enhver sannsynlighetsestimering av eventuell forandring slå feil.

Det oppstår hele tiden nye innholdsformater som distribueres på nett. Et format som er verdt å nevne er de nye mobiltelefonformatene. Dette er begrensede html spesifikasjoner som skal kunne vise meget forenklede sider på grunn av båndbreddebegrensninger og små visningsflater [2]. De to største mobiltelefonformatene er henholdsvis wml og xhtml. Begge disse har hatt en sterk vekst siden WAP protokollen ble lansert for noen år tilbake, mens den nylige lanseringen av nett som 3G og EDGE også har forsterket denne. Det som ofte viser seg å være et problem ikke bare med wml og xhtml men også andre nyere formater, er at de fortsatt publiseres som html eller andre eldre mimetyper av webtjenerene. I responsheaderen kan forskjellige tjenere beskrive innholdet som alt fra html til vanlig text, noe som gjør det vanskelig for crawleren å idenfifisere type innhold før det er lastet ned. Følgelig må det være mulig å gjøre formatsvalideringen på alternative metoder istedetfor å stole blindt på responsheadere, som ikke blir oppdatert riktig i tråd med nye publiserte innholdsformater.

3.4.6 Robots.txt spesifikasjonen

Standarden for ekskludering av automatiske roboter på nett «SRE» ble først foreslått i 1994, som en metode for å hindre automatiske roboter fra bestemte uønskede områder på web [13]. Blant disse kan være:

[2] mobiltelefonskjermer, pdaskjermer, hybridtelefonskjermer med mer

1. Uendelige / sirkulære lenkede sider hvor crawleren aldri blir ferdig
2. Kostbare sider med tanke på ressurser, slik som for eksempel dynamiske sider
3. Innhold som vil trekke til seg ukontrollerbar trafikk slik som vokseninnhold
4. Webområder som kommersielt kan virke hemmende på virksomhetens ansikt utad, som for eksempel feilrapporteringssider
5. Elementer som ikke er globalt nødvendig å indeksere, slik som lokalinformasjon.

Robots.txt er standarden som webansvarlige kan benytte seg av for å kontrollere crawleratferd på de ulikes websteder. Her spesifiseres ganske enkelt hvilke deler av webtrestrukturen som skal kunne crawles. Men det er ikke alltid disse er riktig konfigurert. Et eksempel [3] på dette er mobilsidene til Verdens Gang: http://wap.vg.no/. For de fleste andre nyhetssider, og også VGs ordinære internettsider, ønsker opplagt å være tilgjengelig gjennom søk da søkemotorene står for mye av informasjonspresentasjonen for brukere av WWW. Man skulle tro at dette også er tilfelle VGs mobile nyhetssider. Men http://wap.vg.no/robots.txt forhindrer effektivt dette som vist i Listing 3.4.

Listing 3.4: http://wap.vg.no/robots.txt

```
1 User-agent:       *
2 Disallow:         /
```

Robots.txt spesifikasjonen på VGs mobile nyhetstjener sperrer samtlige crawlere fra å crawle alt som eksisterer. Slik denne ser ut idag er det kun crawlere som velger å overse `robots.txt` spesifikasjoner som vil kunne crawle disse sidene.

Feilkonfigurerte robots.txt spesifikasjoner gjør det altså vanskelig for crawlere å sørge for en allsidig presentasjon av web. I mange tilfeller kan også crawleradministratorer føle seg tvunget til å overse disse spesifikasjonene, med medhold fra webansvarlige for nettsteder hvor det enkelt ikke lar seg gjøre å rette opp i robots.txt-definisjonene, for å kunne indeksere innholdet. Resultatet av et slikt valg kan føre til at andre websteder som implementerer spesifikasjonen korrekt, kan velge å utestenge crawleren fordi den ikke respekterer deres crawler.txt definisjon.

[3] av dato 14. april 2006

3.5 HTML koding

HTML er basert på SGML- eller XML-aktig syntaks som enkelt skal kunne oversettes til en trestruktur for behandling av dataen. Men fordi det ikke eksisterer noen kontroll av HTML koden, skjer det ofte at syntaksen ikke overholdes. Forskjellige kodeformater som for eksempel html og xhtml blir brukt om hverandre [4], bruk av anførselstegn er ikke entydig, ubalanserte tagger [5], blandet bruk av store og små bokstaver i koden, uavsluttede strenger som i verste fall kan være årsak til ustabiliteter i applikasjonen med mer. Det er viktig for crawleren å kunne separere innhold fra design. Vanligvis kan mye av htmlkodingen relateres til design og overses. I tilfeller hvor CSS (*Cascading Style Sheets*) er brukt er mye av jobben med å separere innhold fra design allerede gjort. Sider som bruker CSS er derfor ofte enklere å behandle.

3.6 Serverapplikasjoner

3.6.1 Sesjonsidentifikatorer

For å skille mellom ulike brukere og andre kontekster er bruken av sesjonsidentifikatorer utbredt på nett. Dette er identifikatorer som blir lagt til som parametre i url'en etter ressursen [6]. Sidene som presenteres er ofte like med unntak av noe spesialinformasjon som avhenger av sesjonsidentifikatorene. Her finnes det en stor kilde for duplikater som det kanskje er unødvendig å crawle 100%. Problemet er for crawleren å ta hensyn til identifikatorene og intelligent kunne avdekke hvilke sider som er duplikater selvom noe av innholdet er forskjellig.

3.6.2 Interne stier

Linking internt på et websted forekommer på hovedsakelig to forskjellige måter. Ved hjelp av absolutt sti og relativ sti. Absolutte stier er satt med utgangspunkt i mappen som er forhåndsdefinert som rot. Relativ sti er filstier som er satt utifra mappen hvor aktuell ressurs befinner seg. En blanding av disse to formene kan ofte ved dynamisk autogenererte ressurser resultere i feil stier. Det kan være at en autogenerert url ved en feil blir laget relativ istedetfor absolutt fordi rotmappen blir glemt, eller for

[4] html: <br>, xhtml:

[5] <h1>...... </h1>

[6] http://www.fast.no/index.html?identifikator=ola

eksempel at noen deler av stien forekommer flere ganger. Disse er begge feil som ikke er uvanlig at forekommer ved dynamiske websteder.

3.6.3 Dynamiske sider

Dynamiske sider er ofte tregere enn statiske. Dette ganske enkelt fordi man er avhengig av å spørre et eller flere kilder for informasjon, eller på grunn av programmeringmessige feil. Sider som er dynamiske kan ofte ble tregere enn statiske sider med en faktor fra 10 og opptil 100 [7]. Følger av dette er at en crawler ofte må stå og vente på at slike sider først skal bli generert før den får en respons. Problemstillingen med å få crawleren til å benytte seg av nettverksressurser på en mest mulig jevn måte blir altså forsterket på grunn av dynamiske websteder.

3.7 Duplikatdeteksjon

En av de mest essensielle oppgavene en crawler har er deteksjon av duplikater. Ikke bare er det unødvendig for en crawler å laste ned identisk innhold flere ganger, men under prosessen med å determinere hvorvidt en gitt side er forandret eller ikke må den også kunne detektere duplikater. Det er ikke nødvendig å laste ned http://www.aaa.com/index.html hvis den er identisk med versjonen som ble lastet ned og indeksert ved forrige crawl.

Det er estimert [7] at mer enn 30% av innholdet på web er speilet. Selvom speil intuitivt forbindes med duplikater på forskjellige tjenere, forekommer speilet innhold oftest i forbindelse med aliaser og redundante navn. For eksempel vil http://www.fast.no/ og http://www.fastsearch.no/ peke til samme ipadresse. En naiv crawler vil skille på disse sitene selvom det er den samme siten, og behandle innholdet som disjunkte elementer. Tilsammen utgjør det en betydelig arbeidsreduksjon for crawleren hvis den intelligent kan klare å avdekke hvilke tjenere som er speilbilder av andre tjenere. Men problematikken begrenser seg ikke til duplikatdeteksjon for enkeltelementer ved forskjellige crawl samt duplikatdeteksjon av speilet innhold.

Mengden av dynamisk innhold på web har gjennomgått en utrolig vekst de siste årene. Og ofte er disse dynamisk genererte sidene kun forskjellig med tanke på reklame, design, lokalinformasjon, datoer, klokkeslett med mer. Det kan for eksempel forekomme en og samme nyhetsartikkel som er publisert på flere forskjellige nettsteder. Alle disse nettstedene vil mest sannsynlig ha individuelt design, utgivelsesdatoer

og klokkeslett, forskjellig reklame og lokalinformasjon i tillegg til selveste nyhetsartikkelen. Spørsmålet er så hvor mye forskjellig to webelementer kan være for at de fortsatt skal kunne være duplikater? Og hvilke deler av innholdet skal kunne være forskjellig for at to elementer fortsatt skal være identiske?

Dagens crawlere utfører duplikatdeteksjon ganske enkelt ved å strippe bort html logikken, og beregne en sjekksum av den resterende delen som blir karakterisert som innholdet. Disse sjekksummene blir lagret og sammenlignet for hver gang et nytt element blir lastet ned. Det er helt klart at duplikatdeteksjon er en viktig crawleraktivitet som krever mye oppmerksomhet. Både med tanke på nye algoritmer og forbedring av de eksisterende.

3.7.1 Duplikatdeteksjon ved hjelp av RSYNC

Rsync er et verktøy for å kopiere og synkronisere filer [3]. En ftp overføring vil overføre hele filer selvom kun en byte av de ulike filene er forandret. Spesielt med rsync er at den baserer seg på kun å overføre forskjellen mellom to ulike filversjoner. Fordelene her er intuitivt besparelse av nettverksressurser og tid. Videre er også fildiffene komprimert før de blir overført som igjen fører til ytterligere besparing av nettverksressursene. Rsync blir idag hovedsakelig benyttet som en applikasjon for sikkerhetskopiering.

Verktøyet fungerer ganske enkelt ved at en tjener blir satt opp. Denne konfigureres ved å spesifisere stien til filtreet som skal deles, berettigede brukere og filrettigheter. Maskiner som har nettverkstilkobling til tjeneren kan da koble seg til med klienten og laste ned hele filtreet [7]. Etterhvert som filene på tjeneren endrer seg, vil tilsvarende klientkommando som ble utført ved første nedlasting kun overføre forskjellene for de forandrede filene.

For et webscenario ville tilsvarende filoverføringsmetode som for rsync være optimal. Duplikatdeteksjonen vil bli overflødig fordi rsync tar seg av denne. I tillegg vil kun de delene av websiden som er forandret bli overført som videre gir besparelse av nettverksressursene. Men det er kun duplikatdeteksjonen forbundet med enkeltelementer fra forskjellige crawl som vil bli overflødig. Deteksjonen av speilet innhold på web må fortsatt utføres for å unngå nedlasting av identiske websteder. Men til tross for dette vil spørsmålene stilt i forrige avsnitt for duplikatdeteksjon av enkeltelementer bli overflødige.

[7] første gang må hele treet lastes ned

Problemet med å implementere en crawler som har støtte for den foreslåtte funksjonaliteten, er behovet for endring på webtjenersiden. Alle webtjenere som vil støtte crawlere som implementerer rsync funksjonalitet, må også selv implementere tjenerdelen av rsync algoritmen. Dette kunne i og for seg vært standard logikk som inngikk i samtlige webtjenerløsninger som er tilgjengelige idag. Det kunne på samme måte som for robots.txt spesifikasjonen vært en rsync.txt som inneholdt all informasjon en crawler ville trenge for å muliggjøre en synkronisering ved hjelp av en rsync kobling. Men som nevnt er terskelen for å virkeliggjøre disse endringene på samtlige webtjenere forholdsvis høy. Det krever en systematisk endring av samtlige tjenerinnstallasjoner og oppfølging rundt dette. Man kan spørre seg hvor sannsynlig det er å få til en slik overgang, når man idag har problemer med å få tjenere til å implementere robots.txt spesifikasjonen sin skikkelig.

3.8 Utilgjengelig web - The Deep Web

Ordinære crawlere lager indeksen sin ved å gå utifra et sett med forhåndsdefinerte start-URLer og traversere disse [6]. For at en side skal være crawlbar, må den på en eller annen måte være linket til gjennom de forhåndsdefinerte start-URL'ene. Det finnes websidekolleksjoner som i sin helhet er adskilte fra den globalt tilgjengelige weben. For å kunne crawle disse sidene må en vite url'en til minst en av sidene som er med i mengden. Det finnes altså ingen innadgående linker fra noen sider inn til sider i disse kolleksjonene. Dette er sider som kalles «utilgjengelige». Med det menes at de ikke er tilgjengelige fra omverdenen. For å få innblikk i disse kolleksjonene må minst en url blant elementene i kolleksjonen være kjent. Slike separerte kolleksjoner utgjør en stor utfordring for crawlere. De er rett og slett ikke crawlbare så lenge crawleren ikke kjenner til minst en url.

Tilsvarende er det for dynamiske sider som fortløpende genereres når de besøkes. Det kan være søketjenester som henter ut informasjon fra databaser og viser resultatet som websider når de aktuelle url'ene blir besøkt. Men sålenge crawleren intelligent ikke kan utføre søk slik som vanlige brukere så eksisterer heller ikke disse sidene, og informasjonen forblir bortgjemt og utilgjengelig i databasene. Disse sidene vil på lik linje med elementene i utilgjengelige webkolleksjoner være en utfordring for webcrawlere. Å implementere logikk som lar crawlere utføre søk på samme måte som reelle brukere, er metoden for å hente ut det skjulte innholdet fra databaser rundt omkring på weben.

3.9 Vurdering

Det er klart at hovedvekten av de praktiske utfordringene for en crawler er knyttet til feilimplementasjoner av tjenere og tjenerprogramvarer. Å programmere en crawler kan sammenlignes med å lage et utforskningsverktøy. Det er viktig etterhvert som det utforskes at verktøyet tilpasses og forbedres i henhold til erfaringen. Kombinasjonen av feilkonfigurasjoner av webtjenere samt distribusjonen av nye filformater på web krever en stor nok tilpasningsevne på crawlersiden slik at nye formater likevel kan crawles. Også for å kunne implementere alternative løsninger for duplikatdeteksjon er det viktig for crawleren å være tilpasningsdyktig. World Wide Web vil fortsette å utvikle seg og nye utfordringer vil bli et faktum. Det er viktig å kunne forutsi komplikasjoner tidlig og utforme en crawlerarkitektur som gjør crawleren mest mulig konfigurerbar til enhver tid, og enkel å tilpasse fremtidens behov.

Men det er også viktig å ha et fokus på begrensinger innenfor eksisterende nettverksressurser. Selvom teknologien er under stadig utbedring og tilgjengelige ressurser øker, er også størrelsen på weben i rask vekst. Spørsmålet er hvorvidt ressursveksten noen gang kommer til å bli så stor at man ikke trenger å tenke på begrensningene i forhold til størrelsen av weben. Derfor er det også viktig å tenke på alternative metoder for å utnytte de tilgjengelige ressursene mer effektivt. Effektivisering er et nøkkelord for videre crawlerutvikling. Bruke minst mulig tid og være minst mulig aktiv på å utføre størst mulig og mest vesentlig jobb. Intelligente algoritmer som kan estimere forandringsrater og sannsynligheter for forandringer, kan brukes til å gjennomføre en slik form for selektiv crawling. Med selektiv crawling menes prioritering av elementer i forholdt til hverandre. En prioritering som blir gjort på grunnlag av estimerte forandringssannsynligheter. Dette kan crawleren videre bruke til å oppdatere de elementene som med høyest estimert sannsynlighet er forandret og på denne måten allokere ressursene best mulig. Slik funksjonalitet blir med en fellesbetegnelse omtalt som Adaptive Metoder. En Adaptiv crawler vil altså intelligent kunne avgjøre hvilke sider som med høyest sannsynlighet er forandret og bruke dette til å utføre selektiv crawling. På denne måten kan crawleren mer effektivt kunne utnytte de tilgjengelige netterksressursene.

