View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Helsingin yliopiston digitaalinen arkisto

Laajan mittakaavan Internet-sovelluksia varten kehitetyt hajautetut tietokannat

Joel Sandborg

Helsinki 28.12.2012
Pro gradu -tutkielma
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta – Fakultet –Faculty	Laitos – Institution – Department
Matemaattis-luonnontieteellinen tiedekunta	Tietojenkäsittelytieteen laitos

Tekijä – Författare – Author

Joel Sandborg

Työn nimi – Arbetets titel – Title

Laajan mittakaavan Internet-sovelluksia varten kehitetyt hajautetut tietokannat

Oppiaine – Läroämne – Subject

Tietojenkäsittelytiede

Työn laji – Arbetets art – Level	Aika – Datum – Month and year	Sivumäärä – Sidoantal – Number of pages
Pro gradu -tutkielma	28.12.2012	117

Tiivistelmä – Referat – Abstract

Suurten Internet-yritysten, kuten Googlen ja Amazonin tarjoamat palvelut edellyttävät valtavien hajautettujen tietomäärien käsittelyä ja varastoimista. Tiedon pitää olla hyvin saatavilla. Tietokantajärjestelmältä edellytetään myös hyvää suorituskykyä. Suorituskyvyn ylläpitämiseksi järjestelmän täytyy skaalautua niin, että tarpeen vaatiessa järjestelmään voidaan lisätä enemmän resursseja. Tietokannan rakenteen tulee olla lisäksi joustava ja helposti muokattavissa. Perinteiset relaatiotietokannat transaktionaalisine oikeellisuus- ja eristyvyysvaatimuksineen ovat olleet liian rajoittavia tähän tarkoitukseen, joten näiden laajan mittakaavan Internet-sovellusten vaatimuksiin on kehitetty muita vaihtoehtoja. Näitä järjestelmiä on alettu kutsua NoSQL-tietokantajärjestelmiksi. NoSQL-tietokannat ovat usein niin erikoistuneita, ettei relaatiomallia ja SQL-kyselykielen koko ilmaisuvoimaa tarvita tai voida käyttää. Näiden tietokantojen tietomalli perustuu avain-arvo-pariin, jossa varastoitu arvo on yksilöity indeksoitavan avaimen perusteella. Tietokannan skeema on taas usein hyvin joustava, tai tietokanta saattaa olla jopa kokonaan skeematon. Käytössä olevat funktiot ovatkin usein rajoittuneet yksittäisten avain-arvo-parien lukemiseen ja päivittämiseen. Näiden tietojen laajan mittakaavan rinnakkaiseen laskentaan on lisäksi kehitetty yksinkertainen MapReduce-ohjelmointiparadigma. Google ja Amazon hyödyntävät näitä järjestelmiä varten rakentamaansa laajan mittakaavan infrastruktuuria tarjoamalla sitä myös muiden yritysten sovelluksien alustaksi NoSQL-tietokantapalveluna.

Tässä tutkielmassa pyritään selventämään NoSQL-tietokantajärjestelmien tallennusratkaisun ja tiedon käsittelyn periaatteita, eroja relaatiotietokantajärjestelmiin sekä millaiseen käyttöön nämä uudet tietokantajärjestelmät oikein soveltuvat. Tutkielmassa esitellään myös MapReduce-ohjelmointiparadigma, NoSQL-tietokantapalveluna sekä joitakin NoSQL-tietokantajärjestelmien luokittelutapoja ja tietokannan tietomalleja. Tutkielma perustuu pääosin aikaisemmin aiheesta laadittuun kirjalliseen materiaaliin, kuten lehti- ja konferenssiartikkeleihin sekä kirjoihin.

NoSQL-tietokantajärjestelmien nykyistä kehitysvaihetta voidaan verrata aikaan ennen SQL:ää. Nämä järjestelmät ovat kovin heterogeeninen joukko, joten myös niiden luokittelu on vaikeaa. NoSQL-tietokantajärjestelmissä ei ole perinteisten relaatiotietokantajärjestelmien pitkälle kehitettyjä ominaisuuksia. Suurin osa edellä mainituista ominaisuuksista pitää toteuttaa sovelluslogiikassa, joten ne jäävät sovellusohjelmoijan vastuulle. Mikään tietokantajärjestelmä tai työkalu ei ole paras ratkaisu kaikkiin tehtäviin. Kussakin järjestelmässä on järkevää ja tehokasta käsitellä ja varastoida pääosin tietyn kaltaista sovellusalueen tietoa. Sopiva tietokantajärjestelmä tai työkalu riippuu täysin yrityksen ja sovelluksen vaatimuksista. Yrityksen tulee siis arvioida sovellusalueen tietojen vaatimuksia.

ACM Computing Classification System (CCS):
C.2.4 [Computer-Communication Networks]: Distributed Systems - Distributed databases,
H.2.4 [Database Management]: Systems - Distributed databases, Parallel databases.

Avainsanat – Nyckelord – Keywords

avain-arvo-varasto, hajautetut tietokannat, pilvipalvelut, rinnakkaistietokannat, Web 2.0

