MapReduce-ohjelmointimalli

Mika Viinamäki

Kandidaatintutkielma
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Helsinki, 15. tammikuuta 2015

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Matemaattis-luonnontieteellinen	Tietojenkäsittelytieteen laitos

Tekijä — Författare — Author
Mika Viinamäki

Työn nimi — Arbetets titel — Title

MapReduce-ohjelmointimalli

Oppiaine — Läroämne — Subject
Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Kandidaatintutkielma	15. tammikuuta 2015	20

Tiivistelmä — Referat — Abstract

Suurten tietomäärien analysointi kestää pitkään, jos laskennassa hyödynnetään vain yhtä tietokonetta. MapReduce on Googlen vuonna 2003 esittelemä ohjelmointimalli, jota voi käyttää suurten tietomäärien käsittelyyn jakamalla laskentatyö usean tietokoneen kesken. Ohjelmointimalli käyttää abstraktionaan map- ja reduce-nimisiä funktioita, jotka ohjelmointimallin käyttäjä toteuttaa ohjelmassaan. Abstraktion käyttö vähentää hajautetun laskennan monimutkaisuutta; MapReduce-ohjelmointimallin toteuttavan kirjaston käyttäjän ei tarvitse kiinnittää huomiota moniin hajautetun laskennan yksityiskohtiin, kuten esimerkiksi laskentaan osallistuvien tietokoneiden väliseen kommunikointiin.

Tutkielmassa tutustutaan MapReduce-ohjelmointimallin käyttöön ja toimintaan. Lisäksi tutkielmassa tarkastellaan MapReduce-ohjelmointimallin kahta optimointia, Combiner-funktiota ja indeksien hyödyntämistä. Tutkielmassa myös verrataan MapReduce-ohjelmointimallia hajautettuihin relaatiotietokantajärjestelmiin ja Spark-ohjelmistokehykseen, jotka ovat MapReduce-ohjelmointimallin tavoin laskennan hajauttamiseen tarkoitettuja työkaluja.

ACM Computing Classification System (CCS):
Theory of computation → MapReduce algorithms
Computing methodologies → Distributed computing methodologies

Avainsanat — Nyckelord — Keywords
Hajautettu laskenta, MapReduce

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Sisältö

1 Johdanto

Suurten tietomäärien kerääminen ja analysointi on usein hyödyllistä liiketoiminnan ymmärtämisen ja tehostamisen kannalta. Esimerkiksi verkkokaupankäyntiin erikoistunut eBay kertoi vuonna 2013 säilyttävänsä tietovarastoissaan lähes 90 petatavua kaupankäyntiin liittyvää dataa [1]. Tarvetta suurten tietomäärien käsittelyyn esiintyy kuitenkin muuallakin kuin yrityksissä – vuonna 2010 fysiikan tutkimukseen käytetyn *Large Hadron Colliderin* päätunnistimen tuottamasta datasta jäi karsimisen jälkeen analysoitavaksi noin 13 petatavua dataa [5]. Tällaiset useiden kymmenien petatavujen suuruiset tietovarastot ovat suuruudeltaan monikymmentuhatkertaisia verrattuna tyypillisen kuluttajatietokoneen massamuistin kapasiteettiin [2]. Näin suuria tietomääriä onkin vaikea käsitellä käyttäen laskentaan vain yhtä tietokonetta.

Tässä tutkielmassa *hajautetulla laskennalla* tarkoitetaan kahden tai useamman tietokoneen hyödyntämistä jossain laskentaoperaatiossa. Hajautettua laskentaa voi tehdä käyttämällä esimerkiksi joukkoa tietoliikenneyhteyksillä toisiinsa yhdistettyjä itsenäisiä, usein yleisesti saatavilla olevista komponenteista rakennettuja tietokoneita. Tällaista joukkoa tietokoneita kutsutaan *klusteriksi* [4]. Yleisesti saatavilla olevista komponenteista rakennettujen klustereiden käyttö vaativiin laskentaoperaatioihin on havaittu erityisvalmisteisia supertietokoneita edullisemmaksi [4].

Suurten datan käsittelyyn erikoistuneiden yritysten, kuten Googlen, klustereihin voi kuulua satoja tai tuhansia tietokoneita [6]. Hyödyntääkseen hajautettua laskentaa ei ole kuitenkaan välttämätöntä tehdä suuria investointeja. Oman tietokoneklusterin hankkimisen sijaan yritykset voivat käyttää hyväkseen infrastruktuuria tai laskentaa palveluna tarjoavia yrityksiä, jolloin kustannuksia syntyy vain palvelun käytöstä [3].

Klusterissa tehtyyn laskentaan liittyy kuitenkin haasteita, joita yhdellä koneella tehdyssä laskennassa ei esiinny. Koska laskentaan osallistuu useampi tietokone, klusteria hyödyntävän ohjelman täytyy jakaa laskentatehtävä

[1] Inside eBay's 90PB data warehouse: `http://www.itnews.com.au/News/342615,inside-ebay8217s-90pb-data-warehouse.aspx`

[2] Pelijulkaisualusta *Steam* julkaisee kuukausittain tilastoja käyttäjiensä tietokoneista, mukaan lukien massamuistin koon – tätä kirjoittaessa yleisimmäksi massamuistin kooksi raportoidaan 250-499 gigatavua: `http://store.steampowered.com/hwsurvey/`

usealle tietokoneelle ja huolehtia kommunikaatiosta laskentaan osallistuvien tietokoneiden välillä [15], mikä tekee klusterissa tehdyssä laskennasta yhdellä koneella tehtävää laskentaa monimutkaisempaa. Lisääntyneestä monimutkaisuudesta huolimatta suurten tietomäärien käsittelyyn tarvitaan useampaa tietokonetta, sillä yhdellä tietokoneella ei voida käsitellä yhtä suuria tietomääriä kuin usealla tietokoneella.

Tutkielma esittelee MapReduce-ohjelmointimallin, joka on menetelmä käsitellä suuria tietomääriä hajautetusti. Luvussa 2 käydään läpi ohjelmointimalli sekä sen toiminta. Luvussa 3 esitellään kaksi erilaista MapReduce-suorituskyvyn parantamiseen tähtäävää optimointia, indeksointi sekä *combiner*-vaihe. Luku 4 näyttää esimerkin ohjelmointimallin käyttämisestä PageRank-menetelmän toteuttamisessa. Luvussa 5 esitellään kaksi muuta hajautetun laskennan ratkaisua, hajautetut relaatiotietokannat ja Spark-ohjelmistokehys sekä verrataan niitä MapReduce-ohjelmointimalliin. Luku 6 päättää tutkielman.