3.10 Oppsummering

Kapittelet omtaler generelle utfordringer knyttet til crawleprosessen. Praktiske problemer knyttet til generell nettverksproblematikk, DNS, variasjon i HTTP-implementasjoner, HTML koding, serverapplikasjoner, duplikatdeteksjon og utigjengelige deler av weben blir alle presentert og diskutert. Med utgangspunkt i de presenterte utfordringene kan man understreke behovet for en konfigurerbar arkitektur samt adaptiv crawlerfunksjonalitet.

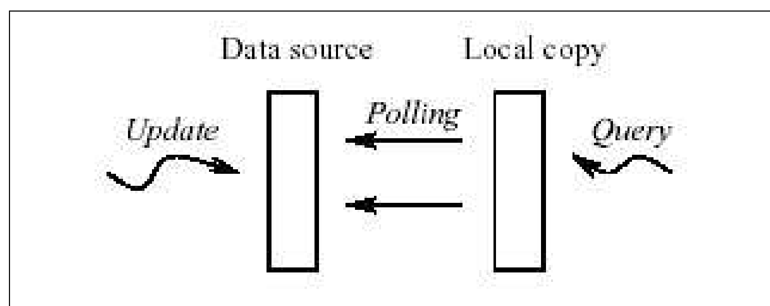
Kapittel 4

Adaptive Metoder

4.1 Introduksjon

Som indikert i forrige kapittel er bruk av adaptive metoder nødvendig for å oppnå en effektivisering i bruken av tilgjengelige ressurser ved crawling. I dette avsnittet gis det et innblikk i eksisterende arbeider rundt adaptive crawleragenskaper. Metodene blir diskutert, satt opp mot hverandre og vurdert. Til slutt presenteres en metode som senere implementeres, testes og dokumenteres. Hele seksjonen tar utgangspunkt i flere forskningsarbeider gjort innenfor emnet [11], [9], [12].

I arbeidet med å holde dataindekser oppdatert, er det typisk at de lokale oppdateringene skjer uavhengig av de faktiske endringene. Dette fordi den lokale oppdateringsprosessen ikke blir initiert av den faktiske forandringen. Programvaren som oppdaterer den lokale indeksen må derimot kontinuerlig spørre og sjekke hvorvidt en endring har funnet sted og eventuelt oppdatere den lokale versjonen. Se Figur 4.1. Utfordringen



Figur 4.1: Illustrerer hvordan lokalindekser oppdateres

er å finne en effektiv algoritme som maksimerer den lokale indeksens ferskhet. I oppgaven diskuteres ulike metoder for synkronisering av ekstern data for å gi en optimal ferskhet på den lokale indeksen. Hovedpunktene er som følger:

- Det presenteres et felles rammeverk for å studere problematikken rundt synkronisering.
- Flere aktuelle synkroniserings metoder som er i bruk idag bli presentert og effektiviteten vurdert. Det vil vise at mange metoder som tilsynelatende virker fornuftige i virkeligheten gir dårligere resultater enn andre mer enkle algoritmer.
- En ny synkroniseringsalgoritme blir presentert som i mange tilfeller kan gi oppsiktsvekkende forbedringer i ferskhet. Denne algoritmen tar i betrakning hvor ofte en side forandrer seg (kartlegger endrings-historikk), hvor viktig en side er og utifra dette utfører en passende oppdateringsavgjørelse.

4.2 Rammeverk

For å kunne studere synkroniseringsaspekter er det nødvendig å forstå begrepene ferskhet og alder. Med *base* vil jeg her mene en samling av filer.

- Ferskhet: Hvis en base A har 10 av 20 elementer oppdatert, mens basen B har 15 av 20 elementer oppdatert regnes basen B for å være ferskere enn basen A. Ferskheten har verdien 1 hvis siden er oppdatert og verdien 0 hvis den ikke er oppdatert [1]. Verdien for en base blir da gjennomsnittet av verdiene for alle elementene i basen til en gitt tid (For base A: 0.5, og for base B: 0.75). Et viktig poeng er at for å kunne beregne verdien for ferskhet, må man vite nøyaktig samtidig hvorvidt alle sidene i samlingen er oppdatert eller ikke. Noe som gjør det praktisk vanskelig å beregne denne verdien. Det blir senere beskrevet en metode for å beregne verdien på grunnlag av en modell over hvordan den faktiske weben forandrer seg.

  Man kan i tillegg bruke en vektet ferskhetsdefinisjon. Slik ferskhet har vært definert hittil bidrar alle sidene i en samling på N elementer likt (1/N). Ved å innføre en vektet ferskhet kan man utifra hvor viktig en side er i forhold til andre gi den en vektet ferskhet (den vanlige verdien multiplisert med en konstant for vektingen).

[1] Dette er en vesentlig begrensning i seg selv

- Alder: Er definert som tiden det har gått fra en side sist ble endret (ikke lenger oppdatert) til tidspunkt for beregning. Hvis en side A ble forandret for 2 dager siden, og denne ikke er blitt synkronisert så er verdien for alder til A lik 2 (i enheten dager). Ellers hvis siden er oppdatert, så er verdien for alder lik 0. Tilsvarende som for ferskheten blir alderen til en base beregnet ved å ta gjennomsnittet av verdiene til alle elementene i basen. Også for alder kan vekting bli brukt.

Forskjellen på ferskhet og alder er intuitivt at ferskhet er en binær verdi, mens alder kun gir mening når ferskheten er 0. Det vil si at alder ikke har verdi når elementet er synkronisert. Men når elementet derimot ikke er oppdatert indikerer verdien for alder hvor lenge elementet har hatt ferskhet lik 0.

4.2.1 Poissonfordelingen

Det er viktig å vite hvordan webbasen forandrer seg i virkeligheten for å kunne beregne hvor effektive ulike oppdateringsmetoder er. Flere arbeider gjort tidligere forutsetter at weben forandrer seg i tråd med poissonfordelingen [10], [11], [12], [9]. Poissonfordelingen brukes ofte for å beskrive en rekke hendelser som skjer uavhengig med ujevne mellomrom dog med konstant rate over tid. Eksisterende litteratur som er referert indikerer at poissonfordelingen er en god tilnærming for å beskrive faktiske endringer på weben. Vi forutsetter at vi oppdaterer en lokal indeks ved $t = 0$ og $t = I$. Ved bruk av poissonfordelingen kan da forventet ferskhet til en gitt tid t uttrykkes slik:

$$E[F(e_i;t)] = e^{-\lambda t} \text{ for } t \in (0, I) \tag{4.1}$$

På tilsvarende måte kan det enkelte elementets alder beskrives:

$$E[A(e_i;t)] = t\left(1 - \frac{1 - e^{-\lambda t}}{\lambda t}\right) \text{ for } t \in (0, I) \tag{4.2}$$

λ er forandringsraten (se avsnitt 4.2.2) som elementet forandrer seg med [2].

4.2.2 Utviklingsmodell for en base

Tidligere er det indikert hvordan man ved hjelp av poissonfordelingen kan beskrive ferskhet og alder for et enkelt element. I denne seksjonen

[2]Denne kan estimeres på grunnlag av de ulike elementenes faktiske forandrings-historikk, og blir omtalt senere.

fokuseres det på en base i sin helhet bestående av mange enkeltelementer. Avhengig av hvordan enkeltelementene i en base forandrer seg, kan en reell base med websider beskrives ved følgende modeller:

- Uniform forandringsrate: I denne modellen regner en med at alle elementer forandrer seg med lik forandringsrate λ. En enkel modell som kan benyttes i følgende scenarioer:
  - Frekvensen for hvordan de ulike elementene forandrer seg er ukjent. Men en kjenner til den gjennomsnittlige forandringsraten for hele databasen og bruker denne som forandringsraten (λ) for de enkelte elementene.
  - De individuelle elementene forandrer seg ulikt men med høyst minimal forskjell i rate. Denne modellen vil da være en god tilnærming til hvordan den virkelige situasjonen er.
- Variabel forandringsrate: I denne modellen regner en med at elementene i basen forandrer seg med vidt forskjellige rater. Et enkeltelement forandrer seg med raten λ_i. En kan plotte denne informasjonen hvor andelen elementer som forandrer seg med en gitt rate, opererer som en funksjon av raten. Ved å sette raten langs x-aksen (λ) og andelen langs y-aksen (%), vil kurven nærme seg en normalfordelingskurve når λ_i varierer og antall elementer øker.

4.3 Presentasjon av ulike synkroniseringsmetoder

Så langt er det beskrevet hvordan referansebasen forandrer seg over tid. I denne seksjonen omtales metoder for å oppdatere den lokale indeksen.

1. Synkroniseringsfrekvens: Det er essensielt å definere hvor ofte man skal synkronisere den lokale indeksen. Desto oftere man gjør dette desto ferskere vil den lokale indeksen være. Man kan definere at en synkroniserer N elementer per I tidsenheter. Ved å variere I kan man bestemme hvor ofte man vil synkronisere databasen. I tillegg vil da $f = 1/N$ som gjenspeiler gjennomsnittlig synkroniseringsfrekvens for et element i basen.

2. Ressursallokering: I tillegg til å bestemme synkroniseringsfrekvensen (hvor mange elementer per tidsenhet), er det essensielt å definere hvor ofte man synkroniserer hvert enkelt element, som kan gjøres på to forskjellige måter:

(a) Uniform ressursallokering: Uavhengig av hvordan de individuelle elementene forandrer seg, oppdaterer vi alle elementene med med lik frekvens f.

(b) Elementavhengig ressursallokering: De ulike elementene oppdateres med ulik frekvens. For eksempel kan et enkelt element oppdateres med en frekvens f_i proporsjonalt med forandringsraten λ_i. Dette gir et entydig forhold f_i/λ_i for alle elementer i en gitt base.

3. Synkroniseringsrekkefølge: I hvilken rekkefølge skal elementene i en base oppdateres.

   (a) Fast rekkefølge: Alle elementene blir hver gang synkronisert i den samme rekkefølgen. Dette gir et periodisk konstant synkroniseringsintervall for et enkelt element i en base.

   (b) Variabel rekkefølge: Alle elementene blir synkronisert, men i en tilfeldig rekkefølge. Det blir plukket ut en tilfeldig permutasjon av elementene i hver løkke og elementene blir oppdatert i den permuterte rekkefølgen. Denne metoden gir også et konstant synkroniseringsintervall for et enkelt element i basen (fordi alle sidene blir oppdatert hver gang, kun rekkefølgen varierer).

   (c) Sterk variabel rekkefølge: Ved hvert synkroniseringspunkt blir en tilfeldig side plukket ut og synkronisert. Dette gir ikke et konstant synkroniseringsintervall for et gitt element da det alltid er lik sannsynlighet for at et element blir trukket ut. Uavhengig av hva som er blitt trukket ut tidligere.

4. Synkroniseringstidspunkt: Man kan bestemme en synkroniseringsfrekvens på 5 ganger per dag. Men disse oppdateringene kan være spredt med jevne mellomrom utover hele døgnet, samlet helt i starten, slutten, eventuelt helt tilfeldige tidspunkter. Avgjørelsen baseres i praksis på mengden av forespørsler på ulike tjenere slik at man ikke spør en tjener i perioder med mye trafikk. Generelt sett kan man forutsette at en base (som i vår kontekst er en webbase) burde bli oppdatert med jevne mellomrom [11]. Dette fordi weben strekker seg over flere tidssoner og perioder med trafikk er spredt utover hele døgnet. Fordi det er vanskelig for en crawler å forutsi et riktig tidspunkt for alle webportaler, er det best å oppdatere med jevne mellomrom [3].

[3] best for crawleren

4.4 Vurdering av ulike synkroniseringsrekkefølger

I denne seksjonen tas de nevnte synkroniseringsmetodene opp og vurderes i henhold til ferskhet og alder. Man går utifra at alle database elementer forandrer seg med den samme forandringsraten λ. Når elementer forandrer seg likt, er det unødvendig å synkronisere med forskjellige rater. Derfor går man i tillegg utifra en uniform ressursallokering for synkroniseringen, som indikerer entydig synkroniseringsrate for alle elementene i basen. På grunnlag av disse forutsetningene vurderes de ulike synkroniseringsrekkefølgene.

4.4.1 Fast rekkefølge

Ved hjelp av tidligere gitte uttrykk, et teorem og litt matematikk presenteres følgende uttrykk for ferskheten til en database:

$$\bar{F}(S) = t\left(1 - \frac{1 - e^{-\lambda/f}}{\lambda/f}\right) \text{ hvor } f = 1/I \tag{4.3}$$

f gjenspeiler den gjennomsnittlige synkroniseringsraten for basen og λ indikerer den gjennomsnittlige forandringsraten for den virkelige basen. Det betyr at man kan beregne gjennomsnittlig ferskhet til en base gitt synkroniseringsrate og forandringsrate som inn verdier. På tilsvarende måte gis et uttrykk for basens alder:

$$\bar{A}(S) = \frac{1}{f}\left(\frac{1}{2} - \frac{1}{\lambda/f} + \frac{1 - e^{-\lambda/f}}{(\lambda/f)^2}\right) \tag{4.4}$$

4.4.2 Variabel rekkefølge

Ved bruk av en slik politikk vil rekkefølgen elementene synkroniseres i variere fra en oppdatering til en annen. Rekkefølgen blir permutert før hver oppdatering. Etter en matematisk utledning kommer en frem til følgende uttrykk for basens ferskhet:

$$\bar{F}(S) = \frac{1}{\lambda/f}\left[\left(1 - \frac{1 - e^{-\lambda/f}}{\lambda/f}\right)^2\right] \tag{4.5}$$

Tilsvarende for alder:

$$\bar{A}(S) = \frac{1}{f}\left[\frac{1}{3} + \left(\frac{1}{2} - \frac{1}{\lambda/f}\right)^2 - \left(\frac{1 - e^{-\lambda/f}}{(\lambda/f)^2}\right)^2\right] \tag{4.6}$$

4.4.3 Sterk variabel rekkefølge

Hver gang et element skal oppdateres plukkes et element tilfeldig ut fra basen. Det gir følgende uttrykk for ferskhet:

$$\bar{F}(S) = \frac{1}{1 + \lambda / f} \tag{4.7}$$

of for «age»:

$$\bar{A}(S) = \frac{1}{f} \left(\frac{\lambda / f}{1 + \lambda / f} \right) \tag{4.8}$$

4.4.4 Sammenligning av ulike synkroniseringsrekkefølger

Ved hjelp av grafisk analyse for de gitte uttrykkene vurderes de ulike metodene opp mot hverandre [11]. Det fremgår av [11] at en bestemt rekkefølge gir best resultater på både ferskhet og alder. Forskjellen varierer med hensyn på hvor ofte man synkroniserer i forhold til hvor ofte elementene virkelig forandrer seg, men uansett så er en fast rekkefølge bedre.

4.5 Vurdering av ulike ressursallokeringsmetoder

4.5.1 Introduksjon

I forrige avsnitt ble det forutsatt at elementer forandrer seg med samme rate. Men hvis elementene ikke forandrer seg med samme rate (mer reellt) og man vet de ulike ratene for de forskjellige elementene, kan man utnytte dette. I en situasjon med mulighet for høyere synkroniseringsrate enn faktisk forandringsrate blir det diskutert hvorvidt dette er positivt eller ikke. Utgangspunktet for diskusjonen blir en reell base med variabel forandringsrate og fast synkroniseringsrekkefølge fordi denne viser seg å være den beste synkroniseringsmetoden. Det vil si at hvert element forandrer seg med forskjellig rate, og oppdateres med like intervaller.