Säilytyspaikka – Förvaringställe – Where deposited

Muita tietoja – Övriga uppgifter – Additional information

Sisältö

1 Johdanto 1

2 Tiedon käsittelyn ja varastoinnin uudet haasteet ja vaatimukset 4

3 Hajautetun tietokantajärjestelmän ja rinnakkaiskäsittelyn toimintaperiaatteita5

3.1 Tietokoneryväs 6

3.2 Tiedon saatavuus ja tiedon toisintaminen....9

3.3 Tiedon osittaminen 10

3.4 Relaation rinnakkaiskäsittely ja rinnakkaisliitos 11

3.5 SQL-operaation laskenta ja optimointi....14

4 Tietokannan laatuun vaikuttavat ominaisuudet 15

4.1 Transaktioiden ACID-ominaisuudet....15

4.2 CAP-teoreema 16

4.3 BASE-oikeellisuusmalli 17

5 Relaatiotietokantajärjestelmät 18

5.1 Relaatiotietokantajärjestelmien käyttökohteet....18

5.2 Relaatiotietokantojen haasteet laajan mittakaavan ympäristössä 20

5.3 Kritiikkiä relaatiotietokantajärjestelmiä kohtaan....21

6 NoSQL-tietokantajärjestelmät 22

6.1 NoSQL käsitteenä ja ilmiönä....23

6.2 Loogiset ominaispiirteet 25

6.3 Fyysiset ominaispiirteet....28

6.4 NoSQL-tietokantajärjestelmien luokittelu....31

6.5 MapReduce-ohjelmointiparadigma 45

7 NoSQL-tietokantapalveluna 51

7.1 Pilvipalvelujen yleiskuvaus 51

7.2 Google App Engine 53

7.3 Resurssien hyödyntäminen 55

7.4 Google App Engine Datastore....58

7.5 Käyttömahdollisuuksien analysointi....70

8 Vertailua ja analysointia 73

8.1 MapReduce-ohjelmointiparadigman ja relaatiotietokantajärjestelmien lähestymistavat laajan mittakaavan analyyttiseen rinnakkaislaskentaan.........................74

8.2 NoSQL-tietokantajärjestelmien ja relaatiotietokantajärjestelmien vertailu 83

8.3 Useamman erilaisen tietokantajärjestelmän ja työkalun hyödyntäminen...............92

9 Yhteenveto 100

Lähteet 104

1 Johdanto

Tietojenkäsittelyn alalla on viimeaikoina kohistu paljon NoSQL-tietokantajärjestelmistä. Näistä järjestelmistä on kirjoitettu runsaasti alan lehdistössä [Cat10, Sto10] sekä Internetin blogikirjoituksissa [Aba10, CHU08]. Aiheen tiimoilta on myös järjestetty konferensseja eri puolilla maailmaa [NOS11, SDE11]. Kyseiseen paradigmaan tai lähestymistapaan pohjautuvia ratkaisuja ovat ottaneet 2000-luvun aikana käyttöön esimerkiksi Internetissä toimivat suuret yritykset, kuten Google, Facebook ja Amazon. Näiden yritysten tarjoamat palvelut, kuten Google-hakukone [Goo12l], Facebook [Fac12] ja Amazon-verkkokauppa [Ama12a] ovat miljoonien ihmisten käytettävissä ympäri maailmaa. Tämän kaltaiset palvelut edellyttävät valtavien hajautettujen tietomäärien käsittelyä ja varastoimista. Tiedon pitää olla hyvin saatavilla. Tietokantajärjestelmältä edellytetään myös hyvää suorituskykyä, vaikka tietoon kohdistuvien operaatioiden määrä vaihtelee. Suorituskyvyn ylläpitämiseksi järjestelmän täytyy skaalautua niin, että tarpeen vaatiessa järjestelmään voidaan lisätä enemmän resursseja. Lisäksi tietokannan rakenteen tulee olla joustava ja helposti muokattavissa.

Perinteiset relaatiotietokantajärjestelmät eivät ole enää riittäneet vastaamaan edellä mainittujen kaltaisten laajan mittakaavan Internet-sovellusten tarpeisiin. Näille perinteisille tietokantajärjestelmille onkin viime vuosina kehitetty muita vaihtoehtoja. Näitä järjestelmiä on alettu kutsua NoSQL-tietokantajärjestelmiksi. NoSQL-tietokannat ovat usein niin erikoistuneita, ettei relaatiomallia ja SQL-kyselykielen koko ilmaisuvoimaa tarvita tai voida käyttää. Näiden tietokantojen looginen tietomalli perustuu avain-arvo-pariin [Cat10]. Avain-arvo-pari muodostaa tietokannan tietoalkion, jossa tietokantaan varastoitu arvo on yksilöity indeksoitavan avaimen perusteella. Tietokannan skeema on taas usein hyvin joustava, tai tietokanta saattaa olla jopa kokonaan skeematon. Käytössä olevat funktiot ovatkin usein rajoittuneet yksittäisten tietoalkioiden lukemiseen ja päivittämiseen. Näiden rakenteettomien tietojen laajan mittakaavan rinnakkaiseen laskentaan on lisäksi kehitetty yksinkertainen MapReduce-ohjelmointiparadigma [DeG08].

NoSQL-tietokantajärjestelmien ominaispiirteisiin kuuluu myös, että ne eivät noudata kaikkia tietokannan laadun takaavia transaktioiden ACID-ominaisuuksia [SKS11, s. 866]. Näistä periaatteista luopumalla tähdätään nimenomaan järjestelmän parempaan skaalautuvuuteen, saatavuuteen ja suorituskykyyn. Näiden tavoitteiden saavuttamiseksi

NoSQL-tietokantajärjestelmissä on jouduttu tekemään toimintaan vaikuttavia kompromisseja. Kyseisten järjestelmien tietokannan transaktio-, taululiitos- ja indeksointikäytännöt ovat usein rajoittuneita verrattuna perinteisiin relaatiotietokantajärjestelmiin.

Google [CDG06, CDG08] ja Amazon [DHJ07] julkaisivat 2000-luvulla artikkelit niiden omien palvelujensa perustaksi kehittämistä tietokantajärjestelmistä sekä niiden arkkitehtuuriratkaisuista. Google [Goo12b] ja Amazon [Ama12b] hyödyntävät näitä järjestelmiä varten rakentamaansa laajan mittakaavan infrastruktuuria tarjoamalla sitä myös muiden yritysten sovelluksien alustaksi palveluna. Näiden suurten Internet-yritysten esiteltyä omat NoSQL-tietokantajärjestelmänsä, uusia NoSQL-tietokantajärjestelmiä on kehitetty ja julkaistu yhä enemmän esimerkiksi avoimeen lähdekoodiin perustuvina projekteina [Tiw11]. Kehitteillä on myös uusia ohjelmointi- ja kyselykieliä [Goo12d, MeB11]. Nämä uudet tietokantajärjestelmät ja työkalut ovat herättäneet paljon kiinnostusta sekä sovellusohjelmoijien että eri kokoisten yritysten keskuudessa. Uudenlaisten tietokantajärjestelmien tarjonnan lisääntyessä voi kuitenkin olla vaikeaa hahmottaa niiden käyttötarkoitusta ja soveltuvuutta. Esille saattaa nousta kysymys miten sovellukset ja yritystoiminta voivat hyötyä NoSQL-paradigman soveltamisesta, vai soveltuvatko nämä järjestelmät kenties vain suurten Internet-yritysten tarpeisiin? Kuten edellä todettiin, NoSQL-tietokantajärjestelmät poikkeavat usein keskeisiltä ominaisuuksiltaan relaatiotietokantajärjestelmistä. Sen lisäksi nämä järjestelmät saattavat olla myös keskenään hyvin erilaisia. Tiettyjen yhteisten ominaispiirteiden lisäksi näiden uusien järjestelmien ominaisuudet voivat poiketa toisistaan suuresti.