2 MapReduce-ohjelmointimalli

MapReduce on Googlen vuonna 2003 kehittämä ohjelmointimalli [7], jota käytetään suurten tietomäärien käsittelyyn ja tuottamiseen [6]. Ohjelmointimallin tarkoituksena on vähentää hajautetun laskennan monimutkaisuutta tarjoamalla useaan hajautetun laskennan sovellukseen sopiva abstraktio [6]. Käyttämällä sovelluksessaan MapReduce-kirjastoa ohjelmoijan ei tarvitse huolehtia monista hajautettuun laskentaan liittyvistä yksityiskohdista, kuten tietokoneiden välisestä kommunikaatiosta [6]. Eräs tunnettu MapReduce-ohjelmointimallin toteutus on osa avoimen lähdekoodin Apache Hadoop -projektia, jonka käyttäjiin kuuluvat muun muassa Facebook ja Yahoo! [17].

2.1 *Map*- ja *reduce*-funktiot

MapReduce-ohjelmointimallissa käyttäjä toteuttaa kaksi funktiota, joita kutsutaan nimillä *map* ja *reduce*. Funktiot ovat funktionaalisessa ohjelmoinnissa esiintyvien samannimisten funktioiden inspiroimia [6], mutta eivät suoraan

vastaa näitä funktioita [10]. Funktiota *map* käytetään käsittelemään syötteen alkioita, ja funktiota *reduce* käytetään yhdistämään alkiot.

Funktioiden *map* ja *reduce* tyypit on määritelty näin [6]:

$$map : (k1, v1) \rightarrow list(k2, v2)$$
$$reduce : (k2, list(v2)) \rightarrow list(v2).$$

Funktion *map* tarkoituksena on tuottaa tuloksia, joita myöhemmin käytetään *reduce*-funktion syötteenä [6]. *Map*-funktio muuntaa MapReduce-ohjelman syötteenään saamat avain-arvo-parit uusiksi avain-arvo-pareiksi. Näitä *map*-funktion tuloksia kutsutaan *välituloksiksi*. MapReduce-ohjelmointimalli ei ota kantaa minkään syötteen tai tuloksen avaimen tai arvon merkitykseen, vaan se riippuu käyttäjän syötteestä sekä *map*- ja *reduce*-funktioiden toteutuksesta.

Ohjelmointimallin toimintaa havainnollistaa seuraava pseudokoodimuotoinen esimerkki, joka laskee *kissa*- ja *koira*-sanojen esiintymien lukumäärää joukossa tekstimuotoisia dokumentteja:

```
def map(avain, arvo):
    # avain: dokumentin nimi
    # arvo: dokumentin sisältö
    for sana in arvo:
        if sana == "koira":
            emit("koira", 1)
        elif sana == "kissa":
            emit("kissa", 1)
```

Esimerkin *map*-funktio käy syötteenä saadun dokumentin jokaisen sanan läpi ja tuottaa avain-arvo-parin, mikäli sana on *koira* tai *kissa*. Avaimena käytetään löydettyä sanaa ja arvona kokonaislukua 1. Käyttäjän tarjoaman *map*-funktion soveltaminen kaikille syötteen avain-arvo-pareille – tässä tapauksessa syötteenä käytetyille dokumenteille – on laskennan ensimmäinen vaihe. *Map*-vaiheen jälkeen joukko välituloksia voisi näyttää esimerkiksi tältä:

$$(kissa, 1), (koira, 1), (kissa, 1), (koira, 1), (kissa, 1), (kissa, 1).$$

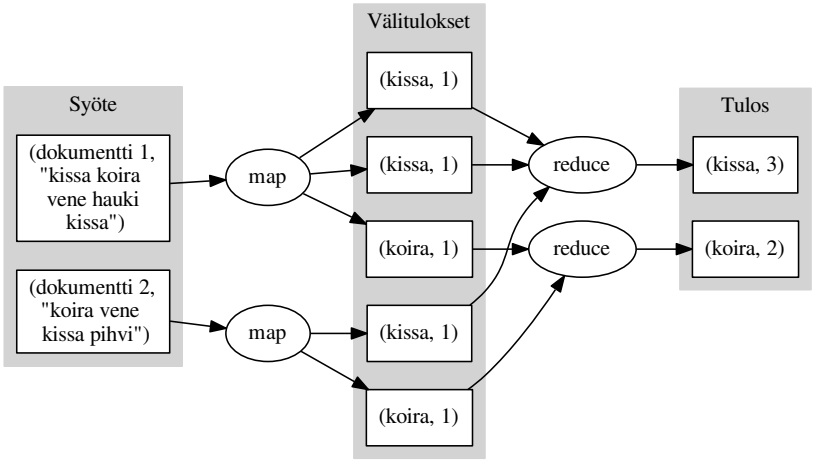
Laskennan toinen vaihe on käyttäjän tarjoaman *reduce*-funktion soveltaminen niihin välituloksiin, joilla on keskenään sama avain. Seuraava *reduce*-funktio laskee yhteen saman sanan esiintymien lukumäärät:

```
def reduce(avain, arvot):
    # avain: sana, "kissa" tai "koira"
    # arvot: lista sanan esiintymien lukumääriä
    summa = 0
    for arvo in arvot:
        summa += arvo
    emit(summa)
```

Jos laskennan tulos vastasi *map*-vaiheen jälkeen aiemmin esittämäämme mahdollista tulosta, näyttää laskennan tulos *reduce*-vaiheen jälkeen tältä:

$$(kissa, 4), (koira, 2).$$

Kuvassa 1 havainnollistetaan määriteltyjä *map*- ja *reduce*-funktioita. *Map*- ja *reduce*-funktioihin nuolella viittaavat laatikot kuvaavat funktioiden syötettä, ja funktion viittaamat laatikot funktion tulosta.



Kuva 1: Mahdollinen MapReduce-laskentatehtävän syöte, välitulokset ja lopullinen tulos

2.2 MapReduce-ohjelman suorituksen kulku

Googlen esittelemässä MapReduce-ohjelmointimallin toteutuksessa ohjelman suoritus alkaa käynnistämällä käyttäjän ohjelmasta kopio kaikilla laskentaan osallistuvilla tietokoneilla [6]. Yksi näistä kopioista on *isäntäprosessi* (master), joka koordinoi laskennan kulkua. Muut ohjelman kopiot ovat varsinaisen laskennan suorittavia *työprosesseja* (worker).