4.5.2 Uniform vs. Elementavhengig ressursallokering

I [11] blir det bevist at uniform ressursallokering alltid er bedre enn en elementavhengig ressursallokering. Dette gjelder uansett hvilken verdi λ har. Beviset er gjort med utgangspunkt i påstanden om at ferskhet for

en uniform taktikk alltid er bedre enn for en proporsjonal taktikk [4]. Ved bruk av egenskapene til konvekse og konkave funksjoner (uttrykket for ferskhet gir en konveks funksjon mens alder gir konkav) utføres beviset. For konvekse funksjoner gjelder:

$$\frac{1}{n}\sum_{i=1}^{n} f(x_i) \geq f(\frac{1}{n}\sum_{i=1}^{n} x_i) \text{ for alle } x_i \text{ hvor } (i = 1, 2, \cdots, n) \tag{4.9}$$

For konkave funksjoner gjelder:

$$\frac{1}{n}\sum_{i=1}^{n} f(x_i) \leq f(\frac{1}{n}\sum_{i=1}^{n} x_i) \text{ for alle } x_i \text{ hvor } (i = 1, 2, \cdots, n) \tag{4.10}$$

4.5.3 Praktisk tolkning av Uniform vs. Elementavhengig ressursallokering

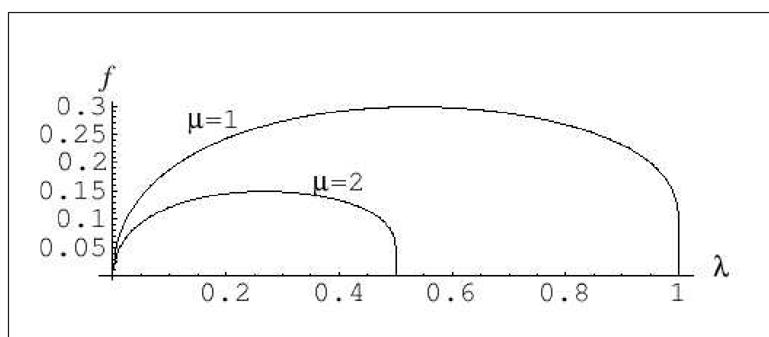
Intuitivt skulle det virke som en proporsjonal metode skulle være bedre enn en uniform allokeringsmetode. Dette fordi man gir elementer med hyppigere forandringsrate (λ) mer oppdateringsressurser enn et element med lavere forandringsrate. Men hvorfor er det da slik at en uniform ressursallokering, teoretisk slik som beviset tilsier, er bedre? For å få en praktisk forståelse for det teoretiske resultatet i forrige avsnitt studeres et eksempel med en base på to elementer e_1 og e_2. Man forutsetter at e_1 forandrer seg 9 ganger per dag mens e_2 forandrer seg 1 gang per dag. La oss si at vi bestemmer oss for å synkronisere kun ett element per dag. Hvilket av elementene er da lurest å synkronisere med tanke på å gi basen høyest mulig ferskhet: e_1 eller e_2?

Vi ser først på hvilken innvirkning e_2 har på raten for ferskhet. Hvis elementet forandrer seg halvveis på dagen eller halvveis i forandringsintervallet [5] og blir synkronisert rett etter dette, så vil elementet med den gitte forandringsraten være oppdatert i en $\frac{1}{2}$ dag. Sannsynligheten for at elementet skal forandre seg på første halvdel av dagen er også $\frac{1}{2}$ som gir en gjennomsnittlig synkronisert tid på $\frac{1}{4}$ dag.

Tilsvarende for e_1 så vil en forandring finne sted på et tilfeldig sted i forandringsintervallet, men gjennomsnittet av et tilfeldig utvalg blir som tidligere halvveis i intervallet. Elementet e_1 vil da være oppdatert i halvparten av intervallet det ble synkronisert i. Det vil si $\frac{1}{2} * \frac{1}{9}$. Videre er

[4] som i og for seg er elementavhengig

[5] som er gjennomsnittet av tidspunktene for et tilfeldig antall forandringer når antallet går mot uendelig



Figur 4.2: Plot av optimal ressursallokeringsmetode

sannsynligheten for at forandringen skjer på første halvdel av dagen (det vil si før synkroniseringen for den aktuelle dagen) lik $\frac{1}{2}$ som gir et resultat på gjennomsnittlig synkronisert tid på $\frac{1}{2} * \frac{1}{9} * \frac{1}{2} = \frac{1}{36}$ dag.

Av dette ser man at det vil være fordelaktig for ferskheten å oppdatere e_2 istedetfor e_1. Dette er ikke i nærheten av noe bevis for hvorvidt en uniform ressursallokering alltid er bedre enn en elementavhengig ressursallokering. Men flere arbeider referert i oppgaven hvor forskjellige scenarioer er studert, gir de samme resultatene.

4.6 Optimal ressursallokering

4.6.1 Introduksjon

I artikkelen [11] presenteres det også en optimal algoritme for å allokere crawlingressurser. Metoden går veldig forenklet ut på å beregne synkroniseringsrater f_i for ulike elementer i en base gitt forandringsratene λ_i for hvert enkelt element. Disse synkroniseringsratene tar sikte på å maksimere den totale ferskheten til basen. Ved bruk av Lagranges metode [2], beregnes de ulike f_i avhengig av ulike λ_i. Det blir ikke presentert en detaljert fremgangsmetode for hvordan utføre selve beregningen, men et eksempel bli presentert og løst analytisk. Hensikten med å løse problemet analytisk er å bevise at grafen over ulike f_i som plottes som funksjon av ulike λ_i, alltid har samme formen. Det bevises at formen på grafen [6] for optimale synkroniseringsrater alltid er lik, men skaleres med en konstant μ som videre avhenger av distribusjonen av de ulike λ_i i basen. Av dette

[6]som vist på Figur 4.2

konkluderes det at i ethvert tilfelle [7], så vil det alltid gi positive resultater på ferskheten å straffe sider som forandrer seg altfor ofte. Av grafen ser man at det er like dumt å oppdatere et element som forandrer seg altfor hyppig, som å oppdatere et element som forandrer seg for sjeldent i forhold reell synkroniseringsrate.

4.6.2 Metode

Den optimale ressursallokeringsmetoden gjør som sagt bruk av Lagranges metode for å beregne virkelige synkroniseringsrater. Funksjonen ser slik ut:

$$\mu = d\bar{F}(\lambda_i, f_i)/df_i \tag{4.11}$$

Funksjonen $\bar{F}$ er uttrykket for ferskheten som er ulikt for de forskjellige synkroniseringsrekkefølgene [8], som alle har forskjellige funksjonsuttrykk for å beskrive ferskheten [11]. Det som er viktig å merke seg ved uttrykket er innverdiene λ_i (forandringsraten) og f_i (synkroniseringsraten) og konstanten μ som er entydig for en gitt base, men forskjellig for ulike baser. Dermed vil hver base med N elementer få N likninger med utgangspunkt i uttrykket ovenfor med N antall $f_i$er og μ som ukjente. Det vil si N likninger med N+1 ukjente for hver base. Det som er interessant for oss er $f_i$ene for de ulike elementene som forteller oss hvor ofte vi burde oppdatere de ulike elementene.

I tillegg til uttrykket ovenfor finnes det ett uttrykk til. Nemlig at gjennomsnittet av synkroniseringsfrekvensene f_i er lik gjennomsnittet for synkroniseringsfrekvensen for hele basen:

$$\frac{1}{N}\sum_{i=1}^{N} f_i = f \tag{4.12}$$

Fordelen med å innføre dette uttrykket er at vi kjenner gjennomsnittsfrekvensen f. Denne kjenner vi ganske enkelt ved å forutsi hvor mange elementer som skal oppdateres [9]. Dermed får vi en likning til som med sitt bidrag resulterer i N+1 likninger med N+1 ukjente.

Metoden består altså av å beregne en skaleringsfaktor μ på grunnlag av distribusjonen av forandringsrater λ for de ulike elementene i en

[7] forskjellige baser med forskjellige λ_i distribusjoner

[8] 1. Fast rekkefølge, 2. Variabel rekkefølge, 3. Sterk variabel rekkefølge

[9] Hvis vi har en base med 1000 elementer og vi kan oppdatere totalt 2000 elementer innenfor en oppdateringssyklus / per tidsenhet, så blir gjennomsnittsfrekvensen for basen $f = 2$.

gitt base. Denne skaleringsfaktoren blir brukt i sammenheng med grafen for synkroniseringsraten f som (når μ er bestemt) er funksjonsverdien av λ, Figur 4.2. Det bevises at denne grafen, som er en funksjon av forandringsraten, alltid har samme form. Det betyr at enhver base, med ulike elementer som forandrer seg med vidt forskjellige forandringsrater, alle har samme form på grafen. Eneste forskjellen er at de skaleres med konstanten μ, som beregnes på grunnlag av forandringsratene λ som er ulik for de ulike basene. Dermed blir skaleringsfaktoren μ også forskjellig for ulike baser. Grafen eller funksjonen blir videre brukt for å lese ut synkroniseringsrater på grunnlag av forandringsrater λ. Akkurat hvordan man skal beregne denne μ er altså utfordringen.

4.6.3 Matematisk utfordring

Problemet er at metoden for å beregne μ baserer seg på løsning av et likningssett med N+1 likninger og N+1 ukjente hvor N er antall elementer (sider) i en base. Det ville tilsynelatende blitt en altfor tidkrevende oppgave i et reellt tilfelle å løse et slikt likningssett. Vi kontaktet i slutten av November 2005 forfatteren selv (Junghoo Cho) hvor vi gjennom en korrespondanse fikk bekreftet tankegangen. Når N øker til et antall som er reellt for en crawlerindeks, vil likningssettet i seg selv bli såpass komplekst at å løse dette vil bli en for tidkrevende operasjon. Ideen videre var å få personer med mer numerisk innsikt til å komme med forslag eller bidrag som kunne virkeliggjøre en praktisk ikke gjennomførbar metode. Men mangel på tilbakemelding stoppet fremdriften på dette området.

Det som likevel er essensielt med teoriene som presenteres og som ikke må overses er at elementer som har en altfor hyppig forandringsrate må straffes. Å oppdatere sider som forandrer seg veldig ofte er ikke alltid det beste. Det avhenger av hvor ofte man er i stand til å crawle og hva slags ressurser man har tilgjengelig i forhold til dette. Det er viktig å vurdere hvilket element som bidrar mest til å holde ferskheten optimal over lengst mulig periode. Et element som da forandrer seg altfor ofte (i forhold til synkroniseringsfrekvensen) burde da nedprioriteres i forhold til et annet element som forandrer seg mer sammenfattende med synkroniseringsfrekvensen.

4.6.4 Konklusjon

En viktig forutsetning for å få noe som helst ut av den optimale synkroniseringsmetoden er å kjenne til forandringsraten til de ulike

elementene i en base. Ved korrespondansen med forfatter Junghoo Cho anbefalte han å dele inn elementene i såkalte grupperinger med hensyn på forandringsrater. Praktisk vil det bety at det opprettes en gruppe for sider som forandrer seg hver dag, en for sider som forandrer seg hver uke, en for hver måned osv. Istedenfor å beregne ulike forandringsrater for alle individuelle elementer (tidkrevende), så klassifiseres heller elementene inn i grupperinger. Da antall elementer vokser vil uansett denne tilnærmingen være god.

En viktig fellesnevner for alle metoder presentert i [11], er at de baserer seg på å få den gjennomsnittlige oppdateringsfrekvensen som innverdi sammen med forandringsraten λ (se funksjonen). Dette betyr at man på forhånd må vite hvor mange elementer man vil oppdatere noe man ikke alltid vet.

Uansett om man vil bruke optimal metode, eller ikke så er det uansett viktig å beregne virkelige forandringsrater. Det er nødvendig for i det hele tatt å kunne gjøre en adaptiv tilnærming på oppdateringsproblematikken. Forandringsraten må være kjent.

4.7 Estimering av forandringsrater

4.7.1 Introduksjon

I denne seksjonen presenteres metoder for å estimere forandringsrater for ulike elementer [9], [12]. Estimeringen baserer seg på gjentatte besøk av ulike elementer på nett gjennom ordinære aktiviteter som for eksempel crawling. Utifra disse besøkene lagres status på hvorvidt de ulike elementene som besøkes er forandret eller ikke. Disse dataene blir så utgangspunkt for å estimeringen.

4.7.2 Forberedende momenter

Det er viktig å ha klart for seg en del innledende momenter før en går igang med å estimere forandringsrater.

1. Hvordan kartlegges forandringshistorien til et element ?

    - Passiv metode: I mange kontekster har en ikke kontroll over når eller hvor ofte en kan besøke et element. Eksempelvis kan en webcache nevnes. Her vil ulike besøk til ulike websider være brukerinitiert. Det vil derfor ikke være noe regelmessig eller forutsigbart besøksmønster. Utfordringen i slike tilfeller

er å analysere den gitte forandringshistorikken til de ulike elementene for å kunne estimere en korrekt forandringsrate.

- Aktiv metode: I slike tilfeller kan en kontrollere besøksmønsteret mot ulike elementer. Som et eksempel må vi nevne en crawlerkontekst. En crawler kan konfigureres til å definere hvor ofte og når den skal oppdatere ulike websider. Utfordringen i en slik situasjon er hvor ofte crawleren burde besøke et element for å kunne estimere en korrekt forandringsrate.

I tillegg til kontroll av besøk, vil ulike kontekster ha forskjellige lengder på oppdateringsintervallene.

- Regelmessige intervaller: I mange tilfeller, spesielt for aktiv kartlegging, blir elementene besøkt i regelmessige intervaller. Slike situasjoner gjør det enklere å estimere forandringsrater.
- Uregelmessige intervaller: Dette gjelder spesielt for passiv kartlegging og gjør det mer komplekst å beregne forandringsrater for ulike elementer.

2. Hvilke data er kjent ?

- Komplett forandringshistorikk: Vi kjenner til akkurat når og hvor ofte ulike elementer har forandret seg. Det er i lignende situasjoner forholdsvis trivielt å estimere korrekte forandringsrater.
- Siste forandring: Vi kjenner til når elementene forandret seg sist, men ikke komplett historie for forandringene.
- Endringseksistens: Vi vet at det er forekommer en forandring per element. Men det er ikke kjent tidspunkt for forandringen eller hvor mange som har forekommet i en gitt tidsperiode.

3. Hvordan benyttes estimerte forandringsrater ?

- Estimering av forandringsrate: I noen tilfeller er det nødvendig å presist kunne estimere forandringsrater for ulike lementer.
- Kategorisert forandringsrater: Det er i noen situasjoner tilstrekkelig å gruppere elementer i ulike grupperinger basert på intervaller av forandringsrater. Som for eksempel en crawler ville det vært tilstrekkelig å kunne definere kategorisere ulike elementer. En presist estimert forandringsrate vil derfor være unødvendig. Hvis det er mulig å kunne bestemme i hvilke intervaller et

element skal crawles, så er dette tilstrekkelig informasjon for crawleren.

4.7.3 Forandringsrater i en Crawlerkontekst

I paperen [12] som er en forkortet utgave av originalversjonen [9], blir forskjellige estimatorer presentert og vurdert opp mot hverandre. De er utformet utifra forskjellige kontekster og behov slik som omtalt i avsnittet ovenfor. Metoden for å fordele elementer i ulike kategorier egner seg godt i en crawlerkontekst. Ofte jobber crawlere utifra ulike kategorier med hensyn på oppdateringsintervaller. En kan derfor utnytte kategoriserte forandringsrater til å definere hvilke elementer som skal oppdateres i forhold til de ulike intervallene crawleren er konfigurert med. På denne måten vil en kunne automatisere en tidkrevende prosess med å fordele start-URL'er i ulike kategorier, basert på hvor dynamiske eller statiske websidene er. Metoden som benyttes baserer seg på bayesiansk slutning.

4.8 Bayesiansk slutning

4.8.1 Introduksjon

En crawler kan være konfigurert til å re-crawle kolleksjonen sin periodisk. I slike tilfeller trenger den ikke vite hvorvidt en side forandrer seg hver time, hver tredje time, hver dag, hver fjerde dag, hver uke også videre. Hvis crawleren er satt opp til [10] å kjøre i periodiske intervaller, så trenger crawleren bare vite hvilket intervall elementet skal oppdateres i. Det vil si hvorvidt elementet skal oppdateres hver dag, hver uke, hver måned også videre, i henhold til faktiske kategorier crawleren jobber etter. Dette kan utføres fint med de avanserte metodene for å estimere presise forandringsrater, men kan gjøres mer effektivt og tilpasset til et scenario med ufullstendig historie ved bruk av Bayesiansk slutning.