Tässä tutkielmassa pyritään selventämään NoSQL-tietokantajärjestelmien tallennusratkaisun ja tiedon käsittelyn periaatteita, eroja relaatiotietokantajärjestelmiin sekä millaiseen käyttöön nämä tietokantajärjestelmät oikein soveltuvat. Tutkielmassa esitellään myös MapReduce-ohjelmointiparadigma [DeG08], NoSQL-tietokantapalveluna sekä joitakin NoSQL-tietokantajärjestelmien luokittelutapoja ja tietokannan tietomalleja. Tutkielmasta rajataan pois eräät relaatiomallista poikkeavat tietokannat, joita ei oltu alun perin kehitetty laajan mittakaavan Internet-sovellusten käyttöön, kuten XML-tietokannat ja perinteiset oliotietokannat. Tämän tutkielman tuloksen toivotaan selventävän kuvaa NoSQL-tietokantajärjestelmien käyttökohteista, ongelmista, puutteista sekä mahdollisesta soveltuvuudesta myös tavanomaisen yrityksen tiedonhallintaan. Tutkielma perustuu pääosin aikaisemmin aiheesta laadittuun kirjalliseen materiaaliin, kuten lehti-

ja konferenssiartikkeleihin sekä kirjoihin.

NoSQL-tietokantajärjestelmien tutkiminen on vaikeaa esimerkiksi, koska NoSQL-tietokantajärjestelmiksi kutsuttavia erilaisia toisistaan poikkeavia järjestelmiä on olemassa hyvin paljon [TuB11]. Vastaavia järjestelmiä myös kehitetään jatkuvasti lisää. Useassa lähdeartikkelissa vertaillaan muutamia eri tahojen kehittämiä eri tyyppisiä NoSQL-tietokantajärjestelmiä keskenään [Cat10, HEL11]. Tässä tutkielmassa sovelletaan kuitenkin hieman erilaista lähestymistapaa ja käytetään esimerkkeinä pääosin edellä mainittuja Googlen kehittämiä NoSQL-järjestelmiä. Muidenkin tahojen kehittämiä järjestelmiä kuitenkin hyödynnetään soveltuvissa kohdin. Osa näistä Googlen kehittämistä NoSQL-järjestelmistä hyödyntää toisiaan jollakin tavalla. Nämä Googlen järjestelmät ovat olleet myös muiden tahojen kehittämien useiden uudempien NoSQL-järjestelmien kehityksen innoittajina [LaM10, The12b].

Tämän tutkielman luvussa 2 käydään läpi tietojen käsittelyn ja varastoinnin uusia haasteita ja vaatimuksia. Luvussa 3 esitellään hajautetun tietokantajärjestelmän rakenteen ja toiminnan periaatteita sekä miten tietokannan tietoja voidaan käsitellä rinnakkain samaan aikaan. Luvussa 4 käydään läpi tietokannan laatuun vaikuttavia ominaisuuksia ja malleja. Luvussa 5 käsitellään relaatiotietokantajärjestelmien käyttökohteita, haasteita laajan mittakaavan hajautetussa ympäristössä sekä minkälaista kritiikkiä relaatiotietokantajärjestelmät ovat saaneet osakseen. Luvussa 6 käsitellään NoSQL-tietokantajärjestelmien syntyä ja kehitystä. Sen lisäksi tutkitaan näiden tietokantajärjestelmien ominaisuuksia ja toimintaperiaatteita sekä esitellään miten näitä tietokantajärjestelmiä on luokiteltu. Tässä luvussa esitellään myös yksinkertainen MapReduce-ohjelmointiparadigma. Luvussa 7 käsitellään NoSQL-tietokantapalveluna sekä esitellään Googlen vastaava palvelu. Lisäksi analysoidaan kyseisen palvelukonseptin käyttömahdollisuuksia. Luvussa 8 tarkastellaan MapReduce-ohjelmointiparadigman ja relaatiotietokantajärjestelmien toisistaan poikkeavia lähestymistapoja laajan mittakaavan analyyttiseen rinnakkaislaskentaan. Sen jälkeen vertaillaan NoSQL-tietokantajärjestelmiä relaatiotietokantajärjestelmiin. Luvussa 8 tarkastellaan myös useamman erilaisen tietokantajärjestelmän ja työkalun hyödyntämistä yhdessä. Työn lopuksi esitetään vielä yhteenveto tutkielman tärkeimmistä tuloksista sekä johtopäätöksistä.

2 Tiedon käsittelyn ja varastoinnin uudet haasteet ja vaatimukset

Digitaalisen tiedon määrä kasvaa koko ajan. Tietoa käsitellään ja varastoidaan yhä enemmän ja kiihtyvällä vauhdilla. Tähän on osaltaan vaikuttanut Internetin nopea kehitys 1990-luvulta lähtien laajaksi maailmanlaajuiseksi tietoverkoksi. Internetissä toimivia palveluja voivat käyttää miljoonat ihmiset ympäri maailmaa. Internetin toimintamalli ja käyttötavat ovat silti muuttuneet jo hyvin lyhyessä ajassa. Internetissä toimivat palvelut olivat alun perin pelkästään staattisia web-sivustoja, joihin käyttäjät eivät itse voineet paljon vaikuttaa. Nykyään monet tahot sen sijaan tarjoavat käyttäjilleen uusia käyttäjäkeskeisiä, interaktiivisia sekä sosiaalista verkostoitumista edistäviä palveluja, jotka toimivat Internetin välityksellä [Mur07]. Näitä palveluja ovat esimerkiksi monet yhteisöllisyyttä tarjoavat sovellukset, kuten Facebook [Fac12], Flickr [Yah12], ja YouTube [Goo12m]. Edellä mainittuja sovelluksia yhdistää usein se, että käyttäjät voivat myös itse tuottaa sisältöä palveluun. Käyttäjät voivat esimerkiksi tuottaa ja ladata tietoa palveluihin digitaalisessa muodossa, kuten blogikirjoituksia, tilapäivityksiä, linkkejä, kuvia, musiikkia ja videoita. Kaikki nämä lataukset lisäävät Internetissä olevaa digitaalisen tiedon määrää.