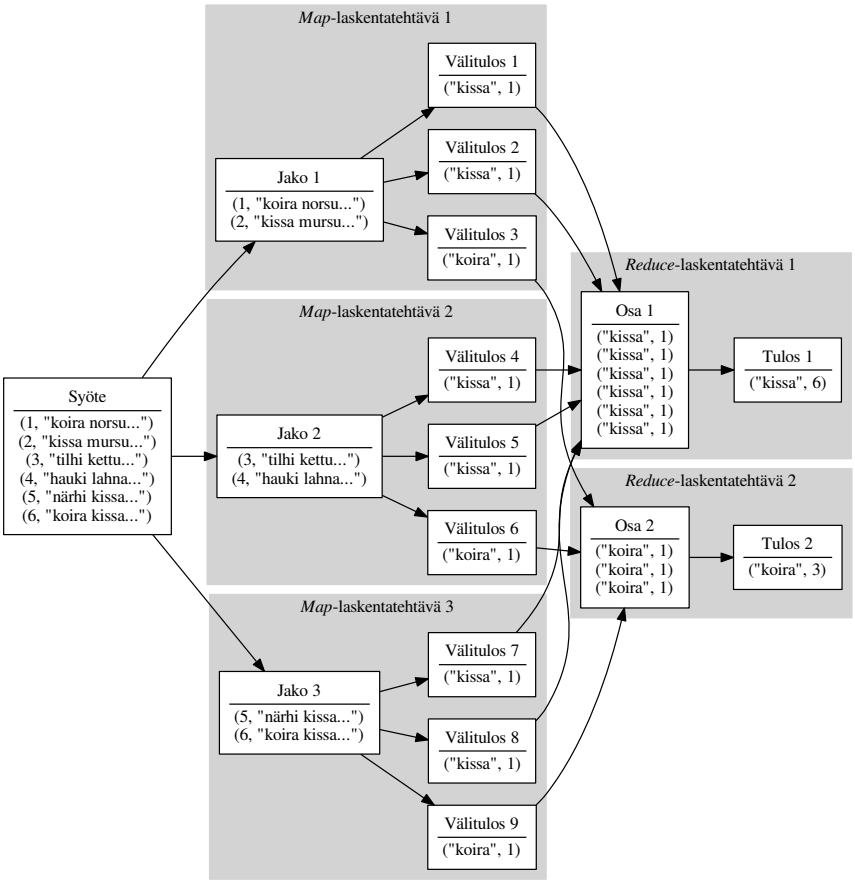
Jos syöte ei ole valmiiksi jaettu, se jaetaan pieniin osiin. Näitä osia kutsutaan *jaoiksi* (split), ja jokaiseen jakoon saattaa kuulua yksi tai useampi *map*-funktiolle annettava syötteen alkio. Jokaisesta jaosta muodostetaan *map*-laskentatehtävä, jonka isäntäprosessi luovuttaa jollekin työprosessille laskettavaksi. Syötteen jakaminen mahdollistaa sen käsittelyn useassa työprosessissa samanaikaisesti.

Map-laskentatehtävien tuloksena saatavista välituloksista muodostetaan *osia* (partition). Jokainen yksittäinen välitulos tallennetaan johonkin osaan, joka valitaan soveltamalla *hajautusfunktiota* välituloksen avaimeen. Näin saadaan aikaan osia, joissa eri avaimet ovat jakautuneet tasaisesti eri osien kesken ja joissa kaikki saman avaimen välitulokset päätyvät samaan osaan.

Jokaisesta osasta muodostetaan *reduce*-laskentatehtävä. Isäntäprosessi sijoittaa *reduce*-laskentatehtävät työprosessien laskettaviksi *map*-laskentatehtävien tavoin. Ennen *reduce*-funktion soveltamista välituloksiin työprosessi järjestää yhden osan välitulokset avaimen mukaan. Näin välitulokset joilla on sama avain ovat osan sisällä peräkkäin, ja avaimia voidaan käsitellä *reduce*-funktiolla yksi kerrallaan. Kun *reduce*-operaatio on yhden avaimen osalta valmis, laskenta on tämän avaimen välitulosten osalta tehty.

MapReduce-ohjelmointimalli ei rajoita syötteen lataamiseen tai tuloksen tallentamiseen käytettyjä tapoja. Syötteenä voidaan esimerkiksi käyttää joukkoa tiedostojärjestelmässä olevia tiedostoja, mutta ohjelmointimallin toteutus voi lisäksi mahdollistaa esimerkiksi tietokannan käytön syötteenä tai tuloksen tallennuskohteena [7]. Usein MapReduce-laskentatehtäviä halutaan ketjuttaa käyttäen saatua tulosta uuden MapReduce-laskentatehtävän syötteenä [6].

Kuva 2 havainnollistaa yllä esiteltyä MapReduce-laskentatehtävän suorituksen kulkua. Kuvassa näytetään, minkä laskentatehtävän alle laskennan eri



Kuva 2: MapReduce-laskentatehtävän suorituksen kulku

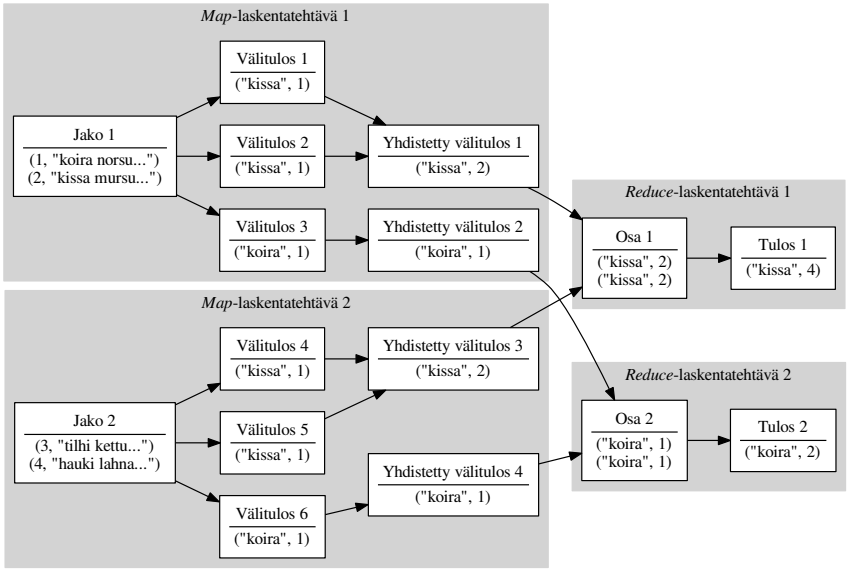
osat kuuluvat. Toisin kuin kuvan syötteellä, on yhteen osaan mahdollista kuulua useamman kuin yhden avaimen omaavia välituloksia.