4.8.2 Metode

I utgangspunktet defineres aktuelle kategorier. For enkelhetens skyld omtales her et oppsett med to kategorier:

- A: en endring per måned

[10] slik FAST sin crawler fungerer idag

- B: en endring per uke

Metoden består i å beregne sannsynligheter for at et element tilhører de ulike kategoriene. Sannsynligheten for at et element tilhører en av disse kategoriene er da i utgangspunktet 0.5 for hver kategori slik at summen blir 1. Dette er utgangspunktet for videre beregning. Metoden består i å vedlikeholde disse sannsynlighetene for alle elementer etterhvert som de forandrer seg, hvor elementet hører til kategorien med høyest sannsynlighetsverdi. Hvis et element e_1 for kategori A har sannsynlighetsverdi 0.8 og for kategori B verdien 0.2, hører elementet til kategori A og skal oppdateres sammen med elementer i denne kategorien. Verdiene oppdateres hver gang det detekteres en forandring. Man kan ikke detektere mer enn maks 1 endring per aksess, som gjør at siden kan ha forandret seg fra og med 0 til uendelig mange ganger. Selvom man skulle tro at en ufullstendig historie gir ufullstendige resultater, viser forsøker at resultatene er heller gode [9]. Følgende uttrykk beregner et elements tilhørighetssannsynlighet P_A mot kategorien A (månedskategorien) ved en detektert forandring etter 5 dager (utgangspunktssannsynlighet er 0.5 pga 2 kategorier):

$$P_A = \frac{(1 - e^{-5/30})0.5}{(1 - e^{-5/7})0.5 + (1 - e^{-5/30})0.5} = 0.23 \tag{4.13}$$

Mens følgende uttrykk beregner det samme elementets tilhørighetssannsynlighet P_B mot kategorien B (ukeskategorien) ved en detektert forandring etter 5 dager (utgangspunktssannsynligheten er 0.5 pga 2 kategorier):

$$P_B = \frac{(1 - e^{-5/7})0.5}{(1 - e^{-5/7})0.5 + (1 - e^{-5/30})0.5} = 0.77 \tag{4.14}$$

I neste syklus vil hver av disse uttrykkene utvikle seg slik, hvis nok en endring skjer etter 5 dager:

$$P_A = \frac{(1 - e^{-5/30})0.23}{(1 - e^{-5/7})0.77 + (1 - e^{-5/30})0.23} \tag{4.15}$$

og:

$$P_B = \frac{(1 - e^{-5/7})0.77}{(1 - e^{-5/7})0.77 + (1 - e^{-5/30})0.23} \tag{4.16}$$

4.9 Konklusjon

I denne seksjonen ble flere eksisterende arbeider rundt adaptive egenskaper presentert og vurdert. Med begrensede nettverksressurser er det essensielt for en crawler intelligent å kunne forutsi hvilke elementer som burde oppdateres. For å kunne forutsi hva som gir positive eller negative resultater på lokalbasen, er det viktig å definere hva som menes med positivt og negativt. I sammenheng med dette blir et rammeverk presentert og uttrykkene ferskhet og alder blir innført. Det er positivt for en lokalbase med elementer med maksimal ferskhet og minimal alder. Utifra rammeverket blir en modell for weben utformet. I praksis er det vanskelig å kartlegge hva som virkelig gir positive eller negative følger for en lokalbase. Dette fordi det på et gitt tidspunkt ikke er mulig å kartlegge hvilke elementer som er oppdaterte eller ikke. Prosessen å utføre en slik kartlegging er tidkrevende og med tid endres referansebasen nemlig weben. For å kunne gjennomføre en beregning av andel oppdaterte elementer i en lokalbase trenger man derfor en modell for weben. Ved hjelp av denne kan tilstanden trivielt beregnes matematisk som igjen kan benyttes til å kartlegge hvilke elementer som er oppdaterte eller ikke.

Videre blir ulike synkroniseringsmetoder presentert og vurdert opp mot hverandre. En optimal ressursallokeringsmetode blir omtalt, som teoretisk kan forsvares men praktisk byr på matematiske utfordringer og gjør den unyttbar i en reell crawlerkontekst.

Til slutt blir forandringsrater for elementer på web omtalt, og hvor avhengige ulike synkroniseringsmetoder er av nøyaktige forandringsrater for å gjøre en presis og meningsfull synkronisering. En passende metode for crawlerkontekster blir omtalt til slutt. Metoden går veldig forenklet ut på å kategorisere elementer med hensyn på hvordan disse forandrer seg. En crawler trenger ikke vite akkurat hvor lang tid det vil ta før individuelle elementer forandrer seg. Det holder å effektivt fordele elementer i ulike kategorier som oppdateres med ulike mellomrom, slik at når først en kategori oppdateres vil maksimalt antall elementer være forandret og nødvendig å synkronisere.

Kapittel 5

Implementasjon av adaptiv crawling for FAST-Crawleren

5.1 Introduksjon

Metoden for beregning av forandringsrater ved hjelp av Bayesiansk Slutning blir i denne seksjonen omtalt. Som antydet i slutten av forrige kapittel er denne direkte nyttbar i en crawlerkontekst.

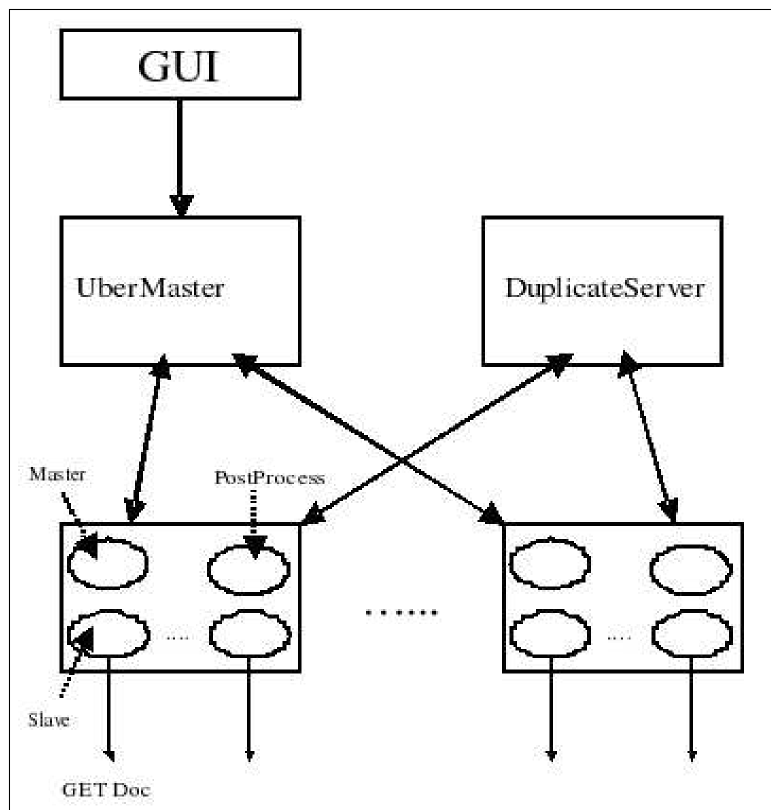
Arkitekturen og designet til en modul som klassifiserer elementer blir planlagt og presentert. Denne blir deretter implementert i sammenheng med FAST-Crawleren for så å dokumentere et eksperiment med dets resultater. Modulen inneholder generiske metoder for oppretting av datastruktur samt logikk som på grunnlag av Bayesiansk Slutning opererer på de aktuelle datastrukturene og behandler elementer.

5.2 Arkitektur og Design

5.2.1 FAST Crawlerarkitektur

FAST-Crawleren er implementert i programmeringsspråket python og er en typisk periodisk crawler som opererer i «batch» med en «in-place» oppdatering av lokalbasen. Som tidligere indikert antyder dette at elementene etterhvert som de blir oppdatert, fortløpende blir lagt til indeksen tilgjengelig for søk. At crawleren er periodisk gjør den lettere og mer naturlig å tilpasse metoden for Bayesiansk Slutning.

Implementasjonen av FAST crawleren er basert på en distribuert struktur [14], se Figur 5.1. Kjøresystemet er modulært som gjør det enkelt å distribuere over flere noder utifra hvor stort crawl som skal



Figur 5.1: FAST's crawlerarkitektur

utføres. Felles for en enkelt- og multi- node installasjon er at det kun eksisterer en forekomst av UberMaster og DuplicateServer prosessene. Det kan riktignok konfigures til å kjøre flere replikaer av duplikatserveren, men i utgangspunktet er det kun en DuplicateServer og UberMaster. En enkeltnodeinstallasjon har i tillegg til UberMaster og DuplicateServer følgende prosesser:

1. Master

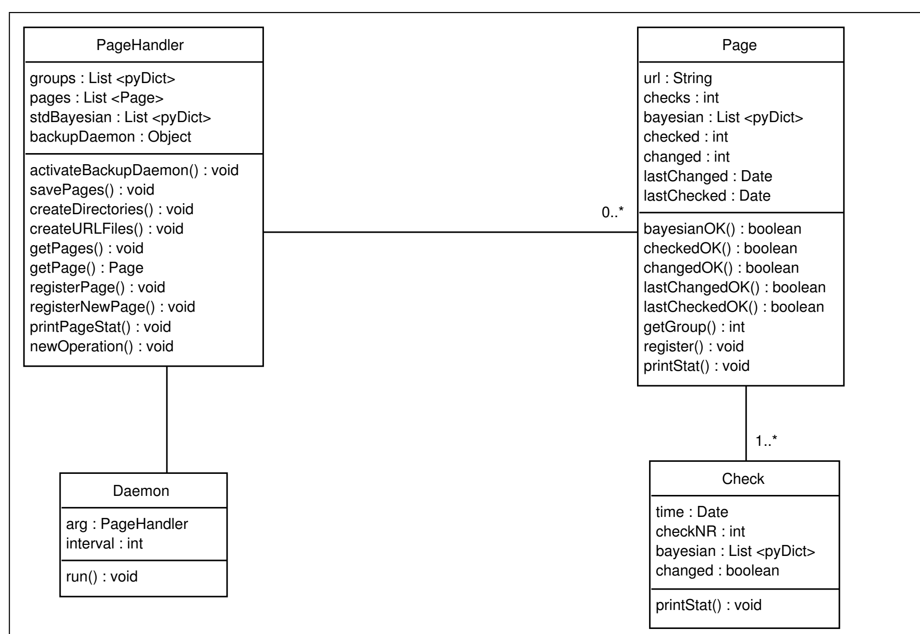
2. PostProcess

3. En eller flere Slave, typisk en Slave per tjener.

De nummererte prosessene er typisk installert på samme node. Det vil si at i en enkeltnodeinstallasjon vil samtlige nummererte prosesser samt UberMaster og DuplicateServer være installert på en og samme node. I en multinode-installasjon vil det derimot i tillegg til en node tilsvarende enkeltnodeinstallasjon-noden, være flere noder som hver har ett sett med de nummererte prosessene, se Figur 5.1.

Master prosessen adminstrerer og delegerer aktiviteter og dokument-nedlastinger mellom Slave prosessene. Det vil dynamisk bli opprettet en Slave prosess per tjener. Etter å ha fått en link for henting fra Master prosessen, er det Slave-prosessen sitt ansvar å laste ned dokumentet og hente ut flere linker fra det nedlastede dokumentet og rapportere disse tilbake til Master-prosessen. Linkene som blir rapportert tilbake til Master-prosessen blir videre delegert blant Slave prosessene innad i noden eller rapportert tilbake til UberMasteren. Hvorvidt linkene blir rapport tilbake til UberMasteren eller delegert innad i noden, er avhengig av hvilke tjenere som behandles av Slave prosessene innad i noden. Mest sannsynlig vil et flertall av linkene distribueres innad i noden på grunn av hyppigere intern- enn ekstern- linking på websteder.

Det er PostProcess sitt ansvar å rapportere dokument meta- informasjon til resten av systemet. Det er en PostProcess per Master prosess og likeledes en per node i en multinodeinstallasjon. PostProcessen kommuniserer med DuplicateServer for å forsikre seg om at hvis flere duplikater, rapporteres kun et dokument tilbake til resten av systemet. Intuitivt er det da at DuplicateServer har en liste over alle linker og sjekksummer for alle dokumentene. Hver PostProcess fra hver node i en multinodeinstallasjon vil derfor korrekt rapportere kun nye dokumenter til resten av systemet i og med at alle kommuniserer med samme DuplicateServer.



Figur 5.2: Klassediagram over modul for Bayesiansk Slutning

5.2.2 Modul for Bayesiansk Slutning

For å kunne estimere de konkrete sannsynlighetene knyttet til metoden for bayesiansk slutning, er man avhengig av å ha en robust men fortsatt enkel datastruktur, som oversiktlig lagrer informasjonen med minimal dobbeltlagring. Se Figur 5.2 for klassediagram over modulen.

For i ettertid kunne gå inn i datastrukturen å hente ut eventuell informasjon som eventuelt ikke logges, ble cPickle modulen fra standard api'et til python brukt. For å integrere denne med datastrukturen er det laget to metoder som dumper hele kjøremiljøet til fil samt kan laste inn et kjøremiljø fra fil. Man kan dermed regelmessig dumpe kjøresmiljøet til disk for så i ettertid hente ut all informasjon som er tilgjengelig i datastrukturen. I tilfelle loggene ikke gir tilstrekkelig informasjon, kan fortsatt data hentes ut fra en eventuell fildump.

5.3 Implementasjon

5.3.1 Introduksjon

Implementasjonen av Bayesiansk Slutning for å gi en adaptiv estimator er gjort i programmeringsspråket python. Dette hovedsakelig på grunn av at FAST crawleren er implementert i dette språket fra før, som gjør det forholdsvis enkelt å montere modulen når den skal testes.

5.3.2 Modul for Bayesiansk Slutning

Datastrukturen for å lagre og gjennomføre estimeringen består av følgende komponenter:

1. Pagehandler
2. Page(s)
3. Check(s)

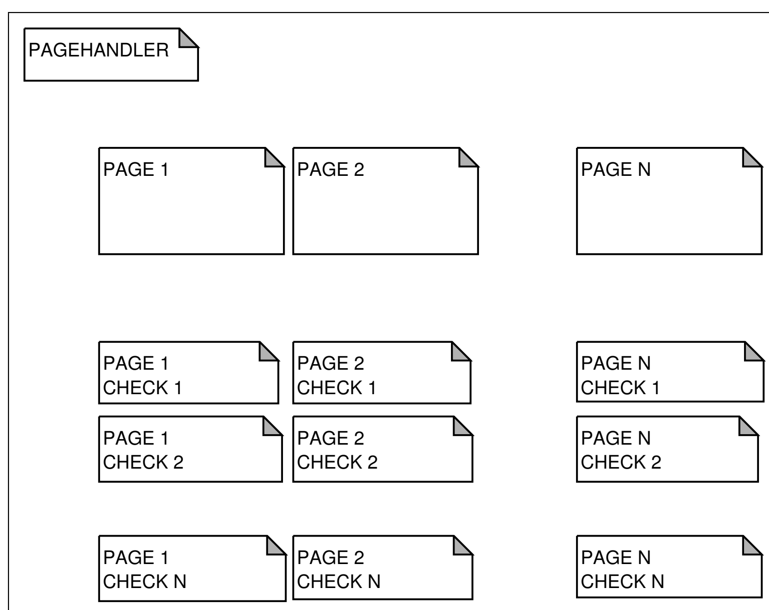
Pagehandleren er administratoren i modulen. Denne initialiseres med:

1. Standard bayesianske kategorier
2. Peker til en liste med «Page» objekter
3. Valgri oppretting av mappestruktur for lagring av esimeringen og andre resultater

Spesielt for klassen er metodene for oppretting av mappestrukturen samt demonisering av kjøringen. Under kjøringen blir det opprettet en mappe for hver forhåndsdefinert bayesiansk kategori. I hver av disse mappene, som blir opprettet for henholdsvis følgende kategorier:

- Daglig
- Hver 3. dag
- Hver 7. dag

, blir det daglig dumpet en fil som inneholder en liste over alle urler som ifølge estimatoren burde oppdateres i de aktuelle intervallene. Demoniseringen av modulen er nødvendig får at den daglige dumpingen av urlene skal fungere. Lagringsdemonen sover i 24 timer før den utfører



Figur 5.3: Organisering av objekter under kjøring

den eneste oppgaven, nemlig lagring til disk. Lagringsaktiviteten er satt opp til å kjøre i en uendelig løkke med 24 timers pauser mellom hver gang.