Näitä palveluja myös tarjotaan hyvin laajalle käyttäjäkunnalle, jopa maailman laajuisesti [Mur07]. Kyseisten palvelujen odotetaan olevan käytettävissä joka paikassa ja milloin tahansa. Tämän kaltaisia sovelluksia kutsutaan usein *Web 2.0 -sovelluksiksi* (Web 2.0 applications). Web 2.0 -käsite on kuitenkin laajempi kokonaisuus ja sisältää edellä mainittujen sovelluksien lisäksi teknologisia ratkaisuja, yritysten liiketoimintastrategioita sekä sosiaalisia trendejä. Web 2.0 -käsite on siis käyttötapaa ja teknologiaa koskeva paradigma, jolla usein tarkoitetaan Internetin toista kehitysvaihetta, jonka aikana Internetin koko toimintamalli kehittyi tiettyyn suuntaan. Web 2.0 tarkoittaa esimerkiksi joustavaa web-sivujen suunnittelua ja ylläpitoa, interaktiivisia käyttöliittymiä, mahdollisuutta tuottaa itse sisältöä ja yhdistellä tietoa eri lähteistä sekä sosiaalisia verkostoja.

Digitaalisen tiedon suuri ja koko ajan kasvava määrä sekä laaja eri puolille maailmaa sijoittuva käyttäjäkunta asettavat aivan uudenlaisia vaatimuksia myös tietojärjestelmille [SKS11, s. 862-863]. Useat Internetissä toimivat tunnetut palvelut, kuten Google-hakukone [Goo12l], Amazon-verkkokauppa [Ama12a], Facebook [Fac12] ja Twitter [Twi12] edellyttävätkin valtavien tietomäärien käsittelyä ja varastoimista. Google pal-

jasti vuonna 2009 käsittelevänsä jo peräti yli 20 petatavua tietoa päivässä [DeG08]. Facebookin mukaan taas käyttäjät olivat ladanneet vuoteen 2009 mennessä sen palveluun jo yli 15 miljardia valokuvaa. Facebook tallentaa neljä erikokoista versiota jokaisesta kuvasta. Nämä kaikki kuvaversiot veivät yhteensä jopa 1.5 petatavua levytilaa [Vaj09]. Twitter tuotti vuonna 2009 yli 12 teratavua varastoitavaa tietoa päivittäin [Wei10]. Tiedon tulee olla myös koko ajan saatavilla monessa eri paikassa. Amazon tarjoaa esimerkiksi maailmanlaajuisia lähestulkoon aina käytettävissä olevia web-palvelujaan jopa kymmenille miljoonille yhtäaikaisille käyttäjille, joille se pyrkii takaamaan palvelujen erittäin korkean toimintavarmuuden [DHJ07]. Tietojärjestelmän pitää siis olla suorituskykyinen, luotettava ja tehokas. Sen tulee myös mukautua kasvavaan tietomäärään. Edellä mainitut vaatimukset edellyttävät tietojärjestelmältä usein hajautettua arkkitehtuuriratkaisua. Tiedot varastoidaan yleensä ympäri maailmaa sijoitettuihin konesaleihin, joissa toimii tuhansia toisiinsa yhteydessä olevia tavanomaisia tietokoneita. Tällaista järjestelmää voidaan kutsua *hajautetuksi tietokantajärjestelmäksi* (distributed database system). Samaa tietoa *toisinnetaan* (replicate) usealle eri koneelle tiedon saatavuuden varmistamiseksi. Toisinnus tarkoittaa saman tiedon tallentamista moneen eri paikkaan, esimerkiksi tiedon saatavuuden varmistamiseksi. Näiden palvelujen taustalla toimivan tietokantajärjestelmän tulee myös *skaalautua* (scale) käyttöasteen mukaan eli tietokantaan kohdistuvien operaatioiden lisääntyessä kapasiteettia tulee voida lisätä. Lisäksi tiedon ja tietorakenteiden ominaisuuksiin voi tulla muutoksia, joten tietokannan rakenteen tulee olla joustava ja helposti muokattavissa.

3 Hajautetun tietokantajärjestelmän ja rinnakkaiskäsittelyn toimintaperiaatteita

Laajan mittakaavan Web 2.0 -sovellukset sekä massiiviset tietomäärät edellyttävät yleensä hajautetun tietokantaratkaisun käyttöä. Tietokannan tiedot tallennetaan useammalle eri tietokoneelle, jotka ovat yhteydessä toisiinsa nopean tietoverkon ja Internetin välityksellä. Tietokannan skaalautuvuus ja tehokkuuden säilyttäminen taas edellyttävät usein tietokantaoperaatioiden käsittelyä rinnakkain. Tämän tekee mahdolliseksi tietokoneiden prosessorien ja levyjen käyttäminen rinnakkain. Järjestelmää jonka suorituskykyä parannetaan rinnakkain suoritettavilla operaatioilla voidaan kutsua myös *rinnakkaisjärjestelmäksi* (parallel system) [SKS11, s. 777]. Tässä luvussa esitellään hajautetun

tietokantajärjestelmän rakennetta sekä toiminnan periaatteita. Lisäksi perehdytään miten tietokannan tietoja voidaan käsitellä rinnakkain samaan aikaan. Luvussa käsitellään tietokoneryväs, tiedon saatavuus ja toisintaminen, tiedon osittaminen sekä tietokannan relaation rinnakkaiskäsittely ja rinnakkaisliitos. Lopuksi katsotaan vielä miten SQL-kyselykielen avulla voidaan ohjelmoida ja optimoida tietokantaoperaatioita edellä kuvatun kaltaisessa järjestelmässä.