3 MapReducen optimointeja

MapReduce-ohjelmointimallia sellaisenaan voidaan pitää melko yksinkertaisena. Ohjelmointimallin suorituskykyä voidaan kuitenkin parantaa laajentamalla sen toimintaa. Tässä luvussa tutustutaan kahteen MapReduce-ohjelmointimallin suorituskyvyn parantamiseen tähtäävään optimointiin, *combiner*-vaiheeseen sekä indeksointiin.

3.1 Combiner

Dean ja muut [6] esittelevät MapReduce-ohjelmointimallin lisäksi optimoinnin, joka lisää MapReduce-operaatioon uuden vaiheen nimeltään *combiner*. *Combiner*-vaiheen käyttö nopeuttaa *MapReduce*-operaation suoritusta erityisesti tilanteissa, joissa saman avaimen omaavia välituloksia on paljon.



Kuva 3: MapReduce-laskentatehtävä *combiner*-funktiolla varustettuna

Optimoinnin ideana on vähentää *map*- ja *reduce*-laskentatehtävien välistä kommunikaatiota tekemällä välitulosten osittaista yhdistämistä jo *map*-laskentatehtävän sisällä. Tämä toteutetaan soveltamalla *combiner*-funktiota yhden *map*-laskentatehtävän välitulosten yhdistämiseen *reduce*-funktion tapaan. Seurauksena on, että *map*- ja *reduce*-laskentatehtävien välillä tarvitsee välittää vain yksi välitulos jokaista avainta kohden. Näin ollen verkon yli *reduce*-laskentatehtäville lähetettävien välitulosten määrä vähenee. *Combiner*-funktion käyttämistä havainnollistetaan kuvassa 3. Kuvaan 2 verrattuna *map*- ja *reduce*-laskentatehtävien välistä kommunikaatiota on vähemmän.

Combiner-funktio toimii *reduce*-funktion tapaan, ja *combiner*-funktiona voidaankin usein käyttää *reduce*-funktioksi määriteltyä funktiota. *Combiner*-funktion käyttö muuttaa kuitenkin MapReduce-laskennan kulkua, sillä *combiner*-funktion käytöstä seuraa, että *reduce*-funktio ei saa enää parametrikseen kaikkia yhden avaimen välituloksia. Tästä seuraa, että kaikkia *reduce*-funktiota ei voida käyttää *combiner*-funktiona: alla oleva funktio on esimerkki *combiner*-funktioksi soveltumattomasta *reduce*-funktiosta. *Reduce*-funktiona käytettäessä funktion antama tulos kertoo, kuinka monta *kissa*- tai *koira*-sanaa tarvitaan, jotta niitä olisi sata.

```
# funktio toimii oikein reduce-funktiona,
# mutta ei combiner-funktiona
def combiner_ja_reduce(avain, arvot):
    tarvitaan = 100
    for arvo in arvot:
        tarvitaan -= arvo
    emit(tarvitaan)
```

Käyttämällä tätä funktiota *reduce*-funktion lisäksi *combiner*-funktiona tulokset olisivat virheellisiä, mutta käyttämällä luvussa 2.1 määrittelemäämme *reduce*-funktiota *combiner*-funktiona laskentatehtävän tulos pysyisi oikeana. Täsmällisesti *reduce*-funktiota voidaan käyttää myös *combiner*-funktiona jos *reduce*-funktio on vaihdannainen sekä liitännäinen [6].

3.2 Indeksien käyttäminen

MapReduce-ohjelmointimalli soveltuu sellaisenaan hyvin tarkoituksiin, joissa halutaan käsitellä suuren tietomäärän kaikkia tietueita. Usein kuitenkin

halutaan käsitellä vain pientä osaa jostain tietomäärästä, esimerkiksi jollain aikavälillä luotuja tai tietyn sanan sisältäviä dokumentteja. Pelkästään näiden dokumenttien käsittely MapReduce-ohjelmointimallin avulla edellyttää koko tietomäärän käymistä läpi ja haluttujen dokumenttien suodattamista *map*-vaiheessa, mikä suurilla tietomäärillä saattaa olla hidasta [14].

Tämän tyyppisten laskentatehtävien tehostamiseksi on luotu tekniikoita, jotka laajentavat MapReduce-ohjelmointimallia *indekseillä*. Indeksillä tarkoitetaan tietorakennetta, jolla pyritään nopeuttamaan tietueiden hakemista jonkin tietueeseen liittyvän kentän perusteella [8]. Indeksin käyttö kuitenkin edellyttää ensin indeksin olemassaoloa, ja sen luominen saattaa olla paljon laskentaresursseja vaativa operaatio – tällöin indeksointi onkin perusteltua vain, mikäli samaa syötettä käytetään laskentaoperaatioissa useita kertoja.

Richer ja muut esittelevät työssään [14] Apache Hadoop -projektin päälle rakennetun *Hadoop Aggressive Indexing Library* (HAIL) -kirjaston, jonka avulla voidaan hyödyntää indeksointia Hadoop-laskentatehtävissä. HAIL tarjoaa indeksin luomiseen kaksi erilaista menetelmää, joista molemmat välttävät erillisen, indeksin rakentavan laskentaoperaation. *Staattinen indeksointi* tarkoittaa tiedon indeksointia samalla, kun sitä siirretään Hadoop-projektiin kuuluvaan hajautettuun tiedostojärjestelmään. *Adaptiivinen indeksointi* tarkoittaa indeksin rakentamista samalla, kun indeksoitavaa dataa käytetään jonkin MapReduce-laskentatehtävän yhteydessä. Adaptiivinen indeksointi mahdollistaa indeksin hyödyntämisen myös sellaisella datalla, jolle ei ole rakennettu indeksiä hajautettuun tiedostojärjestelmään siirtämisen yhteydessä.

Käyttäjä voi hyödyntää indeksiä esimerkiksi määrittelemällä *map*-funktion yhteyteen suodattimen, jolloin *map*-laskentatehtävä saa syötteekseen vain suodattimen hyväksymiä tietueita. Koska indeksistä tietueiden hakeminen on nopeaa, on indeksin käyttäminen suodatuksessa tehokkaampaa kuin datan suodattaminen vasta *map*-laskentatehtävän yhteydessä. Artikkelissa esiteltyjen tuloksien mukaan datan siirtoon käytetyn HAIL-asiakasohjelman tehokkuuden vuoksi staattinen indeksointi datan siirtämisen yhteydessä ei ole hitaampaa kuin Hadoop-projektin mukana tulevan asiakasohjelman käyttö datan siirtämiseen. Varsinaisen Hadoop-laskentatehtävän suorituskykyä indeksin käyttäminen paransi 64-kertaisesti [14].