Det er gjennom Pagehandleren crawleren gir nødvendig informasjon til estimeringen. Initiert av crawleren oppretter Pagehandleren dynamisk Page objekter for hver url. En url har et eget Page objekt, men et Page objekt kan ha flere Check objekter. Se Figur 5.3

Avhengig av om en gitt url har et Page objekt eller ikke kaller Pagehandleren rett metode. Resultatet er enten opphavet av et nytt Page objekt og et initielt Check objekt for dette Page objektet, eller kun et nytt Check objekt som linkes til riktig Page objekt som eksisterer fra før. På denne måten har hver webside med unik url et eget Page objekt, som peker til et antall Check objekter tilsvarende antall ganger elementet er oppdatert.

Hvert Page objekt inneholder informasjon om hver url. I tillegg til url, inneholder Page også følgende data:

1. Siste estimerte sannsynligheter for hver bayesiansk kategori
2. Totalt antall ganger siden er blitt lastet ned og sjekket
3. Totalt antall ganger siden har vært forandret
4. Tidsstempel for siste gang siden ble sjekket

5. Tidsstempel for siste gang siden var forandret

På grunnlag av denne informasjonen blir det ved hver oppdatering laget et nytt Check objekt, hvor ny estimering blir utført og resultatdata lagret. Hvert Check objekt inneholder følgende data:

1. Tidsstempel for når sjekken ble utført

2. Hvilket nummer sjekken er den kronologiske rekken med alle utførte sjekker.

3. Resultatverdiene for den bayesianske estimeringen utført ved sjekken. Denne gjenspeiler de aktuelle verdiene som også lagres i Page objektet.

4. En boolsk verdi som antyder hvorvidt siden var forandret eller ikke ved den aktuelle sjekken.

Selveste estimeringen utføres i bayesian modulen. Denne er helt generisk og kan utføre estimeringen for et oppsett med vilkårlig antall kategorier. Se seksjonen 4.8 for mer informasjon rundt det teoretiske ved implementasjonen. Listing 5.1, 5.2 og 5.3 inneholder kildekoden for henholdsvis page, pagehandler og bayesian modulene.

Listing 5.1: Kildekode for page modulen

```
1  import time
2  from bayesian import *
3
4  class Page:
5      def __init__(self,url,bayesian):
6          self.url = url
7          self.checks = []
8
9          self.bayesian = bayesian
10         self.checked = 0
11         self.changed = 0
12         self.lastChanged = time.time()
13         self.lastChecked = self.lastChanged
14
15     def bayesianOK(self):
16         if self.bayesian is self.checks[-1].bayesian:
17             return 1
18         else:
19             return 0
20
21     def checkedOK(self):
22         if self.checked is len(self.checks):
23             return 1
24         else:
25             return 0
26
27     def changedOK(self):
28         changed = 0
29         for check in self.checks:
30             if check.changed is 1:
31                 changed = changed+1
32         if self.changed is changed:
33             return 1
34         else:
35             return 0
36
37     def lastChangedOK(self):
38         for check in self.checks[::-1]:
39             if check.changed is 1:
40                 ct = "%e" % check.time
41                 sl = "%e" % self.lastChanged
42                 if ct == sl:
43                     return 1
44         return 0
45
46     def lastCheckedOK(self):
47         if self.checks[-1].time == self.lastChecked:
48             return 1
```

```
49          else:
50              return 0
51
52      def getGroup(self):
53          max = 0
54          current = None
55          for item in self.bayesian:
56              if max < self.bayesian[item]:
57                  max = self.bayesian
58                  current = item
59          return current
60
61      def register(self,changed):
62          self.checked = self.checked+1
63          self.lastChecked = time.time()
64
65          if changed is 1:
66              self.changed = self.changed+1
67              days = (self.lastChecked-self.lastChanged)/3600/24
68              self.bayesian = estimate(days,self.bayesian)
69              self.lastChanged = self.lastChecked
70          checkItem = Check(self.lastChecked,self.checked,self.
                bayesian,changed)
71          self.checks.append(checkItem)
72
73      def printStat(self):
74          if self.checkedOK() is 1 and self.changedOK() is 1 and \
75              self.lastChangedOK() is 1 and self.lastCheckedOK()
                    is 1:
76                  print "URL:␣"+self.url
77                  print "CURRENT_BAYESIAN:␣",self.bayesian
78                  print "################################␣CHECKS␣
                        ###########################################"
79                  print "----------------------------------------"
80                  for check in self.checks:
81                      check.printStat()
82                      print "----------------------------------------"
83                  print "################################␣DONE␣
                        CHECKS␣#######################################"
84          else:
85              print "WTF??"
86
87  class Check:
88      def __init__(self,time,nr,bayesian,changed):
89          self.time = time
90          self.checkNR = nr
91          self.bayesian = bayesian
92          self.changed = changed
93
```

```
94      def printStat(self):
95          print "NR:␣",self.checkNR
96          print "TIME:␣",time.strftime("%d-%m-%Y-%H:%M:%S",time.
                localtime(self.time))
97          print "CHANGED:␣",self.changed
98          print "BAYESIAN:␣",self.bayesian
```

Listing 5.2: Kildekode for pageHandler modulen

```
1  import os
2  import sys
3  import cPickle
4  import threading
5  from page import *
6
7  ph = None
8  def handleEvent():
9      global ph
10     if not ph:
11         ph = PageHandler(1)
12     return ph
13
14 class PageHandler:
15     def __init__(self,active):
16         self.groups =     {
17                          "1.0":"URL_1",
18                          "3.0":"URL_3",
19                          "7.0":"URL_7"
20                           }
21         self.pages = []
22         prob = 1.0/len(self.groups)
23         stdBayesian = {}
24         for group in self.groups:
25             stdBayesian[float(group)] = prob
26         self.stdBayesian = stdBayesian
27         if active == 1:
28                 self.createDirectories()
29                 self.activateBackupDaemon()
30
31     def activateBackupDaemon(self):
32         self.bd = Daemon(self,60*60*24).start()
33
34     def savePages(self):
35         ts = time.strftime("%d_%m_%Y")
36         fileobj = open("/h/jasvik/master/backup/pages_"+ts,'w')
37         cPickle.dump(self.pages,fileobj)
38         fileobj.close()
39
40     def createDirectories(self):
41         try:
42             os.mkdir("/h/jasvik/master/backup")
43             os.mkdir("/h/jasvik/master/urifiles")
44         except:
45             #dirs allready exists
46             print "Error: backup- and/or uri- files allready
                   exists!"
47             #sys.exit(0)
```

```
48          for group in self.groups:
49              dirname = self.groups[group]
50              os.mkdir("/h/jasvik/master/urifiles/"+dirname)
51
52      def createURLFiles(self):
53          ts = time.strftime("%d_%m_%Y")
54          #ts = time.strftime("%d_%m_%Y-%H:%M:%S")
55          #ts = time.strftime("%d_%m_%Y-%H:%M")
56          for group in self.groups:
57              filename = self.groups[group]
58              fileobj = open("/h/jasvik/master/urifiles/"+filename
                    +"/"+ts+".txt",'a')
59              for page in self.pages:
60                  if page.getGroup() == float(group):
61                      fileobj.write(page.url+"\n")
62          fileobj.close()
63
64      def getPages(self,date):
65          #try:
66          fileobj = open("pages_"+date,'r')
67          self.pages = cPickle.load(fileobj)
68          #except:
69          #    print "No such file or directory"
70
71      def getPage(self,url):
72          for page in self.pages:
73              if page.url == url:
74                  return page
75          return None
76
77      def registerPage(self,url,changed):
78          url = url.strip()
79          p = self.getPage(url)
80          if p is not None:
81              p.register(changed)
82          else:
83              self.registerNewPage(url)
84
85      def registerNewPage(self,url):
86          p = Page(url,self.stdBayesian)
87          p.register(1)
88          self.pages.append(p)
89
90      def printPageStat(self,url):
91          p = self.getPage(url)
92          if p is not None:
93              p.printStat()
94          else:
95              print "ERROR: page not found"
```

```
96
97      def printStat(self):
98          print "STDBAYESIAN:␣",self.stdBayesian
99          for page in self.pages:
100                 page.printStat()
101
102 class Daemon(threading.Thread):
103     def __init__(self,arg,interval):
104         self.arg = arg
105         self.interval = interval
106         threading.Thread.__init__(self)
107         threading.Thread.setDaemon(self, 1)
108
109     def run(self):
110         while 1:
111             self.arg.savePages()
112             self.arg.createURLFiles()
113             time.sleep(self.interval)
```

Listing 5.3: Kildekode for bayesian modulen

```
import math

def estimate(days,bayesian):
        tmp = {}
        for item in bayesian.iteritems():
                tmp[item[0]] = calculate(days,item,bayesian)
        return tmp

def calculate(days,item,bayesian):
        DAYS = float(days)
        CAT = float(item[0])
        P = item[1]
        TELLER = (1-math.exp(-DAYS/CAT))*P
        NEVNER = 0.0
        for cat in bayesian.iteritems():
                NEVNER = float(NEVNER) + (1.0-math.exp(-DAYS/
                    float(cat[0])))*cat[1]
        return TELLER/NEVNER
```

5.3.3 Montering i Fast Crawleren

Fast Crawleren er som beskrevet tidligere en distribuert crawler. Den jobber asynkrynt som betyr at det ikke umiddelbart er intuitive steder i koden man kan kalle modulen. Crawleren består av et omfattende rammeverk for logging av utførte aksjoner. Her logges også hver nedlasting samt hvorvidt disse nedlastingene er forandringer eller duplikater av allerede nedlastede dokumenter. Naturlig sted å kalle modulen er når loggingen for nedlastingene blir utført. Her fantes det fullstendig informasjon om hvorvidt urlen var ny, duplikat, forandret eller slettet. Fordi loggingen blir utført så og si umiddelbart etter nedlasting er dette naturlig sted å kalle modulen. Urlen og status på siden blir sendt inn til modulen som lagrer et Check objekt til et allerede eksisterende Page objekt, eller oppretter et nytt Page objekt for lagring av sjekkingen.

Et annet viktig moment er at crawleren er distribuert som gjør at flere forskjellige prosesser kan kjøre samtidig i forskjellige python miljøer. Konkret betyr dette at flere forskjellige slave prosesser kan kjøre samtidig ved oppretting av flere forskjellige objekter. N antall slaveprosesser kan i bash shell for eksempel startes slik i en løkke:

- for i in 'seq 1 10'; do ./python slave.py

Eksekveringen fører til at de forskjellige slave prosessene som sagt kjører uavhengig av hverandre i forskjellige verdener. Sålenge det ikke implementeres kode for kommunikasjon mellom forskjellige python prosesser kan ikke disse prosessene dele de samme objektene realtime. Poenget er at det er slave prosessene som initierer loggingen, fordi det er slave sitt ansvar å laste ned dokumenter og rapportere resultat. Hvis man da velger å opprette instans av modulen for Bayesiansk Slutning i slave koden, så vil forskjellige prosesser lage hver sin datastruktur for hver av modulene. Slik som crawleren er distribuert vil også estimeringen bli distribuert uten at det er ønsket. Det er nødvendig å ha all informasjon angående hver url samlet i en og samme datastruktur, slik at hver estimering ved hver nedlasting blir korrekt.

I et praktisk tilfelle blir derimot ikke slave prosessene startet opp ved hjelp av kommando som antydet tidligere. Det blir av Uberslaven opprettet flere slaveobjekter under kjøring. Dette fører til at de individuelle slave objektene ikke kan kommunisere med hverandre. Derimot kan Uberslaven kommunisere med sine slave objekter, og hvert slave objekt kan kommunisere med sin Ubermaster. Det ideelle vil da være å opprette modulinstansen for Bayesiansk Slutning i Uberslaven for å unngå oppretting av en modulinstans per slave instans. Men da Fast

Crawleren er distribuert og opererer asynkront under nedlasting, finnes det flere kodelag som må tilpasses for at instanser opprettet i Uberslaven skal være tilgjengelig i slave instansene. Det er utifra dette en større kodejobb som må til for å implementere denne løsningen, som dermed blir utenfor rammen til denne oppgaven.

Problemet ble avdekket tidlig under testingen da forking av n+1 slave prosesser resulterte i n feilmeldinger når mappestrukturen i modulen skulle opprettes. Det eksisterte for n slaveforekomster allerede et mappetre som resulterte i feil. For å komme rundt problematikken på en enkel måte, ble crawleren startet opp med en spesifikk parameter slik at kun en slave prosess blir brukt. Crawleren ble startet opp med følgende kommando:

- ./python master.py -f /MasterCrawl.xml -c 1

Her spesifiserer det med -c 1 parameteren at det kun skal startes en forekomst av slave prosessen.

5.4 Eksperiment

5.4.1 Introduksjon

Selve eksperimentet ble utført på en av de virtuelle maskinene FAST har til disposisjon. På grunn av deling av maskineressurser som minne, prosessor og diskplass er de tilgjengelige maskinvareressursene vesentlige begrensninger i forhold til gjennomføringen og omfanget av eksperimentet. Oppstartskonfigurasjonen som bestemmer at det kun skal startes en slaveprosess, fører i tillegg til en flaskehals under kjøringen. Men på grunn av begrensede maskinvareressurser, vil ikke en enkelt slaveprosess føre til de store forsinkelsene. Uansett er ikke poenget å få crawlet flest mulig sider ved dette eksperimentet. Hovedmotivasjonen er å kunne følge utviklingsmønsteret for spesifikke sider, og kunne estimere tilhørende Bayesianske sannsynligheter for disse sidene. Deretter kan man analysere sannsynlighetene og se hvorvidt disse samsvarer med de faktiske forandringene. De begrensede maskinvareressursene og en enkelt slave prosess er derfor ikke ødeleggende for testgjennomføringen.

5.4.2 Konfigurering av FAST crawler

FAST Crawleren blir konfigurert til å operere på såkalte *kolleksjoner*. En kolleksjonsspesifikasjon inneholder informasjonen som er nødvendig

for at crawleren skal kunne gjennomføre det spesifiserte crawlet. FAST Crawleren har støtte for å operere på flere kolleksjoner samtidig. De aktuelle spesifikasjonene kan mates til crawleren på to forskjellige måter:

1. Ved bruk av et såkalt XMLRPC interface som er et Administrator interface for crawleren
2. Fra kommandolinjen når crawleren blir startet opp.

Crawleren startes typisk opp ved å gi kommando som indikert i Listing 5.4.

Listing 5.4: Oppstart av Crawler

```
./python master.py
```

Det finnes et omfattende sett med oppstartsparametere som tilpasser eksekveringen av crawlerapplikasjonen. Blant de følgende ble brukt under eksperimentet.

- -h Show usage information. Use this switch to print a list of and a short description of the various switches that are available.
- -v Enable verbose logging. Use this switch to make EC6 output additional information on its progress.
- -d <path> Data storage directory. Store crawl data, runtime configuration and logs in sub directories in the specified directory. Unless specified, EC6 will by default use the ¿data¿ directory in the current directory as storage directory.
- -f <file> Specify Collection Specification(s). Use this parameter to specify the location of a file containing one or more Collection Specifications. If specified, EC6 will read the contents of the file and start crawling the specified collections.
- -F <file> Specify crawler configuration file. Use this parameter to specify the location of a file containing EC6 configuration. A crawler configuration file is XML based and may contain default values for all commandline parameters. Note that no commandline switches may be specified in this configuration file. Also note that the Master process processes the commandline switches in-order.
- -c <num> Set the number of slave processes to start.
- -V Print version, then exit.

Under oppsettet for eksperimentet gjort med Bayesiansk Slutning var det viktig å gå gjennom alle parameterene for å få en korrekt eksekvering. Typisk fungerer det slik at crawler.py prosessen starter opp en eller flere slave prosesser dynamisk under kjøring. Dette fører til at for ett enkelt crawl kan flere forskjellige slave prosesser eksistere samtidig. På grunn av dette var det viktig å starte crawleren opp med konsekvente parametere for å få en konsistent test. Mer om dette under seksjonen 5.3.3.