3.1 Tietokoneryväs

Tietokoneryväs (computer cluster) on joukko toisiinsa yhteydessä olevia tietokoneita jotka sijaitsevat jollain tietyllä samalla alueella, kuten esimerkiksi konesalissa [SKS11, s. 867]. Nämä tietokoneet voivat jakaa tehtäviä tai toimia toistensa sijalla, jos jokin niistä ei ole käytettävissä. Tietokonerypäitä kutsutaan usein hajautetun tietokantajärjestelmän *pisteiksi* (site) .

Hajautetun tietokannan tiedot on sijoitettu hajautetun tietokantajärjestelmän eri pisteiden tietokoneisiin. Tietojen sijoittaminen voi olla tilapäistä tai pysyvämmän luonteista [SKS11, s. 827]. Pisteisiin sijoitetut tiedot ovat yleensä pysyvämpiä. Pisteeseen pysyvästi voidaan sijoittaa sellaiset tiedot, joita esimerkiksi käytetään useammin juuri siinä pisteessä johon ne on sijoitettu. Verkon kautta liikuteltavaan tietoon liittyy aina kustannuksia, kuten *latenssi* (latency), joka tarkoittaa aikaa, joka kuluu paketin matkaan lähettäjältä vastaanottajalle. Latenssiin taas vaikuttaa verkon kaistanleveys. Näin ollen lähimmästä pisteestä löytyvät tiedot vähentävät verkon kautta tehtäviä hakuja verkon muihin pisteisiin ja sitä kautta tiedon siirrosta aiheutuvia latenssikustannuksia. Yksittäiselle rypään tietokoneelle tiedot sen sijaan saattaa olla sijoitettu vain tilapäisesti. Tietoja yksittäiseltä tietokoneelta toiselle voidaan siirtää esimerkiksi tietokoneiden kuormituksen tasaamiseksi [SKS11, s. 865]. Tietokoneet saattavat myös olla ajoittain poissa käytöstä tietokoneen kaatumisen tai verkkovian takia. Tämän takia niiden palveluja toisinnetaan eri koneille. Yksittäisen tietokoneen poistuessa käytöstä, sen tehtäviä voi alkaa hoitaa joku toinen vielä toiminnassa oleva rypään tietokone.

Tietokoneryväs on skaalautuva yksikkö eli tiedon ja operaatioiden lisääntyessä suorituskyky voidaan pitää ennallaan lisäämällä rypääseen uusia tietokoneita tarpeen mukaan. Nämä tietokoneet ovat usein tavanomaisia ja suhteellisen halpoja tietokoneita, jotka eivät välttämättä ole yhtä luotettavia kuin kalliimmat ja tehokkaammat palvelintie-

tokoneet [BDH03]. Ratkaisu perustuu siihen, että myös laadukkaammat tietokoneet kaatuvat joskus. Hajautetun tietokantajärjestelmän pitää siis joka tapauksessa varautua laitevikoihin. Kuten edellä mainittiin, tietoja toisinnetaan monelle eri koneelle esimerkiksi laitevikojen varalta. Tietokantajärjestelmän virhetilanteet pyritään myös löytämään ja käsittelemään mahdollisimman automaattisesti. Hajautetussa tietokantajärjestelmässä operaatioita voidaan lisäksi käsitellä aidosti rinnakkain useammalla eri tietokoneella samaan aikaan, joten yksittäisten tietokoneiden vasteajoilla ei ole suurta merkitystä. Tämä tietojen *rinnakkaiskäsittely* (parallel processing) tarkoittaa siis operaation jakamista osatehtäviin, jotka voidaan käsitellä samaan aikaan rinnakkain useammalla eri tietokoneella [SKS11, s. 401].

Tietokonejärjestelmät jotka suorittavat rinnakkaiskäsittelyä voidaan organisoida niiden yhteisten resurssien mukaan. Tietokoneryväs on yleensä organisoitu joko *yhteislevyjärjestelmäksi* (shared-disks system) tai *yksityislevyjärjestelmäksi* (shared-nothing system) [SKS11, s. 781]. Yhteislevyarkkitehtuurissa jokaisella tietoja käsittelevillä yksiköllä eli prosessorilla on oma yksityinen keskusmuisti, mutta se käyttää kaikille yhteisiä levyjä [SKS11, s. 783, Rah93]. Jokaisella prosessorilla on näin ollen pääsy erityisen *kytkentäverkon* (interconnection network) välityksellä ulkopuolisiin levyihin, joihin tietokannan tiedot on varastoitu. Tässä arkkitehtuuriratkaisussa tietokannan operaatioiden suorittaminen rinnakkain saattaa kuitenkin olla monimutkaista [SKS11, s. 802]. Yksittäinen tietokannan sivu voi olla samaan aikaan useamman prosessorin puskurissa, koska jokainen prosessori puskuroi tietokannan sivuja omassa puskurissaan. Kun prosessori toteuttaa tietokantaoperaatioita eli sillä oleva tietokannan transaktio haluaa käyttää prosessorin puskuroitua sivua, on varmistuttava siitä että sivu on sen viimeisin versio. Tätä kutsutaan myös *välimuistin yhteneväisyysongelmaksi* (cache-coherency problem). Prosessorien on lisäksi tehtävä tiettyjä tietokannan luotettavuuden varmistavia toimenpiteitä. Eri prosessoreilla olevat transaktiot varaavat esimerkiksi lukkoja samaan tietokantaan. Transaktiot myös kirjataan yhteiseen transaktiolokiin. Tietoja käsittelevien prosessorien on koordinoitava toimintojaan, jotta tietokannan luotettavuus säilyisi. Koordinointi edellyttää viestien välitystä eli kommunikointia eri prosessorien välillä. Kommunikointi voi olla kuitenkin hitaampaa kuin esimerkiksi käytettäessä prosessorien välistä yhteistä muistia, koska prosessorit liikennöivät toistensa välillä kytkentäverkon kautta [SKS11, s. 783]. Yhteislevyjärjestelmän ongelmaksi voi myös muodostua yhteys yhteisiin levyihin, jos tietokantajärjestelmä suorittaa paljon levyoperaatioita. Yhteys levyihin voi hi-

dastua liikenteen lisääntyessä, koska prosessorit liikennöivät yhteisiin levyihin kytkentäverkon kautta. Yhteislevyjärjestelmä on kuitenkin *vikasietoinen* (fault tolerant) eli se kestää hyvin prosessori- tai muistivikoja. Jos joku prosessori tai sen muisti lakkaa toimimasta, muut prosessorit voivat alkaa hoitaa sen tehtäviä.