Indeksointia voi kuitenkin hyödyntää ilman, että MapReduce-ohjelmointimallia laajennetaan millään tavalla. Tämä onnistuu käyttämällä syötteenä tietoa, jonka muodostamisessa on käytetty indeksointia. Tälläinen syöte voi olla esimerkiksi indeksejä käyttävään tietokantaan tehdyn kyselyn tulos [7].

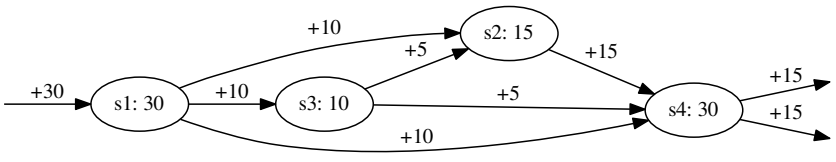
4 MapReducen sovellus: PageRank

PageRank on menetelmä, jolla voidaan järjestää Internet-sivuja tärkeysjärjestykseen niihin osoittavien linkkien perusteella [12]. Menetelmän ajatuksena on, että usein viitatut Internet-sivut ovat tärkeämpiä kuin sellaiset, joihin on vähemmän viitattauksia. Mitä tärkeämpi sivu on, sitä enemmän sen viittaukset nostavat viitattujen sivujen PageRank-arvoa. Google-hakukone sai alkunsa PageRank-menetelmästä [12]. PageRank-menetelmä toimii esimerkkinä hieman monimutkaisemmasta MapReduce-ohjelmointimallin avulla toteutettavasta algoritmista.

Määritellään PageRank-menetelmän yksinkertaistettu versio. Olkoon s jokin Internet-sivu, ja V_s sivuun s viittaavien sivujen joukko. Internet-sivun s PageRank on nyt:

$$PageRank(s) = \sum_{v \in V_s} \frac{PageRank(v)}{linkkienMaaraSivulla(v)}.$$

Kuvassa 4 näytetään esimerkki PageRank-menetelmästä käytännössä. Kuvassa on otos PageRank-arvon laskemiseen käytetyistä sivuista, niiden PageRank-arvoista, sekä viittausten vaikutuksista sivujen PageRank-arvoon.



Kuva 4: Esimerkki PageRank-menetelmästä

PageRank lasketaan usein käyttäen *iteratiivista menetelmää* [11]. Esimerkkinä iteratiivisesta menetelmästä on seuraava algoritmi, joka laskee PageRank-arvon yksinkertaistetun version jollekin joukolle sivuja siten. Algoritmissa sivujen PageRank-arvot ovat aluksi karkeita, mutta tarkentuvat jokaisen iteraation jälkeen.

1. Aseta jokaiselle sivulle PageRank-arvoksi jokin vakio, esimerkiksi 1.
2. Laske jokaiselle sivulle uusi PageRank-arvo käyttäen yllä esitettyä kaavaa siten, että laskettaessa uutta iteraatiota käytetään viime iteraation PageRank-arvoja.

$$uusiPageRank(s) = \sum_{v \in V_s} \frac{edellinenPageRank(v)}{linkkienMaaraSivulla(v)}.$$

3. Toista kohtaa 2, kunnes ollaan tehty haluttu määrä iteraatioita tai ollaan saavutettu haluttu tarkkuus.

Toteutetaan tämän iteratiivisen algoritmin kohta 2. käyttäen MapReduce-ohjelmointimallia. Aloitetaan määrittelemällä MapReduce-ohjelmamme saama syöte. Algoritmi tarvitsee tiedon jokaisesta Internet-sivusta, niiden nykyisistä PageRank-arvoista sekä listan viittauksista toisiin Internet-sivuihin. Syötteeseen saadaan kaikki tarvittava sisältö määrittelemällä se niin, että syötteen avaimena on jokin sivun yksilöivä tunniste ja arvona pari, jonka ensimmäinen alkio on sivun nykyinen PageRank-arvo ja toinen alkio sivulta löytyvät viittaukset. Esitetään nämä viittaukset listana sivuja yksilöiviä tunnisteita.

```
def map(sivu_id, (page_rank, viittaukset)):
    pr_per_viittaus = page_rank / len(viittaukset)
    for viittaus in viittaukset:
        emit(viittaus, pr_per_viittaus)
```

Yllä määritelty *map*-funktio luo jokaisesta sivulta löytyvästä viittauksesta välituloksen, jonka avaimena on viitatun sivun tunniste ja arvona viittavan sivun vaikutus viitatun sivun PageRank-arvoon. Nyt *reduce*-funktion tehtäväksi jää yhdistää nämä osittaiset PageRank-arvot sivun lopulliseksi PageRank-arvoksi:

```
def reduce(sivu, arvot):
```

```
    page_rank = 0
    for arvo in arvot:
        page_rank += arvo
    emit(sivu, page_rank)
```

Määritelty MapReduce-ohjelma laskee jokaiselle sivulle uuden PageRank-arvon ja näin suorittaa yhden iteraation aiemmin kuvaillusta iteratiivisesta algoritmista. Algoritmissa on kuitenkin pieni ongelma: ohjelman tuloksessa ei ole enää tietoja sivujen viittauksista toisiin sivuihin. Niinpä ohjelman tulosta ei voi käyttää suoraan uuden iteraation syötteenä. Koska algoritmi on iteratiivinen, mahdollisuus käynnistää uusi iteraatio käyttäen syötteenä aiemman iteraation tulosta on toivottu ominaisuus.

Ongelma voidaan ratkaista usealla eri tavalla. Koska sivujen väliset viittaukset eivät muutu iteraatioiden välillä, ei ole välttämättä mielekästä säilyttää tätä tietoa syötteessä lainkaan. Sen sijaan tiedot sivujen välisistä viittauksista voidaan siirtää kaikille työläisprosessien tietokoneille ennen iteraatioiden aloittamista, jolloin *map*-laskentatehtävät voivat lukea tiedot sivujen välisistä viittauksista varsinaisen MapReduce-laskentatehtävässä käytetyn syötteen ulkopuolelta.