Ved eksperimentet med Bayesiansk Slutning ble crawler-spesifikasjonen gitt ved hjelp av oppstartsparameteren -f og filnavnet til kolleksjons-spesifikasjonen. Denne så ut som følger:

CrawlerSpec

```
<?xml version="1.0"?>
<CrawlerConfig>
    <!-- Template -->
    <DomainSpecification name="MasterCrawl">
        <!-- String that can contain general-purpose information -->
        <attrib name="info" type="string">
            DEMO EC6 crawl specification for MasterThesis
        </attrib>

        <!-- Allowed MIME types -->
        <attrib name="allowed_types" type="list-string">
            <member>text/html</member>
            <member>text/plain</member>
        </attrib>

        <!-- Allowed schemes (http/https/ftp) -->
        <attrib name="allowed_schemes" type="list-string">
                <member> http </member>
        </attrib>

-
-
-
-
-
        <!-- Crawl Mode -->
        <section name="crawlmode">
            <!-- Crawl depth -->
            <attrib name="mode" type="string"> DEPTH:3 </attrib>
-
-
-
-
-
        </section>
```

```
        <!-- Delay between document fetches (in seconds) -->
        <attrib name="delay" type="real"> 60.0 </attrib>

        <!-- Delay between collection refresh (in minutes, 1440 is 24 hours) -->
        <attrib name="refresh" type="real"> 1440 </attrib>
        <attrib name="refresh_mode" type="string"> scratch </attrib>

        <!-- Use /robots.txt? ( Yes/no ) -->
        <attrib name="robots" type="boolean"> yes </attrib>

        <!-- List of start uris -->
        <attrib name="start_uris" type="list-string">
                <member> http://www.startsiden.no/ </member>
        </attrib>

-
-
-
-
-
        <!-- List of included domains (may be regexp, prefix, suffix and exact) -->
        <section name="include_domains">
            <attrib name="suffix" type="list-string">
                <member> .no </member>
                <member> .com </member>
            </attrib>
        </section>
-
-
-
-
-

    </DomainSpecification>
</CrawlerConfig>
```

Det er visse innstillinger som er verdt å legge merke til i spesifikasjonen:

- I «allowed_types» setter man typisk opp hvilke mimetyper man vil at crawleren skal akseptere. Som omtalt i 3.4.1 sendes det alltid en request header til den respektive webtjeneren når en side eller annen ressurs skal lastes ned. Denne headeren inneholder all tilstrekkelig informasjon som er nødvendig for webtjeneren å returnere riktig type innhold. Ved konsekvent å definere elementer i «allowed_types» kan man sette hvilke mimetyper man ønsker at crawleren skal spørre etter. Den forespørrende headeren som

crawleren sender til de ulike webtjenerene vil da typisk inneholde tilsvarende elementer som i «allowed_types» seksjonen. Fordi vi er interessert i html sider i dette adaptive crawler eksperimentet, settes elementene til dette. Når crawleren så får et dokument tilbake, sjekkes tilsvarende felt i respons headeren. På denne måten kan crawleren utelate å laste ned innhold av uønsket type.

- Nok en viktig innstilling for eksperimentet er «mode» elementet. Her kan man definere hvor mange nivåer crawleren skal følge av linker med start-URLen som utgangspunkt. I forsøket ble denne satt til 3 nivåer. Dette får å få flest mulig langvarige sider som ikke dynamisk eksisterer og forsvinner. For at vi skal kunne lagre historikk og utføre estimering som tidligere indikert, er vi avhengig av at de ulike sidene ikke forsvinner og eksisterer under mesteparten av eksperimentet.

- Som omtalt tidligere er en høflig crawlingtaktikk nødvendig for å sikre at ressursene alltid skal være tilgjengelig. For hyppig crawling risikerer utestengelse fra websteder. Under «delay» defineres hvor lang tid crawleren må vente før den spør en og samme webtjener. Så hvis crawleren har sidene http://www.vg.no/ og http://www.vg.no/sport på toppen av crawlerkøen, så må crawleren vente i 60 sekunder (som definert i spesifikasjonen) mellom hver av urlene.

- I tillegg må crawleren vite hvor lang tid det skal ta før den skal oppdatere kolleksjonen. Det betyr typisk at crawleren sletter det den har av eventuelle ikkecrawlede urler i køen, og starter på nytt med utgangspunkt i start-URLene som er definert under «start_uris». Definisjonen av denne recrawlingssyklusen er satt opp ved hjelp av «refresh» elementet i spesifikasjonen. Kolleksjonen i eksperimentet blir som spesifisert recrawlet hver dag. Årsaken til at det er valgt å recrawle relativt hyppig er for å detektere flest mulig forandringer per element for å gi et mest mulig korrekt adaptivt estimat.

- Urlen http://www.startsiden.no/ er satt opp som eneste start-URL i konfigurasjonen. Dette er gjort helt enkelt for å sikre at flest mulig av de crawlede elementene ikke skal forsvinne eller oppstå underveis. Det tar en god del tid for at estimatene skal bli korrekte, og da er det viktig at elementene som inngår eksisterer under mesteparten av eksperimentet.

5.4.3 Gjennomføring

Selve eksperimentet består av flere mindre deleksperimenter. I dennes seksjonen forklares hvorfor det var nødvendig med flere mindre deleksperimenter, samt hvordan resultatene på de individuelle gjennomføringene påvirket de neste. Det var også nødvendig med tilpasninger underveis for optimale testresultater.

Det ble gjort flere individuelle gjennomkjøringer med forskjellige konfigurasjoner for å se hvilke påvirkninger dette hadde på estimeringsresultatene. Til å begynne med inngikk helt vilkårlige start-URLer i kolleksjonsspesifikasjonen. Urlene ble tatt fra et eksisterende crawl som blir gjort for mobilt innhold på web, henholdsvis wml og mobile xhtml.

Grunnen er at spesielt for websider med mobilt innhold, er hovedvekten av dynamiske sider som egner seg best til testing av adaptive metoder. Årsaken til dette er tett knyttet til brukere og mobiltelefonene som har støtte for å vise innholdet. WAP protokollen som gjør det mulig å vise mobilt innhold på internett, ble lansert i 1998. Hendelsen ble i 1999 tett fulgt opp av nye mobiltelefoner med støtte for å vise det mobile internettet. Selvom teknologien nå har eksistert i over 15 år, er antallet brukere av WAP temmelig beskjedent i forhold til brukere av World Wide Web. Dette skyldes hovedsakelig brukeropplevelsen. Mobiltelefoner med vanskeligstilte taster, lav båndbredde, små skjermer uten farger og uegnet programvare er alle viktige årsaker til en ikke-tilfredsstillende brukeropplevelse. Det er etterhvert gjort flere forsøk på å utvide den beskjedne brukergruppen. Blant annet å tilby tidsaktuell informasjon som nyheter, vær, ruteopplysning med mer. Hovedvekten av mobile websider på internett er derfor av dynamisk art.

Problemet med å bruke mobile internettsider under eksperimentet, var at 100% av disse sidene forandret seg hver eneste gang. Resultatet ble altså at ved de daglige fildumpene for hver kategori, var samtlige filer for kategoriene 2 og 3 [1] uten innhold. Alle urlene ble logget i de daglige fildumpene for kategori 1 [2]. Ifølge oppsettet skal altså disse sidene ha forandret seg en eller flere ganger hver dag, i og med at syklusen var begrenset til 24 timer. Med testoppsett som i vårt tilfelle var vi interessert i å dele sidene inn i kategorier på henholdsvis daglige, hver 3. dag og hver 7. dag. Ved et slikt oppsett er det da meningsløst å utføre eksperimentet på sider som forandrer seg hver dag. Vi vil få verifisert at metoden fungerer for den daglige kategorien, men ikke for kategoriene hver 3. dag og hver 7. dag. Mobile websider egnet seg derfor ikke til å teste den Bayesianske

[1] hver 3. dag og hver 7. dag

[2] hver 1. dag

modulen.

Etterhvert ble start-URLene sortert slik at mer statiske sider ble crawlet for å kunne dele urlene mer jevnt i de forskjellige kategoriene. Siste redefinisjon av start-URLer ble gjort til http://www.startsiden.no/. Innholdet i kolleksjonen ble da bestående av sider som utelukkende var linket til fra http://www.startsiden.no/. Testen fikk stå og gå i overkant av en måned. Testing av adaptive metoder vil nærme seg riktige verdier desto lenger de får kjøre. Det er derfor viktig å la eksperimenter kjøre i lengere perioder for å verifisere at metodene virkelig fungerer.

Etter eksperimentet hadde fått kjøre i overkant av en måned ble det gjort et overraskende funn. Selv sider som er linket til av http://www.startsiden.no/ forandret seg tilnærmet daglig. På dette stadiet av eksperimentet var det klart at den Bayesianske metoden fungerer for elementer som forandrer seg daglig. Dette kan begrunnes av resultatene fra crawlingen av mobile internettsider hvor sidene endret seg daglig. Urlene for de ulike sidene ble daglig dumpet i den korrekte kategorien [3]. Men det var ikke klart hvorvidt metoden fungerer for andre kategorier. Ved crawlingen av http://www.startsiden.no/ kunne vi derfor se bort ifra sider som endret seg hver dag. For å bekrefte eller avkrefte metoden for de andre kategoriene, var det tilstrekkelig å fokusere på sidene som ikke endret seg hver dag. For noen få sider som tilsynelatende fungerer som hubsider, ble det gjort deteksjoner hvor sidene ikke forandret seg hver dag. Tilsammen ble det under eksperimentet crawlet i overkant av 1 million dokumenter tilsvarende over 6 gigabyte med innhold. Av disse ble det trukket ut følgende informasjon om sidene som ikke forandret seg daglig:

- Antall sider som en eller flere ganger ikke var forandret: 202775
- Antall sider av disse som ikke er med i kategori for daglige oppdateringer: 0

Av tallene ovenfor ser vi at sannsynlighetsverdiene IKKE gjenspeiler endringsmønsteret for sidene. Selvom sidene ikke forandret seg hver eneste dag, blir den estimerte sannsynligheten likevel beregnet slik at elementene blir satt i kategorien for daglige oppdateringer. For eksempelsiden http://www.startsiden.no/ ser de første 12 oppdateringene ut som på Listing 5.5, printet ut ved hjelp av printStat() funksjonen.

Listing 5.5: Bayesiansk oversikt for startsiden.no

[3] den daglige kategorien

```
1  NR:   1
2  TIME:   05−07−2006−23:36:47
3  CHANGED:   1
4  BAYESIAN:   {
5  1.0:  0.67741924016253285,
6  3.0:  0.22580641338751095,
7  7.0:  0.096774346449956208}
8  ————————————————————————————————————————
9  NR:   2
10 TIME:   06−07−2006−23:37:58
11 CHANGED:   1
12 BAYESIAN:   {
13 1.0:  0.84774046346050402,
14 3.0:  0.12674775623466952,
15 7.0:  0.025511780304826433}
16 ————————————————————————————————————————
17 NR:   3
18 TIME:   08−07−2006−00:08:14
19 CHANGED:   1
20 BAYESIAN:   {
21 1.0:  0.93127327371666091,
22 3.0:  0.062781395371254059,
23 7.0:  0.0059453309120849193}
24 ————————————————————————————————————————
25 NR:   4
26 TIME:   09−07−2006−23:53:47
27 CHANGED:   1
28 BAYESIAN:
29 {1.0:  0.96182279974849338,
30 3.0:  0.036417205810555547,
31 7.0:  0.0017599944409510367}
32 ————————————————————————————————————————
33 NR:   5
34 TIME:   11−07−2006−00:24:59
35 CHANGED:   1
36 BAYESIAN:   {
37 1.0:  0.9828364229712323,
38 3.0:  0.016781969380230462,
39 7.0:  0.00038160764853725207}
40 ————————————————————————————————————————
41 NR:   6
42 TIME:   12−07−2006−04:18:49
43 CHANGED:   1
44 BAYESIAN:   {
45 1.0:  0.99199733675229218,
46 3.0:  0.0079169166781540394,
47 7.0:  8.5746569553716746e−05}
48 ————————————————————————————————————————
49 NR:   7
```

```
50 TIME:    12-07-2006-23:03:45
51 CHANGED:    1
52 BAYESIAN:    {
53  1.0:  0.99661978592321976,
54  3.0:  0.0033634346363998228,
55  7.0:  1.6779440380431655e-05}
56 ------------------------------------------------------------
57 NR:    8
58 TIME:    13-07-2006-23:54:17
59 CHANGED:    1
60 BAYESIAN:    {
61  1.0:  0.99847149309515093,
62  3.0:  0.0015249232818868579,
63  7.0:  3.583622962158432e-06}
64 ------------------------------------------------------------
65 NR:    9
66 TIME:    14-07-2006-19:22:50
67 CHANGED:    1
68 BAYESIAN:    {
69  1.0:  0.99934848256640973,
70  3.0:  0.00065081105628714557,
71  7.0:  7.0637730306514961e-07}
72 ------------------------------------------------------------
73 NR:    10
74 TIME:    15-07-2006-23:56:06
75 CHANGED:    1
76 BAYESIAN:    {
77  1.0:  0.99969347307998546,
78  3.0:  0.00030636816056592547,
79  7.0:  1.5875944853690934e-07}
80 ------------------------------------------------------------
81 NR:    11
82 TIME:    16-07-2006-16:43:05
83 CHANGED:    1
84 BAYESIAN:    {
85  1.0:  0.99987332071517876,
86  3.0:  0.00012664927639139837,
87  7.0:  3.0008429816963953e-08}
88 ------------------------------------------------------------
89 NR:    12
90 TIME:    18-07-2006-01:00:14
91 CHANGED:    1
92 BAYESIAN:    {
93  1.0:  0.99993810307146791,
94  3.0:  6.1889833408657329e-05,
95  7.0:  7.0951234245777086e-09}
```

Man ser av listen at siden endret seg hver eneste gang selvom den tilsynelatende skulle virke som en statisk side. Dette mest sannsynlig på

grunn av en veldig streng duplikatdeteksjon. Men som tallende ovenfor indikerer var det heller ingen statiske sider som ble estimert til annet enn daglige oppdateringer.

Utifra disse resultatene ble det gjort en utvidet test av den Bayesianske metoden. Under denne testen ble det simulert oppdateringer hvor modulen ble kalt og verdier estimert. Verdiene ble estimert slik at 100 oppdateringer ble simulert og tilhørende sannsynligheter ble beregnet. For alle oppdateringene ble det satt et statisk dagsintervall n. For n=100 betyr det at det ble simulert 100 oppdateringer hvor det gikk 100 dager mellom hver detektert forandring. Det ble simulert sannsynligheter for vårt eksperiment, nemlig 3 kategorier på intervallene daglige (d=1), hver 3. dag (d=3) og hver 7. dag (d=7) hvor n = 1, 5 og 10. N=1 betyr da at det ble detektert daglige forandringer, og sannsynlighetsverdiene for hver kategori etter 100 forandringer blir registrert. Tilsvarende for n=5 at det ble detektert forandringer hver 5. dag, og sannsynlighetsverdiene for hver kategori etter 100 endringer blir registrert også videre. Testingen resulterte i følgende verdier:

n	$d = 1$	$d = 3$	$d = 7$
1	1.0	$1.481e^{-35}$	$2.213e^{-68}$
5	0.999	$1.594e^{-9}$	$1.229e^{-29}$
10	0.974	0.026	$1.236e^{-12}$

For samtlige simuleringer ser vi at elementet likevel blir estimert med høyest sannsynlighet å tilhøre kategori 1, med elementer som burde oppdateres daglig. Vi ser av tallene at en side endrer seg hver 10. dag, som for n=10, og holder dette konstante endringsmønsteret i 100 forandringer. Da skulle det virke naturlig å plassere dette elementet i kategori 3 og ikke kategori 1. Etter disse resultatene var det naturlig i tillegg å simulere en ytterkant situasjon for virkelig å kunne avkrefte metoden med Bayesiansk slutning. Følgende simulering ble gjort:

- antall forandringer ble satt til 1000 hvor sidene forandret seg med 100 dagers mellomrom. Kategoriene ble satt til henholdsvis hver 1. dag, hver 30. dag og hver 90. dag.