Yksityislevyarkkitehtuurissa tietoja käsittelevät yksiköt toimivat itsenäisinä *solmuina* (node), joilla kullakin on oma prosessori, muisti ja yksi tai useampia levyjä [SKS11, s. 783, Rah93]. Jokainen solmu toimii sen levylle tai levyille sijoitettujen tietojen palvelimena. Solmut on kytketty toisiinsa nopean kytkentäverkon kautta. Solmun paikallisille levyille kohdistuvat tietokantaoperaatiot ovat hyvin nopeita, koska niiden ei tarvitse kulkea kytkentäverkon kautta. Tietokantaoperaatiot ja niiden tulokset jotka eivät koske solmun omille levyille tallennettuja tietoja, välitetään silti kytkentäverkon kautta. Solmujen välinen kommunikointi edellyttää myös molempien osallisena olevan solmun toimintaa ohjaavan järjestelmän vuorovaikutusta. Yksityislevyjärjestelmässä tietojen saanti muilta kuin solmun omilta levyiltä ja kommunikointi solmujen välillä on siis hitaampaa kuin yhteislevyjärjestelmässä. Yksityislevyjärjestelmien kytkentäverkko on kuitenkin usein suunniteltu skaalautuvaksi. Kytkentäverkon välityskapasiteettia voidaan lisätä sitä mukaan kun siihen liitetään lisää solmuja. Tällä tavoin yksityislevyjärjestelmään voidaan liittää suuri määrä solmuja suorituskyvyn siitä kärsimättä. Yksityislevyjärjestelmä onkin yhteislevyjärjestelmää skaalautuvampi.

Tässä arkkitehtuuriratkaisussa tietokannan operaatioiden suorittaminen rinnakkain ei ole välttämättä kaikilta osin yhtä monimutkaista kuin yhteislevyarkkitehtuurissa. Tietokannan eheys voidaan varmistaa yksinkertaisemmin ja tehokkaammin ainakin paikallisten tietokannan transaktioiden osalta. Jokainen solmu on nimittäin mahdollisimman itsenäinen yksikkö. Jokainen solmu puskuroi vain omien levyjensä tietokantasivuja, joten välimuistin yhteneväisyydestä ei tarvitse varmistua yhteislevyjärjestelmän tapaan [Rah93, SKS11, luvut 17 ja 18]. Jokaiseen solmuun on myös sijoitettu transaktioiden hallinnan edellyttämät lokitiedostot sekä katoavat keskusmuistirakenteet, kuten lukkotaulu ja tietokantapuskuri. Paikallisten transaktioiden hallinta ei siis edellytä kommunikointia eri yksiköiden välillä samaan tapaan kuin yhteislevyjärjestelmässä. Mikäli nämä transaktiot voivat sen sijaan kohdistua useammassa solmussa sijaitseviin tietoihin, niitä joudutaan koordinoimaan samaan tapaan kuten hajautettuja transaktioita yleensä koordinoidaan hajautetun tietokantajärjestelmän eri pisteiden välillä.

3.2 Tiedon saatavuus ja tiedon toisintaminen

Sovellukset jotka hyödyntävät hajautettua tietokantajärjestelmää edellyttävät usein, että tietokantajärjestelmä on aina käytettävissä. Hajautetussa ympäristössä tämä on kuitenkin erityisen haasteellista. Hajautetun tietokantajärjestelmän pisteet ja solmut eivät välttämättä ole aina saatavilla erilaisten verkko- ja laitevikojen takia. Laajan mittakaavan ympäristö on lisäksi usein levittäytynyt ympäri maailmaa ja sen piirissä saattaa olla tuhansia tietokoneita ja verkkolaitteita, joten erilaiset viat ovat yleensä hyvin todennäköisiä. Hajautetun tietokantajärjestelmän pisteet ja solmut toisiinsa kytkevän verkkolaitteen katkos saattaa esimerkiksi eristää järjestelmän pisteet eri osiin, kuten aliverkkoihin jotka eivät voi enää kommunikoida toistensa kanssa [SKS11, s. 847]. Tätä kutsutaan nimellä *verkon pirstoutuminen* (network partition). Hajautetun tietokantajärjestelmän pitää siis osata löytää ja korjata erilaisia virheitä, sekä toipua niistä ja jatkaa toimintaansa mahdollisimman automaattisesti. Tällaista hajautetun tietokantajärjestelmän vikasietoisuutta ja kykyä jatkaa toimintaansa kutsutaan tietokantajärjestelmän *kestävyydeksi* (robustness).

Hajautetun tietokantajärjestelmän tietokannan tiedoista on usein varastoitu kopio useampaan kuin yhteen solmuun tai pisteeseen. Tätä varten tietokannan relaatio toisinnetaan eli relaation kopio tallennetaan kahteen tai useampaan solmuun tai pisteeseen [SKS11, s. 826-827]. Tietokannasta on mahdollista tehdä myös *kokonaan toisinnettu* (full replication), jolloin relaation kopio on tallennettu hajautetun tietokantajärjestelmän jokaiseen solmuun tai pisteeseen. Tämän menettelyn tarkoituksena on parantaa tietokannan saatavuutta. Jos joku solmu tai piste ei ole enää käytettävissä, relaation tietoja voidaan edelleen hyödyntää sen solmun tai pisteen kautta jonne relaation *toisinne* (replica) on tallennettu. Tämän lisäksi tiedon saaminen lähimmästä toisinteesta vähentää verkon kautta liikuteltavasta tiedosta aiheutuvia latenssikustannuksia. Hajautetun tietokantajärjestelmän toimintaa voidaan tehostaa, esimerkiksi suorittamalla tietokannan lukuoperaatioita samaan aikaan rinnakkain saman relaation eri solmuihin tai pisteisiin tallennetuissa toisinteissa. Jos lukuoperaation tulos saadaan mahdollisimman läheltä operaation suorittavaa pistettä, se yleensä vähentää liikennettä muihin pisteisiin ja sitä kautta edellä mainittuja latenssikustannuksia.