Mikäli näin ei haluta tai voida menetellä, voidaan muuttaa määrittelemiämme *map*- ja *reduce*-funktioita niin, että tiedot sivujen välisistä viittauksista eivät häviä. Tämä voidaan toteuttaa esimerkiksi käyttämällä kahden eri tyyppisiä välituloksia, joista toiset ilmaisevat viittauksia sivulta toiselle ja toiset vaikutuksia PageRank-arvoihin [11]. Alla olevat *map*- ja *reduce*-funktiot toteuttavat tämän idean.

```
def map(sivu_id, (page_rank, viittaukset)):
    pr_per_viittaus = page_rank / len(viittaukset)
    for viittaus in viittaukset:
        emit(viittaus, PageRank(pr_per_viittaus))
        emit(sivu_id, Viittaus(viittaus))


def reduce(sivu_id, arvot):
    page_rank = 0
    viittaukset = []
    for arvo in arvot:
```

```
        if type(arvo) == PageRank:
            page_rank += arvo
        elif type(arvo) == Viittaus:
            viittaukset.append(arvo)
    emit(sivu_id, (page_rank, viittaukset))
```

5 Muut hajautetun laskennan ratkaisut

MapReduce-ohjelmointimalli ei ole ainoa tai ensimmäinen ratkaisu suurien tietomäärien käsittelyyn. Tässä luvussa tutustutaan kahteen muuhun hajautetun laskennan ratkaisuun ja verrataan näitä ratkaisuja MapReduce-ohjelmointimalliin.

5.1 Hajautetut relaatiotietokantajärjestelmät

Relaatiotietokantajärjestelmien ja SQL-kyselykielen suosion vuoksi saattaa olla houkuttelevaa käyttää kyselykieltä myös suurien tietomäärien analysoinnissa – SQL saattaa olla ohjelmoijalle valmiiksi tuttu, tai kenties kehityksen kohteena oleva järjestelmä käyttää jo SQL-kyselykieltä hyödyntävää tietokantaa hyväkseen.

Tiedon käsittely MapReduce-ohjelmointimallilla ja relaatiotietokantajärjestelmillä eroaa monin tavoin, muun muassa seuraavasti:

- **Ohjelmointimalli**: MapReduce-ohjelmointimallin käyttäjä toteuttaa tiedon käsittelyn *map*- ja *reduce*-funktioiden avulla. *Map*- ja *reduce*-funktiot toteutetaan tavallisesti yleiskäyttöisellä ohjelmointikielellä. Tästä seuraa, että funktioiden sisältämälle logiikalle tai sisäiselle rakenteelle ei ole asetettu rajoituksia. Relaatiotietokannassa tiedon käsittely tehdään SQL-kyselykielellä, jolla – toisin kuin *map*- tai *reduce*-funktioilla – kuvataan kyselyn haluttua tulosta eikä laskennan yksityiskohtia [13]. Suurin osa tunnetuista relaatiotietokantajärjestelmistä tukee kuitenkin myös jonkinlaista ohjelmointikieltä kyselyiden rakentamisessa.
- **Tiedon rakenne**: MapReduce-ohjelmointimalli ei ota kantaa syötteen tai tuloksen rakenteeseen [7]. Relaatiotietokannat käyttävät tiedon

ilmaisemiseen kaksiulotteisia tauluja, joiden rakenne määritellään ennen kuin tietokantaan voidaan lisätä sisältöä [13].

- **Suoritus**: MapReduce-ohjelmointimalli sellaisenaan rajoittuu malliin, jossa tieto käy ensin *map*- ja sen jälkeen *reduce*-vaiheen läpi. Koska SQL-kyselyissä ei esitetä laskennan yksityiskohtia, voi käytetty relaatiotietokantajärjestelmä valita kyselylle parhaan mahdollisen suoritusmallin [13].

Stonebraker ja muut vertasivat työssään [16] Hadoop-kirjaston suorituskykyä hajautettuihin relaatiotietokantajärjestelmiin 100 tietokoneen klusterilla. Suorituskykytesteissä hajautetun relaatiotietokantajärjestelmän *Vertican* sekä toisen, nimeämättä jätetyn hajautetun relaatiotietokantajärjestelmän havaittiin olevan merkittävästi testattuja kyselyitä vastaavia Hadoop-ohjelmia tehokkaampia. Osasyyksi todetaan Hadoop-ohjelmien indeksoinnin puute – toisaalta, kuten luvussa 3.2 todetaan, indeksien käyttö MapReduce-sovelluksissa ei ole mahdotonta. Testatut hajautetut relaatiotietokantajärjestelmät olivat kuitenkin nopeampia myös tehtävissä, joissa indeksointia ei voitu hyödyntää. Artikkelissa kuitenkin todetaan, että tulokset kertovat enemmän testattujen järjestelmien kuin niiden käyttämien mallien eroista.

On olemassa myös järjestelmiä, jotka mahdollistavat perinteisten relaatiotietokantajärjestelmien käytön hajautetusti – tällaisia järjestelmiä ovat esimerkiksi *pgpool-II* [1] sekä MapReduce-ohjelmointimallia toteutuksessaan hyödyntävä *HadoopDB* [2]. Molemmat toimivat ylimääräisenä kerroksena käyttäjän ja itsenäisten PostgreSQL-tietokantapalvelinten välissä. MapReduce-ohjelmointimallin joustavuuden vuoksi sen päälle voi rakentaa lisäksi järjestelmiä, joiden avulla MapReduce-laskentatehtäviä voidaan määrittää SQL:ää muistuttavan kyselykielen avulla. Esimerkki tällaisesta järjestelmästä on *Apache Hive* [17].

5.2 Spark

Spark on Scala-ohjelmointikielellä toteutettu ohjelmointikehys hajautettua laskentaa varten [19]. Ohjelmistokehyksen esitetään MapReduce-ohjelmointimalliin verrattuna helpottavan käyttötarkoituksia, joissa samaa joukkoa tietoa käytetään useaan kertaan. Tällaisia ovat muun muassa iteratiiviset algoritmit, kuten luvussa 4 esitelty algoritmi PageRank-menetelmän

toteuttamiseen.

Spark-ohjelmointikehyksen käyttämä abstraktio on *kestävät, hajautetut tietojoukot* (Resilient Distributed Dataset, RDD), jotka ovat kokoelma laskennassa käytettäviä alkioita [18]. Kestäviä, hajautettuja tietojoukkoja voidaan niiden luomisen jälkeen ainoastaan lukea, joten kaikki muutokset tietojoukkoihin luovat uuden tietojoukon. Käyttäjän näkökulmasta hajautettu laskenta Spark-ohjelmistokehyksellä muistuttaa MapReduce-ohjelmointimallin käyttämistä: käyttäjä voi käsitellä kestäviä, hajautettuja tietojoukkoja erilaisin funktioin. Funktioihin kuuluvat muun muassa *flatMap*- ja *reduceByKey*, joilla tietoa voidaan käsitellä MapReduce-ohjelmointimallin *map*- ja *reduce*-funktioiden tapaan.