Kategoriene ble spredd over lengre perioder for å gi metoden bredere intervaller å operere på. Testingen resulterte i følgende verdier:

n	$d = 1$	$d = 3$	$d = 7$
1000	0.999	$3.959e^{-174}$	$1.674e^{-16}$

Resultatet tilsier at dokumentet burde med 99% sannsynlighet oppdateres

hver dag hvis elementet har forandret seg 1000 ganger og hver gang med 100 dagers mellomrom. Dette virker som alt annet enn en fornuftig taktikk.

5.4.4 Bayesiansk Slutning. Teori vs. Praksis

I teorien virker metoden for estimering av forandringsrater ved hjelp av Bayesiansk Slutning som en fornuftig og riktig metode for å kategorisere sider. Men etter det praktiske eksperimentet og testene som ble gjort, er det klart at metoden ikke er brukbar i en reell situasjon. Resultatene fra eksperimentet samsvarer ikke med det som virker som et intuitivt oppdateringsintervall. Det er to sentrale aspekter som må vurderes men også skilles. For det første så er det viktig at metoden ikke korrekt klarer å beregne intervaller i praktiske situasjoner. Men nok et viktig poeng er at hvis en forutsetter at metoden hadde fungert tilfredsstillende, så hadde man i en praktisk situasjon likevel hatt problemer.

La oss si vi forutsetter at den Bayesianske modulen virker som den skal. Det vil si at ved simuleringene så beregner den korrekte sannsynligheter, og plasseres elementene korrekt i de forskjellige kategoriene ved hjelp av de estimerte sannsynlighetene. Til tross for dette, så vil et tilbakeblikk på tallene fra eksperimentet fort avsløre problemer. Dette er problemer som ikke er direkte knyttet til den Bayesianske modulen, men metoden for å detektere forandringer og duplikater. Når toppdomenesider som http://www.startsiden.no/ blir detektert til å forandre seg daglig, er terskelen for akseptable forandringer altfor lav til å kunne benytte adaptive metoder for å effektivisere crawlingrutiner. Sålenge man ikke er istand til å klassifere to versjoner som identiske, hvor forskjellen kun er et datostempel, blir det praktisk tilnærmet umulig å gjøre fornuftig adaptive forbedringer i en crawlerkontekst. Datoen trenger i virkeligheten heller ikke være visuelt synlig. Enkel duplikatdeteksjon vil likevel beregne 2 visuelle identiske sider som forskjellig sålenge noe usynlig innhold som kommentarer eller lignende er forskjellig. En viktig forutsetning for å kunne gjøre en fornuftig adaptiv tilnærming for crawlere er dermed en robust duplikatdeteksjon.

5.5 Oppsummering

FAST-Crawleren er modulært bygget opp og har en arkitektur som gjør den enkel å tilpasse en multinodeinstallasjon. Det er derimot omfattende å legge til egenprodusert logikk for enkelt å kunne teste ut

ny funksjonalitet og algoritmer. En viktig positiv egenskap er at den har mange konfigurasjonsparametere som likevel gjør det til en smal sak å kalle egenproduserte biblioteker.

Modulen for Bayesiansk Slutning er implementert i python for å gjøre den enkel å integrere med FAST-Crawleren. Den baserer seg på å lagre endringshistorikk i en enkel datastruktur, for å gjøre det mulig å beregne kategoriserte sannsynligheter til enhver tid. Selvom teorien tilsier at metoden er en fornuftig adaptiv tilnærming for webcrawlere, strider praktiske eksperimenter og simuleringer mot dette. Resultatene samsvarer ikke med intuitive forandringsrater, og er i en praktisk situasjon vanskelig å benytte seg riktig av.

I tillegg til modulen er deteksjon av duplikater en viktig forutsetning for å gjøre en fordelaktig adaptiv implementasjon for crawlere. Det burde derfor være et mer prioritert satsingsområde å utvikle en avansert men likevel robust metode for duplikatdeteksjon. Har man en riktig og konfigurerbar metode for å detektere duplikater, gir det mening å satse på å utvikle adaptive forbedringer for crawlere. I motsatt fall blir det vanskelig å gjøre riktig bruk av en fungerende adaptiv metode da det er en forutsetning intelligent å kunne identifisere duplikater.

Kapittel 6

Modulær crawlerarkitektur

6.1 Introduksjon

Etter å ha brukt mye tid på teori og forskningsarbeider rundt metoder, er det interessant å se resultater av eksperimenter på praktisk bruk. Metoden fra implementasjonen i forrige kapittel er kun ett av flere aktiviteter rundt Adaptiv crawling. Flere av disse er tidligere omtalt i kapittel 4. Erfaringen fra forrige kapittel tilsier at gjennomføring av tilsvarende prosjekter stiller høye krav til crawleren med tanke på fleksibilitet og mulighet for adoptering av funksjonalitet. En omfattende arbeidsmengde var en forutsetning for å kunne gjøre de forskjellige tilpasningene som var nødvendig for eksperimentet. Spesielt når eksperimenter ikke gir tilfredsstillende resultater, er det viktig å kunne jobbe med en crawleraktitektur som gjør den minimalt tidkrevende å tilpasse.

I dette kapittelet tas den nevnte problematikken opp med paralleller til erfaringer fra det gjennomførte eksperimentet og tidligere teori 3, 3.9. På grunnlag av dette blir det utformet en serie med krav ti len crawler. For senere kunne gjøre en tilfredsstillende adaptiv tilnærming for webcrawlere er det viktig å ha en konfigurerbar og tilpasningsdyktig platform. Sentrale punkter for en vellyket crawler blir tatt opp i forhold til de nevnte kravene. En crawlerarkitektur blir foreslått som gjør utvikling og testing av adaptive crawlertilpasninger til en enkel gjennomførbar rutineaktivitet.

6.2 Krav til Crawlerarkitekturen

Oppdateringer av eksisterende programvare og applikasjoner er en forutsetning innenfor databransjen. Det stiller store krav til ressurser

som personer, kompetanse og tid. I forhold til utviklere er dette sentrale ressurser som man må forholde seg til. Nyere forskning og stadige forbedringer av algoritmer og metoder gjør det nødvendig for bedrifter å allokere ressurser til aktivitet rundt dette. For virksomheter som baserer seg på utvikling av programvare er det en forutsetning å måtte delta i forsnkningskappløpet og tilby oppdatert teknologi til enhver tid. Men for bedrifter er det en viktig balansegang å allokere tid til generell utvikling eller utvikling dedikert til forskning. Derfor er det viktig å gjøre forskningsdedikert utvikling til en mest mulig rutinert aktivitet.

Som erfaringen fra crawlerutvikling tilsier, er det viktig å kunne gjøre eksperimenter på forskningsarbeid effektiv slik at begrensede utviklerressurser blir bespart. Prosessen å sette seg inn i ny teori og forskning er en aktivitet som er en viktig innledende aktivitet for eventuell implementering. Det er viktig å gjøre en solid jobb under denne fasen slik at man unngår feil i forståelsen av selve forskningsarbeidet. Implementasjonen derimot er en fase hvor mye tid kan bespares hvis man utformer en platform med tanke på å kunne utvide den senere. En crawlerarkitektur som er typisk pluggbar med tanke på funksjonalitet, vil være mindre tidkrevende å tilpasse enn en flat implementasjon hvor det er nødvendig med omfattende kodeedringer for å utvide funksjonaliteten.

6.2.1 Adaptive crawlertilpasninger

Etter arbeidet med å implementere modulen for Bayesiansk Slutning som beskrevet i kapittel 5, er det flere aspekter ved en crawler som kunne vært naturlig å tilpasse for å gjøre lignende eksperimenter mindre tidkrevende. Det kunne tenkes at et spesifikt eksperiment gav veldig gode resultater, og man ikke hadde umiddelbare behov for å utføre flere eksperimenter rundt adaptive metoder i nærmeste fremtid. I slike tilfeller kan det være akseptabelt at det er en tidkrevende jobb å utføre lignende eksperimenter. Men som i vårt tilfelle er det ikke sikkert at eksperimentet gir gode nok resultater. Man må da enten være istand til å avsløre i en tidlig fase at metoden ikke vil fungere, eller ha en enkelt tilpasselig crawlerarkitektur som muliggjør en tilpasset implementasjon i løpet av kort tid. En konfigurerbar crawlerarkitektur vil være optimal for å kunne teste flere alternative adaptive metoder 4, hvis en spesifikk metode ikke gir gode nok resultater.

Som den tidligere dokumenterte optimale ressursallogeringsmetoden 4.6, fant vi i en tidligere fase ut at algoritmen ville bli problematisk i en praktisk implementasjon. Løsning av et likningssystem med N+1

likninger og N+1 ukjente hvor N er antall websider i kolleksjonen, ville bli for altfor tidkrevende å utføre ved oppdatering av hvert eneste element. Se seksjon 4.6 for mer informasjon. Poenget er uansett at vi tidlig klarte å forutse at metoden ikke ville være praktisk i et reellt tilfelle. Vi sparte dermed tiden det ville ta oss å gjøre en eventuell implementasjon for så å finne ut av at metoden ikke gav gode nok resultater.

Et annet scenario er eksperimentet vi praktisk utførte med metoden for den Bayesianske Slutningen 5. Ved dette tilfellet var vi ikke istand til å forutse de faktisk resultatene og at de ikke ville være tilredsstillende. I en slik situasjon hadde det vært fordelaktig å arbeide med en pluggbar crawlerarkitektur, som gjorde det mulig å teste ut funksjonaliteten med minimal programmeringsaktivitet og i løpet av begrenset tid. Hvis det eksisterer et generisk rammeverk i crawleren hvor man kan plugge inn logikk som implementerer det generiske rammeverket, vil det være forhodsvis lite tidkrevende å teste ut alternativ logikk. Selve tiden det tar å implementere nye algoritmer og funksjonalitet vil oppta størsteparten av den totale tiden som blir brukt. Tilpassingen og kodeendringer i forhold til en eventuell crawler vil ta opp minimalt med tid i og med at et rammeverk for dette eksisterer. I tillegg vil det være mindre behov for gjennomgående og detaljert systemdokumentasjon for hele crawlerarkitekturen, samtidig som en god dokumentasjon av rammeverket vil være tilstrekkelig for å gjennomføre en adaptiv algoritmetest. Årsaken til at en detaljert systemdokumentasjon ikke vil være nødvendig for vanlige tilpasninger, er at slike tilpasninger vil være mulig å utføre ved hjelp av det generiske rammeverket. Det vil derfor holde å sette seg inn i en eventuell dokumentasjon av rammeverket.

6.2.2 Duplikatdeteksjon i en tilpasset crawler

Duplikatdeteksjon er en av de mest sentralte aktivitetene crawleren utfører. Som oppsummert i kapittel 5 er det en viktig forutsetning at crawleren intelligent klarer å avgjøre hvilke elementer som er duplikater og ikke, for å gjøre en fordelaktig adaptiv implementasjon for crawleren. Slik som crawlere flest opererer idag baserer de duplikatdeteksjonen på sjekksum-beregninger (se 3.7 for mer informasjon) og enkel sammenligning av disse. Av eksperimentet kunne vi se at slik duplikatdeteksjon blir for naiv til å benytte seg av i sammenheng med adaptive metoder. Det er viktig å kunne implementere en metode hvor man selv kan velge hvor stor andel av siden som kan være forsjellig for at elementene likevel skal klassifisereres som duplikater. En slik vektet

duplikatalgoritme vil gjøre det lettere å kunne benytte seg av eventuelle fungerende adaptive algoritmer.

I seksjonen 3.7.1 ble det foreslått en metode for å detektere duplikater i en crawler ved hjelp av RSYNC. Metoden ble her presentert i en tjener-klient kontekst hvor man unngår å laste ned innhold som man allerede har lastet ned. Tilsvarende som for en generelle RSYNC installasjoner:

- Klienten populerer et filtre i henhold til spesifikasjonen og data som blir lastet ned fra tjeneren. Ved hver oppdatering overføres kun forskjellene i de konsekvente filene som er endret, med tilhørende informasjon om hvilke filer og hvor i filen forskjellen ligger.

Problemet med å implementere tilsvarende tjener-klient kontekst i en crawler er at det stiller krav til endringer i webtjenerrutinene for behandling av crawlerforespørsler. Webtjeneren må implementere tjenersiden av RSYNC som ikke er enkelt å få til med alle forskjellige webtjenerinstallasjoner som finnes idag.

Men det finnes likevel en alternativ metode basert på RSYNC som kan brukes til duplikatdeteksjon i crawlerapplikasjoner. Selve algoritmen fungerer som følgende [3]:

- Vi forutsetter at vi har to forskjellige noder α og β. Noden α har aksess til filen A og β har aksess til filen B, hvor A og B regnes som like filer. RSYNC algoritmen består av følgende steg:

  1. β splitter filen B i en serie av disjunkte blokker av fast størrelse på S byte. Den siste blokken kan være mindre enn S byte.
  2. For hver av disse blokkene beregner β to sjekksummer: en ssvak"32-bit sjekksum og en sterk"128-bit MD4 sjekksum.
  3. β sender så disse sjekksummene til α.
  4. α søker gjennom A for å finne alle blokker av størrelse S byte [1], som har den samme svake"og sterkesjekksummen som en av blokkene i B. Dette kan utføres raskt i en enkel løkke ved å benytte seg av en spesiell egenskap i den svakesjekksummen [3].
  5. α sender til β en sekvens av intruksjoner for for å rekonstruere en kopi av A. Hver intruksjon er enten en referanse til en blokk i B, eller rådata. Rådata blir kun sendt for blokkene i A hvor det ikke fantes noe treff for de genererte sjekksummene i B.

[1]Fra hvilket som helst sted i fila. Ikke kun faktorer av S

Sluttresultatet er at β får en kopi av A, men kun de delene av A som ikke ble funnet i B. I tillegg kommer en liten mengde med data for sjekksummene og blokk indekser. Algoritmen krever kun en behandling av sjekksummene som gjør at det holder å sende informasjon parvis kun en gang. Først sender β til α før rekonstruksjonsinformasjon sendes fra α til β.

Det er selvfølgelig mulig å benytte seg av denne formen for duplikatdeteksjon lokalt istedetfor. Man kunne tenke seg å ekstrahere algoritmen som beskrevet ovenfor og implementere denne i en crawler. Man kunne konfigurere crawleren til å operere på 2 adskilte databeholdere. En beholder B som inneholder alle websidene etter duplikatdeteksjon og en tilsvarende beholder A med websidene før duplikatdeteksjon. B kan så implementere klientsiden av RSYNC algoritmen mens A kan implementere tjenersiden. Internt kan da crawleren opprette flere instanser av klient-tjener forbindelser, og bruke forskjellene i sjekksummene til å beregne en vektet verdi for forskjellen mellom ulike websideversjoner.

En mer elegant løsning vil selvølgelig være å implementere en RSYNC modul som for to dokumenter beregner en vektet verdi for forskjellen. Ved å ekstrahere algoritmen med sjekksummer fra RSYNC og lage en algoritme som på grunnlag av disse beregner et forhold for hvor mye som er forskjellig, kan man la crawleren benytte seg direkte av denne modulen og unngå avhengigheter av tredjeparts programkode. Men å la crawleren benytte seg av en slik modul, setter krav til muligheter for å montere moduler i crawleren. Hvis crawleren utad tilbyr et generelt rammeverk for plugging av eksterne moduler, er det tilstrekkelig å la RSYNC-modulen implementere dette rammeverket for å la crawleren benytte seg av funksjonaliteten. Forutsetningen er uansett muligheten for å kunne plugge moduler i crawleren. Hvis ikke vil det mest sannsynlig være en omfattende og mer tidkrevende strukturendring som må til for å muligjøre duplikatdetektering ved hjelp av RSYNC.

6.2.3 Nedlasting i en tilpasset crawler

Under en tidligere seksjon 3.4 nevnes såvidt utfordringene rundt behandling av forespørrende headere på webtjenersiden. Man kan i flere tilfeller ha en korrekt konfigurert forespørsel men få feil resultattyper tilbake som respons, fordi logikken som behandler forespørselen på tjeneren ikke er riktig implementert eller konfigurert. I et hav av forskjellige webtjernerimplementasjoner er det viktig for crawleren å

kunne «lære» seg og «teste» ut flere forskjellige forespørsler som gir det korrekte innholdet i riktig format. Typisk kan det være at en spesiell «User-Agent» vil gi innhold kun av en type uansett om forespørselen virkelig ber om innhold av en annen type. For crawleren å kunne «teste» ut forskjellige «User-Agents» som gir dokument i ønsket format, er en høyt verdsatt egenskap for et mest mulig komplett crawlingresultat.