Tietokantajärjestelmän pitää myös varmistaa, että kaikki tietokannan relaation toisinteet ovat oikeellisia alkuperäisen relaation kanssa. Jos alkuperäistä relaatiota päivitetään,

niin kaikki päivitykset pitää myös *levittää* (propagate) relaation kaikkiin toisinteisiin [SKS11, s. 826-827]. Levittäminen tarkoittaa ajan tasalla olevien tietojen kopioimista ja päivittämistä tietokantajärjestelmän kaikille osapuolille. Hajautetussa tietokantajärjestelmässä tämä saattaa kuitenkin olla monimutkaista ja tehotonta. Hajautettujen tietokannan transaktioiden koordinointi on monimutkaisempaa kuin keskitetyssä tietokantajärjestelmässä. Transaktioiden samanaikaisuudenhallinta- ja sitoutumismenetelmien soveltaminen eri pisteisiin sijoitettuihin tietoihin voi olla hankalaa. Tarve liikennöidä useaan eri pisteeseen taas kasvattaa esimerkiksi operaation latenssikustannuksia, joka hidastaa operaation suoritusta. Vaikka pisteet ovat saavutettavissa, ne voivat olla hitaan yhteyden takana.

3.3 Tiedon osittaminen

Tietokannan relaatio voidaan *hajauttaa* (decluster) *osittamalla* (partition) se useammalle levylle [SKS11, s. 798-799, 827]. Ositus tarkoittaa tietokannan relaation jakamista pienempiin osiin useammalle eri levylle jollain tiedonositusmenetelmällä. Näitä relaation osarelaatioita kutsutaan usein myös *paloiksi* (fragment tai tablet). Relaation paloissa tulee olla kaikki tarvittava tieto, jolla alkuperäinen relaatio voidaan mahdollisesti rakentaa uudelleen. Hajautetuissa ja rinnakkaistietokannoissa tiedonositusmenelmänä käytetään yleisimmin *vaakasuoraa ositusta* (horizontal partitioning). Vaakasuorassa osituksessa relaatio hajautetaan useamman levyn kesken niin, että jokainen relaation monikko sijaitsee vain yhdellä levyllä. Relaation r monikot hajautetaan n:lle levylle D_0, D_1…, D_{n-1} yhtä monessa osarelaatiossa r_0, r_1,…, r_{n-1} niin, että jokaisen osarelaation r_i monikot sijoitetaan vastaavalle levylle D_i, kun jokainen osarelaatio r_i on alkuperäisen relaation r osa ja osarelaatiot ovat toisistaan erillisiä.

Vaakasuora ositus voidaan toteuttaa *kiertovuoro-osituksella* (round-robin), *hajautusosituksella* (hash partitioning) tai *osaväliosituksella* (range partitioning) [SKS11, s. 798-802]. Kiertovuoro-osituksessa tietokannan relaation monikot sijoitetaan tasaisesti riippumatta niiden tietosisällöstä kaikille tietokannan eri levyille eli jokainen levy saa suurin piirtein saman määrän monikoita. Hajautusosituksessa taas käytetään jotain tiettyä tai vaihtoehtoisesti useampaa relaation attribuuttia hajautusfunktiossa ositusattribuuttina, jonka perusteella monikot sijoitetaan tai hajautetaan mahdollisimman tasaisesti tietokannan eri levyille. Vastaavasti osaväliosituksessa monikot sijoitetaan levyille joltain

ositusattribuutin arvoväliltä. Näin jokaiselle levylle tallennetaan vain tietyn arvovälin monikot. Yleensä osituksen tavoitteena on jakaa monikot mahdollisimman tasaisesti tietokannan levyille. Osaväliosituksen ongelmana on *vinouma* (skew), joka johtuu siitä, että jonkun relaation osavälien koko saattaa poiketa toisistaan jolloin monikot jakautuvat kyseisen relaation osalta epätasaisesti levyille. Sama ongelma saattaa esiintyä myös hajautusosituksessa, jos hajautusattribuutti esiintyy relaatiossa hyvin monta kertaa.

Ositettujen tietokannan relaatioiden palojen hajauttaminen tekee myös mahdolliseksi hajautetun tietokantajärjestelmän pisteiden ja solmujen kuorman tasauksen. Paloja voidaan tarvittaessa siirtää enemmän kuormitetuilta solmuilta vähemmän kuormitetuille solmuille [CDG06, CDG08]. Alkuperäisessä relaatiossa peräkkäin sijoittuneet monikot päätyvät hyvin suurella todennäköisyydellä hajautuksessa hakuavaimen perusteella tehdyn vaakasuoran osituksen tuloksena samaan tai muutamaan palaan. Pientä riviväliä koskevat lukuoperaatiot voidaan siis kohdistaa tehokkaasti vain muutamaan palaan, jotka on todennäköisesti tallennettu vain muutamaan rypään solmuun. Samalla tavalla ositusta voidaan hyödyntää jonkun tietyn relaation tietoja koskevan ositusattribuutin, kuten esimerkiksi sijaintipaikkakunnan osalta. Jos relaatio ositetaan sijaintipaikkakunnan perusteella, kyseistä paikkakuntaa koskevat tietokantakyselyt voidaan kohdistaa tehokkaasti vain tiettyihin paloihin.