Seuraava esimerkki havainnollistaa, miten saman tietojoukon käyttäminen useaan kertaan onnistuu Spark-ohjelmistokehystä käytettäessä:

```
// olkoon muuttuja "rdd" jokin kokonaislukuja sisältävä
// tietojoukko
var parilliset = rdd.filter(x -> x % 2 == 0).cache()
var maara = parilliset.count()
var summa = parilliset.reduce((a, b) -> (a + b))
```

Esimerkissä suodatetaan tietojoukko *filter*-funktion avulla niin, että suodatuksen tuloksena saadussa uudessa tietojoukossa on vain alkuperäisen tietojoukon parilliset alkiot. *Cache*-funktion käyttö kertoo Spark-ohjelmistokehykselle, että tietojoukko kannattaa pyrkiä säilyttämään muistissa sen ensimmäisen laskentakerran jälkeen.

Tällainen laskettujen tulosten uudelleenkäyttö on kuitenkin mahdollista myös MapReduce-ohjelmointimallin avulla – yhden MapReduce-laskentatehtävän tulos voidaan tallentaa ja käyttää sitä muissa MapReduce-laskentatehtävissä. Spark-ohjelmistokehyksen esitetään kuitenkin toimivan MapReduce-ohjelmointimallia tehokkaammin, sillä MapReduce-laskentatehtävien täytyy tallentaa ja ladata tulokset levyltä [19]. Kuitenkin, koska MapReduce-ohjelmointimallissa tulosten tallennuskohteena toimivaa tekniikkaa ei ole rajoitettu, voidaan tulokset tallentaa ja ladata esimerkiksi muistissa toimivasta hajautetusta tiedostojärjestelmästä.

Kestävien, hajautettujen tietojoukkojen lisäksi Spark-ohjelmistokehyksessä

on ominaisuuksia, joilla voidaan hallita laskentaan osallistuvien tietokoneiden välistä yhteistä tilaa [19]. Esimerkiksi *yleislähetys*-muuttujien (broadcast) avulla voidaan lähettää tietoa kaikille tietokoneille tehokkaasti. Tätä ominaisuutta voisi käyttää esimerkiksi luvussa 4 esitellyn PageRank-algoritmin sivujen välisten yhteyksien lähettämiseen kaikille tietokoneille.

Lei Gu ja Huan Li vertasivat Hadoop- ja Spark-ohjelmistokehyksien suorituskykyä iteratiivisissa laskentatehtävissä [9]. Spark-ohjelmien todettiin olevan keskimäärin Hadoop-ohjelmia nopeampia, mutta käyttävät enemmän keskusmuistia sekä hidastuvat, jos keskusmuistia ei ole käytössä riittävästi.

6 Yhteenveto

MapReduce on ohjelmointimalli, jota voi käyttää suurten tietojoukkojen käsittelyyn ja luomiseen. MapReduce-ohjelmointimalli helpottaa hajautettua laskentaa käyttävien ohjelmien luomista monin tavoin: käyttämällä ohjelmointimallin toteuttavaa kirjastoa käyttäjän ei tarvitse huolehtia useista hajautettuun laskentaan liittyvistä yksityiskohdista, ja luvussa 2 käsiteltyjen esimerkkien perusteella ohjelmointimallin käyttö yksinkertaisten ohjelmien luomiseen on suoraviivaista.

MapReduce-ohjelmointimallin yksinkertaisuudesta seuraa kuitenkin, että sellaisenaan se soveltuu tiettyihin käyttötarkoituksiin huonosti. MapReduce-ohjelman on esimerkiksi käytävä koko sen saama syöte läpi, joka joissain sovelluksissa vähentää suorituskykyä tarpeettomasti. Ohjelmointimallin joustavuudesta kuitenkin kertoo, että sen puutteita voidaan paikata muuttamalla ohjelmointimallin toimintaa sopivasti: laajentamalla ohjelmointimallia *indeksoinnilla* vältetään koko syötteen käyminen läpi, ja laskentaan osallistuvien tietokoneiden välistä kommunikaatiota voidaan vähentää käyttämällä *combiner*-funktiota. Muutokset eivät kuitenkaan ole puhtaasti suorituskykyä parantavia – esimerkiksi indeksin rakentaminen vähentää turhaan suorituskykyä, jos indeksille ei ole käyttöä. Niinpä käyttäjän on punnittava eri laajennosten hyötyjä ja haittoja sovelluskohtaisesti.

MapReduce-ohjelmointimallia voidaan käyttää monentyyppisten algoritmien toteuttamiseen. Luvussa 3 esitelty PageRank-menetelmän toteuttava ohjelma osoittaa, että ohjelmointimalli soveltuu yksinkertaisten ohjelmien lisäksi

esimerkiksi verkkoja käyttävien tai iteratiivisten algoritmien toteuttamiseen.

Spark on hajautettuun laskentaan tarkoitettu ohjelmistokehys, jolla voidaan käsitellä tietoa MapReduce-ohjelmointimallin tapaan. Spark-ohjelmistokehys sisältääkin funktiot, jotka vastaavat MapReduce-ohjelmointimallista löytyviä *map*- ja *reduce*-funktioita, tarjoten kuitenkin myös lisää ominaisuuksia MapReduce-ohjelmointimalliin verrattuna. Ohjelmistokehys helpottaa tietoa useaan kertaan käyttävien algoritmien toteuttamista sekä laskentaan osallistuvien tietokoneiden välisen yhteisen tilan hallintaa.

Hajautetut relaatiotietokantajärjestelmät sen sijaan eroavat MapReduce-ohjelmointimallista selkeämmin, ja molemmissa on toisiinsa verrattuna hyviä sekä huonoja puolia. Yleiskäyttöisen ohjelmointikielen sijaan relaatiotietokannoissa käytetään yleensä SQL-kyselykieltä, joka kuvaa laskennan yksityiskohtien sijaan haluttua tulosta. Relaatiotietokannoissa käytetään tiedon esittämiseen kaksiulotteisia taulukoita, kun taas MapReduce-ohjelmointimallissa tiedon esittämiseen käytettyä mallia ei ole rajoitettu. Toisaalta, koska kyselyissä ei määritellä laskennan yksityiskohtia, relaatiotietokantajärjestelmät ovat vapaita suorittamaan kyselyt parhaaksi katsomallaan tavalla.