Intuitivt ser vi for oss en global liste med «Accept» mimetyper som inneholder alle riktige formater man ønsker å ha i kolleksjonen. Tilsvarende en global liste med «User-Agent» som inneholder beskrivelser av nettlesere for det riktige formatet av innhold man er interessert i. Deretter kan crawleren intelligent gjøre utplukk og generere forespørsler på grunnlag av tidligere erfaringer helt til ønsket format blir returnert av webtjeneren. Den vellykkede kombinasjonen kan deretter lagres for senere bruk og eventuell oppdatering. Praktisk er det en tidkrevende og omfattende jobb å utføre analyser av enkelturler for å finne korrekte forespørsler. Med korrekt forespørsel menes en kombinasjon av felt som gir innhold i et ønsket format. I verste fall kan man risikere at innhold ikke blir crawlet i det hele tatt fordi innholdet ikke returneres i ønsket format. Det er derfor viktig å sjekke i hvilket format innholdet blir returnert, men en manuell gjennomgang av samtlige urler i et crawl er altfor tidkrevende og lite praktisk. En automatisert algoritme for å gjøre intelligente utplukk er derfor ønskelig.

Men også en eventuell kodeendring for å gjøre en tilpasset nedlasting, kan i mange tilfeller være altfor omfattende hvis arkitekturen ikke er planlagt for det. Det enkleste vil også i dette tilfellet være å implementere funksjonaliteten som en modul, og tilby i crawleren et generisk rammeverk som modulen kan implementere. På denne måten kan man også legge til andre algoritmer som relaterer til tilpasset nedlasting i crawleren. Forutsetningen er igjen en arkitektur som gjør det mulig å plugge moduler som implmenterer et rammeverk som crawleren tilbyr.

6.3 Forslag til Crawlerarktitektur

Tidligere seksjoner indikerer at en konfigurerbar crawlerarkitektur er fordelaktig for effektivt kunne videreutvikle funksjonalitet og adaptive egenskaper. For å kunne utføre tilpasset nedlasting, duplikatdeteksjon ved hjelp av RSYNC og selektiv crawling basert på adaptive metoder setter det spesifikke krav til crawlerarkitekturen. Men de nevnte aktivitetene skjer på vidt forskjellige stadier i løpet av perioden når crawleren behandler et dokument. Aktivitetene kan deles inn i tre kategorier:

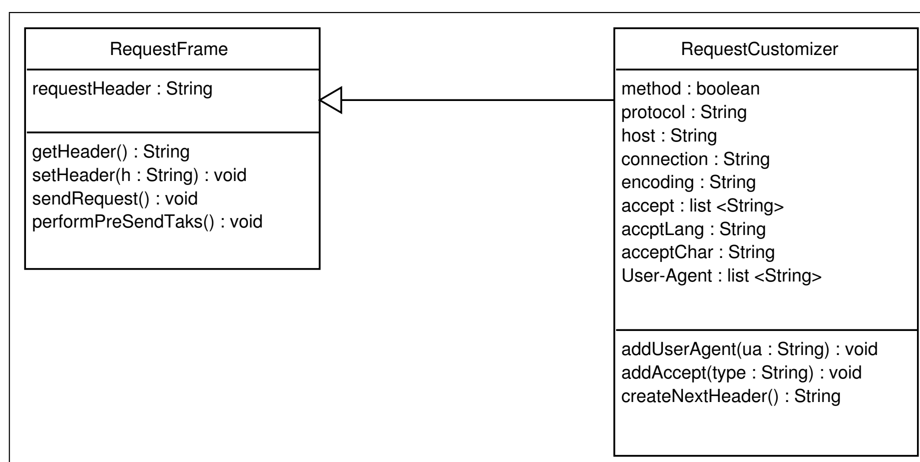
1. Behandling av HTTP forespørsel
2. Behandling av HTTP svar
3. Behandling av prioritert crawlerkø

Punkt 1 og punkt 2 går typisk på behandling av data som sendes til og mottas fra tjeneren. Under punkt 1 er det naturlig å utføre tilpasningen av HTTP forespørselen med tanke på mimetyper og «User-Agent» for å løse problemene rundt feilreturnerte typer. Som en del av punkt 2 vil man typisk utføre duplikatdeteksjonen da hele dokumentet som lastes ned fra tjeneren sendes i HTTP svaret. Ved punkt 3 vil man utføre typiske adaptive aktiviteter som gjør seleksjoner og behandler køen over urler som skal crawles.

Poenget er at det ved disse tre forskjellige stadiene i crawleren er behov for å kunne plugge moduler som utfører tilpasset logikk. Det må typisk lages et rammeverk for hvert av punktene som gjør det mulig å implementere moduler med spesiallogikk som blir utført på variabler og data tillatt av rammeverket. I punkt 1 vil det være naturlig at rammeverket tillatter modulene å operere på selve forespørselen, i punkt 3 burde selve dokumentet som er lastet ned være tilgjengelig og for punkt 3 burde tilhørende rammeverk tillatte manipulering av crawlerkøen.

Implementasjonen av rammeverkene vil føre til at riktige implementasjoner i modulene blir kalt til riktig tid. For at det skal fungere er det intuitivt et krav til modulene at de implementerer disse funksjonene riktig i henhold til rammespesifikasjonen samt returnerer korrekt data i riktig form og navn. På denne måten kan crawleren la eksterne moduler utføre spesiallogikk som omtalt tidligere i kapittelet uten at kode sentralt i kjernen må gjøres om. Selve crawlerlogikken forblir statisk mens modulene vil kunne utvides til ønskelig funksjonalitet sålenge rammeverket tillatter utførelsen.

Designet og en eksempelvis implementasjon av et rammeverk for 1. Behandling av HTTP forespørsler kan se ut som indikert i Figur 6.1. Her er det skissert et meget enkelt rammeverk for henting og setting av forespørselen. Typisk vil kjernen i crawleren kalle performPreSendTaks() for så å kall sendRequest() funksjonen. Typisk vil da modulene hvis nødvendig redefinere performPreSendTasks() funksjonen og utføre get- og set- på forespørselen for å lage nye forespørsler for de ulike «accept» verdiene i listen. Algoritmene er opptil modulen å definere. Modulen har ansvaret for å sette forespørselen som en del av perfromPreSendTasks(). Deretter vil ny forespørsel være tilgjengelig og kjernen i crawleren kan kalle sendRequest() kallet. Rammeverket for 2. Behandling av HTTP



Figur 6.1: Klassediagram over rammeverk for forespørselsbehandling

svar og 3. Behandling av prioritert crawlerkø vil kunne utformes helt tilsvarende. For 2. Behandling av HTTP svar, vil hovedforskjellen bli å gjøre tilgjengelig responsen og eventuelt forrige nedlastet versjon slik at modulen kan bruke disse til å utføre duplikatdeteksjon. Derimot for 3. Behandling av prioritert crawlerkø, vil hovedforskjellen være tilgjengeligjeten på crawlerkøen og muligheter for å manipulere denne etter hver nedlasting og lagring av nytt dokument.

Fordelene ved en slik crawlerarkitektur er helt enkelt mindre tidkrevende crawlertilpasninger og høyere terskel for feiltoleranse når nye forskningsteorier og algoritmer skal testes.

6.4 Oppsumering

Prioritering av utviklerressurser er sentralt for allokering av tid til forskning og forbedringer av eksisterende funksjonalitet. For å kunne tilby mest mulig oppdaterte og effektive produkter er det viktig at virksomheter følger med på forskningsutviklingen og tilegner seg nye kunnskaper. Et viktig steg i dette arbeidet er implementering og testing av nye algoritmer i en virkelig kontekst for å se hvorvidt det er mulig med forbedringer. Det er intuitivt at desto lettere og mindre tidkrevende der er å gjennomføre slike eksperimenter, desto mer utbedret og nyutviklet teori kan implementeres og dras nytte av. Men tidsperspektivet er avhengig av hvilken arkitektur som foreligger og hvor tilpasningsdyktig denne er. Det er derfor essensielt med en platform som er så modulær som mulig slik at

ny forskning enkelt kan dras nytte av.

En helt enkelt prosedyre kan være å implementere et rammeverk som moduler må implementere, slik at kjernen i applikasjonen kan utnytte utvidet funksjonalitet og alternative algoritmeimplementasjoner.

Kapittel 7

Konklusjon og videre arbeid

Dette kapittelet oppsummerer og konkluderer oppgaven. Det gis en samlet oversikt over oppnådde resultater og forslag til fremtidig arbeid.

7.1 Oppnådde resultater

Adaptive metoder forenkler og effektiviserer crawlervirksomhet i en verden av kontinuerlig dynamisk innhold. Flere omfattende forskningsarbeider er utført innenfor adaptive metoder, men flere av disse er ikke implementerbare i praktiske og reelle situasjoner. I tillegg baserer mange av metodene seg på forutsetninger som forenkler utledning av metodene. Disse forutsetningene fører til at de endelige algoritmene er akseptable i en teoretisk sammenheng, men ikke i en virkelig kontekst.

I en dynamisk omgivelse er det viktig at crawlere er konfigurerbare og lett kan tilpasses nye formater og tjenere. Terskelen for å sette opp webtjenere for å distribuere innhold er forholdsvis lav som igjen fører til feilkonfurasjoner og inkosistente resultater. For at søkemotorer effektiv skal kunne tilby et mest mulig komplett speilbilde av world wide web, er det en betingelse at crawlere robust kan tilpasse seg inkonsistenser og hente ned innhold uten å feile.

En viktig tilpasning blant dynamisk endrende websider, er for crawleren å kunne forutsi når eventuelle forandringer har skjedd eller vil skje for individuelle elementer. På denne måten kan crawleren utføre et selektiv crawl kun på sider som med stor sikkerhet er forandret. På denne måten spares både nettverksressurser og crawlerressurser. I tillegg resulterer det i en mest mulig oppdatert kolleksjon som ikke går på bekostning av unødvendig bruk av tilgengelige ressurser. Erfaringer gjennom oppgaven tilsier dog at utarbeidede adaptive teorier ikke all-

tid gir ønskelige resultater i praksis på grunn av vanskeligheter med duplikatdeteksjon.

Implementasjonen har i tillegg indikert et sterkt behov for en pluggbar og konfigurerbar crawlerarkitektur. I situasjoner hvor det er nødvendig å gjennomføre testimplementasjoner for å verifisere algoritmer, burde arkitekturen være tilpasningsdyktig for å bruke minimal tid på test av uakseptable metoder.

Resultatene kan konkret oppsummeres i følgende liste:

- Identifikasjon og analyse av problemer med crawling av feilimplementerte tjenerinnstallasjoner, se kapittel 3
- Innsikt i nylig utførte forskningsarbeider innenfor adaptiv crawling, se kapittel 4
- Identifikasjon av mangler ved initielle forutsetninger for diverse adaptive metoder, se kapittel 4
- Fremheving av duplikatdeteksjon som en sentral crawleraktivitet og en viktig innledende aktivitet for adaptiv crawlertilpassing, se kapittel 5

På basis av punktene ovenfor er det utformet:

- Presentasjon av alternative metoder for duplikatdeteksjon, se kapittel 6
- Implementasjon og vurdering av adaptiv crawlertilpassing ved hjelp av Bayesiansk slutning, se kapittel 5
- Forslag til pluggbar crawlerarkitektur for forenklet tilpasning til dynamiske miljøer, inkonsistente tjenereimplementasjoner og alternative algoritmer for duplikatdeteksjon, se kapittel 6

7.2 Fremtidig arbeid

Med utgangspunkt i kapittel 6, vil vi foreslå arbeidet med en alternativ crawlerarkitektur som den mest prioriterte oppgaven med tanke på fremtidig arbeid. Både for en enkel tilpasning til konstant endrende arbeidsmiljø, og forenklet utvidelse av funksjonalitet. Som indikert i kapittelet er det 3 hovedområder for hvilke det er interessant å tilpasse design av crawlerarkitektur:

1. Alternative algoritmer for duplikatdeteksjon

2. Implementasjon av adaptive crawlertilpasninger

3. Dynamisk generering av HTTP forespørsler

Alle aktivitetene er relevante som fremtidige aktiviteter. Men fordi samtlige metoder stiller krav til en pluggbar crawlerarkitektur, er første prioritet arbeidet med arkitekturen slik den er foreslått i kapittel 6.

Implementasjon av alternative algoritmer for duplikatdeteksjon er også en viktig fremtidig aktivitet. Hovedsakelig fordi dette er en forutsetning for enhver fungerende adaptiv metode, se kapittel 5 for utdypende informasjon. Som utdypet i kapittel 6, er bruk av RSYNC til å utføre en vektet duplikatdetektering en interessant metode som burde tilegnes mer oppmerksomhet.

I tillegg er det naturlig å bruke mer tid på adaptiv teori og implementerbare metoder. Har man en riktig duplikatdeteksjon på plass, er det ingenting i veien for at en riktig adaptiv metode skal gi positivt utslag i en crawlerkontekst. Betingelsen er selvfølgelig at metoden er riktig i teorien og at den lar seg implementere.

Som resultat av erfaringer fra kapittel 3, er satsing på dynamisk genering av HTTP forespørsler en viktig egenskap for gjennomføring av mest mulig komplette globale crawl. I kapittel 6 er det skissert en enkel algoritme for hvordan dette kan gjøres. En relevant innledende aktivitet er igjen et enda mer pluggbart crawlerdesign, men moduler kan uansett implementeres og dokumenteres.

Bibliografi

[1] Hypertext transfer protocol-http/1.1 specification. *http://www.w3.org/Protocols/rfc2616/rfc2616.html*.

[2] Mathworld at wolfram research. *http://mathworld.wolfram.com/LagrangeMultiplier.html*.

[3] Rsync. *http://samba.anu.edu.au/rsync/*.

[4] Wikipedia, the free encyclopedia. *http://www.wikipedia.org/*.

[5] A. Signorini A. Gulli. The indexable web is more than 11.5 billion pages. *http://www.cs.uiowa.edu/ asignori/web-size/size-indexable-web.pdf*, May 2005.

[6] Michael K. Bergmann. The deep web: Surfacing hidden value. 2001.

[7] Ricardo Beaza-Yates Carlos Castillo. Practical issues of web crawling. *http://www.dcc.uchile.cl/ ccastill/papers/castillo_05_practical_web_crawling.pdf*, May 2005.

[8] Carlos Castillo. Effective web crawling. *In Frontmatters of the Ph. D. thesis, pages 2-8, http://www.dcc.uchile.cl/c̃castill/crawling_thesis/frontmatter.pdf*, November 2004.

[9] Hector Garcia-Molina Junghoo Cho. Estimating frequency of change. *Extended Version*, 2000.

[10] Hector Garcia-Molina Junghoo Cho. The evolution of the web and implications for an incremental crawler. *In Proceedings of International Conference on Very Large Data Bases(VLDB), pages 200-209, http://oak.cs.ucla.edu/ cho/papers/cho-evol.pdf*, Oktober 2000.

[11] Hector Garcia-Molina Junghoo Cho. Effective page refresh policies for web crawlers. *ACM Transactions on Database Systems, 28(4), http://oak.cs.ucla.edu/ cho/papers/cho-tods03.pdf*, Desember 2003.

[12] Hector Garcia-Molina Junghoo Cho. Estimating frequency of change. *ACM Transactions on Internet Technology, 3(3), http://oak.cs.ucla.edu/ cho/papers/cho-freq.pdf*, August 2003.

[13] Martijn Koster. Evaluation of the standard for robots exclusion. *http://www.robotstxt.org/wc/eval.html*, 1996.

[14] Audun Nordal. Pyropus, a scalable distributed web crawler. Master's thesis, Universitetet i Tromsø, May 2004.

[15] Hal R. Varian Peter Lyman. How much information 2003. *http://www2.sims.berkeley.edu/research/projects/how-much-info/internet.html*, 2003.