Tutkielman tarkoituksena oli luoda katsaus MapReduce-ohjelmointimalliin, sen toimintaan sekä näyttää, miten ohjelmointimallia voi hyödyntää tiedon käsittelyssä. Tutkielma ei ole kovinkaan kattava katsaus erilaisiin MapReduce-ohjelmointimalliin liittyviin asioihin. Ulkopuolelle ovat jääneet esimerkiksi usein MapReduce-ohjelmointimallin kanssa käytetyt hajautetut tiedostojärjestelmät ja niihin liittyvät MapReduce-ohjelmointimallin optimoinnit. On olemassa myös hajautetun laskennan järjestelmiä, joita ei tässä tutkielmassa verrattu MapReduce-ohjelmointimalliin. Esimerkki tälläisestä järjestelmästä on Spark-ohjelmistokehyksen päälle rakennettu GraphX-kirjasto, jota voidaan käyttää verkkojen käsittelyyn.

Lähteet

[1] *pgpool-II.* `http://www.pgpool.net/`.

[2] Abouzeid, Azza, Bajda-Pawlikowski, Kamil, Abadi, Daniel, Silberschatz, Avi ja Rasin, Alexander: *HadoopDB: An Architectural Hybrid of MapRe-*

duce and DBMS Technologies for Analytical Workloads. Proc. VLDB Endow., 2(1):922–933, elokuu 2009, ISSN 2150-8097. http://dx.doi.org/10.14778/1687627.1687731.

[3] Armbrust, Michael, Fox, Armando, Griffith, Rean, Joseph, Anthony D., Katz, Randy, Konwinski, Andy, Lee, Gunho, Patterson, David, Rabkin, Ariel, Stoica, Ion ja Zaharia, Matei: *A View of Cloud Computing*. Commun. ACM, 53(4):50–58, huhtikuu 2010, ISSN 0001-0782. http://doi.acm.org/10.1145/1721654.1721672.

[4] Baker, Mark ja Buyya, Rajkumar: *Cluster computing: the commodity supercomputer*. Software-Practice and Experience, 29(6):551–76, 1999.

[5] Brumfiel, Geoff: *Down the petabyte highway*. Nature, 469(20):282–283, 2011.

[6] Dean, Jeffrey ja Ghemawat, Sanjay: *MapReduce: Simplified Data Processing on Large Clusters*. Commun. ACM, 51(1):107–113, tammikuu 2008, ISSN 0001-0782. http://doi.acm.org/10.1145/1327452.1327492.

[7] Dean, Jeffrey ja Ghemawat, Sanjay: *MapReduce: A Flexible Data Processing Tool*. Commun. ACM, 53(1):72–77, tammikuu 2010, ISSN 0001-0782. http://doi.acm.org/10.1145/1629175.1629198.

[8] Fuh, Gene YC, Jou, Michelle Mei Chiou, Kazimierowicz, Kamilla, Tran, Brian Thinh Vinh ja Wang, Yun: *Generalized model for the exploitation of database indexes*, 2001. US Patent 6,253,196.

[9] Gu, Lei ja Li, Huan: *Memory or Time: Performance Evaluation for Iterative Operation on Hadoop and Spark*. Teoksessa *High Performance Computing and Communications 2013 IEEE International Conference on Embedded and Ubiquitous Computing, 2013 IEEE 10th International Conference on*, sivut 721–727, Nov 2013.

[10] Lämmel, Ralf: *Google's MapReduce programming model—Revisited*. Science of computer programming, 70(1):1–30, 2008.

[11] Lin, Jimmy ja Schatz, Michael: *Design Patterns for Efficient Graph Algorithms in MapReduce*. Teoksessa *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*, MLG '10, sivut 78–85, New York,

NY, USA, 2010. ACM, ISBN 978-1-4503-0214-2. http://doi.acm.org/10.1145/1830252.1830263.

[12] Page, Lawrence, Brin, Sergey, Motwani, Rajeev ja Winograd, Terry: *The PageRank Citation Ranking: Bringing Order to the Web.* Technical Report 1999-66, Stanford InfoLab, November 1999. http://ilpubs.stanford.edu:8090/422/, Previous number = SIDL-WP-1999-0120.

[13] Pavlo, Andrew, Paulson, Erik, Rasin, Alexander, Abadi, Daniel J., DeWitt, David J., Madden, Samuel ja Stonebraker, Michael: *A Comparison of Approaches to Large-scale Data Analysis.* Teoksessa *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, sivut 165–178, New York, NY, USA, 2009. ACM, ISBN 978-1-60558-551-2. http://doi.acm.org/10.1145/1559845.1559865.

[14] Richter, Stefan, Quiané-Ruiz, Jorge Arnulfo, Schuh, Stefan ja Dittrich, Jens: *Towards Zero-overhead Static and Adaptive Indexing in Hadoop.* The VLDB Journal, 23(3):469–494, kesäkuu 2014, ISSN 1066-8888. http://dx.doi.org/10.1007/s00778-013-0332-z.

[15] Sadashiv, Naidila ja Kumar, SM Dilip: *Cluster, grid and cloud computing: A detailed comparison.* Teoksessa *Computer Science & Education (ICCSE), 2011 6th International Conference on*, sivut 477–482. IEEE, 2011.

[16] Stonebraker, Michael, Abadi, Daniel, DeWitt, David J., Madden, Sam, Paulson, Erik, Pavlo, Andrew ja Rasin, Alexander: *MapReduce and Parallel DBMSs: Friends or Foes?* Commun. ACM, 53(1):64–71, tammikuu 2010, ISSN 0001-0782. http://doi.acm.org/10.1145/1629175.1629197.

[17] Thusoo, A, Sarma, J.S., Jain, N., Shao, Zheng, Chakka, P., Zhang, Ning, Antony, S., Liu, Hao ja Murthy, R.: *Hive - a petabyte scale data warehouse using Hadoop.* Teoksessa *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, sivut 996–1005, March 2010.

[18] Zaharia, Matei, Chowdhury, Mosharaf, Das, Tathagata, Dave, Ankur, Ma, Justin, McCauley, Murphy, Franklin, Michael J, Shenker, Scott ja Stoica, Ion: *Resilient distributed datasets: A fault-tolerant abstraction for*

in-memory cluster computing. Teoksessa *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, sivut 2–2. USENIX Association, 2012.

[19] Zaharia, Matei, Chowdhury, Mosharaf, Franklin, Michael J, Shenker, Scott ja Stoica, Ion: *Spark: cluster computing with working sets*. Teoksessa *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, sivut 10–10, 2010.