Tietokannan *sirpalointi* (sharding) tarkoittaa tietokannan jakamista pienempiin osiin hajauttamalla ja osittamalla tietokannan relaatioita [BDH03, Cod12, IMS10]. Tavoitteena on sijoittaa ne tiedot, joihin kohdistuu tietokantaoperaatioita usein samaan aikaan, lähekkäin toisiaan samaan tai muutamaan palaan. Tämän lisäksi tavoitteena on relaation hajauttaminen eri tietokantapalvelimille tai fyysiseen sijaintiin niin, että paloihin kohdistuvat tietokantakyselyt kohdistuvat mahdollisimman tasaisesti jokaiselle palvelimelle. Sirpaloinnissa relaation osituksessa muodostettu pala voi muodostaa osan *sirpaleesta* (shard). Sirpale taas voi muodostua useammasta samaan paikkaan tallennetusta palasta. Hajautetuissa tietokantajärjestelmissä tietokannan sirpalointi voidaan toteuttaa hajauttamalla tietokannan relaatio eri pisteisiin ja yksityislevyjärjestelmän solmuihin.

3.4 Relaation rinnakkaiskäsittely ja rinnakkaisliitos

Tietokannan relaatioiden osittaminen mahdollistaa tietokannan tietojen rinnakkaiskäsittelyn. Relaation monikoita voidaan käsitellä samaan aikaan rinnakkain kaikilla niillä

levyillä D_i, joille on hajautettu relaation r monikoita sisältäviä paloja r_i [SKS11, s. 798-799]. Vastaavasti kun relaatio hajautetaan osittamalla, relaation r palaset r_i voidaan kirjoittaa levyille D_i samaan aikaan rinnakkain. Relaatioon r kohdistuvaa tietokantaoperaatiota voidaan nopeuttaa jakamalla se rinnakkain suoritettaviksi osaoperaatioiksi, jotka suoritetaan samaan aikaan osarelaatioissa r_i [SKS11, s. 804-814]. Tästä käytetään termiä *operaationsisäinen rinnakkaisuus* (intraoperation parallelism). Tietokantakysely on mahdollista rinnakkaistaa, riippuen sen sisältämistä tietokantaoperaatioista. Yksittäinen operaatio, kuten järjestäminen, valinta, projektio ja liitos voidaan rinnakkaistaa jakamalla se osaoperaatioiksi, joiden tulokset voidaan helposti yhdistää koko operaation tulokseksi. Vastaavasti tietokantakyselyn *laskentastrategian* (execution plan) sisältämä useampi eri operaatio voidaan rinnakkaistaa. Laskentastrategia koostuu kaikista yksittäisistä toisilleen tietoa välittävistä tehtävistä ja niiden oikeasta suoritusjärjestyksestä, joiden kaikkien avulla koko operaation tulos saadaan laskettua [SKS11, s. 814-815]. Laskentastrategian mahdollisesti sisältämät toisistaan riippumattomat eri operaatiot on mahdollista laskea rinnakkain [SKS11, s. 804-814]. Tästä käytetään sanontaa *operaatioiden välinen rinnakkaisuus* (interoperation parallelism). Jos taas operaatiot ovat toisistaan riippuvaisia, toisen operaation tulos voidaan *putkittaa* (pipeline) toisen operaation syötteeksi.

Rinnakkaiskyselyjen tehokkuuden taso riippuu tietokannan relaation osittamiseen käytetystä menetelmästä. Yksinkertainen koko relaation läpi käyvä *tauluselaus* (table scan) esimerkiksi käsitellään rinnakkain lukemalla relaation kaikki palaset rinnakkain [SKS11, s. 799-800]. Tämän kaltaiset operaatiot ovat tehokkaita kiertovuoro-ositetuissa relaatioissa, koska relaation monikot ovat jakautuneet tasaisesti kaikille levyille. Jonkun attribuutin perusteella tiettyyn monikkoon kohdistuvat *pistekyselyt* (point query) taas käsitellään kohdistamalla kysely siihen relaation palaseen, joka sisältää osavälin johon arvo kuuluu tai jolle hajautusfunktio kuvaa haettavan arvon. Jos relaatioita ei ole ositettu osaväleittäin tai hajauttamalla, relaation kaikki palaset joudutaan kuitenkin lukemaan rinnakkain läpi. *Osavälikyselyjen* (range query) käsittely riippuu siitä onko relaatio ositettu sen attribuutin perusteella, jonka osaväliä kysellään. Jos näin on, ja relaatio on osaväliositettu, kysely voidaan toteuttaa lukemalla rinnakkain kaikki ne relaation palaset, jotka leikkaavat kyseistä osaväliä. Jos relaatio on ositettu jonkun muun attribuutin mukaan, relaation kaikki palaset joudutaan lukemaan taas rinnakkain läpi.

Tietokannan relaatioiden liitoksien laskenta voidaan myös rinnakkaistaa. Tämä onnistuu, jos kyseessä on yhtäläisyys- tai luonnollinen liitos. Liitos voidaan laskea *ositettuna rinnakkaisliitoksena* (partitioned parallel join) [SKS11, s. 806-811]. Molemmat liitettävät relaatiot ositetaan hajautusosituksella samalla hajautusfunktiolla eri prosessoreille. Jokainen prosessori saa osituksen tuloksena palan molemmista relaatioista, joiden liitoksen se voi laskea itsenäisesti jollakin tavanomaisella liitosmenetelmällä. Kaikki prosessorit voivat näin laskea omien palojensa liitokset samaan aikaan rinnakkain. Lopullinen tulos saadaan aikaan yhdistämällä operaation lopuksi kaikkien prosessorien tulokset.

Tietokantajärjestelmän kapasiteettia voidaan lisätä lisäämällä tietokannan tietokantaoperaatioiden rinnakkaiskäsittelyä. Tietokantaoperaatioiden rinnakkaiskäsittelyn tavoitteena on yleensä tietokantajärjestelmän suorituskyvyn säilyttäminen, vaikka tietokannan koko ja operaatioiden lukumäärä kasvaisi. Tietokantaoperaatioiden laskentaa rinnakkaistamalla voidaan vaikuttaa tietokantaoperaatioiden kustannuksiin jakamalla työmäärää suoritettavaksi samaan aikaan useammalle eri taholle, kuten eri levyille tai prosessoreille [SKS11, s. 779-780]. Operaatioihin kuluvaa aikaa voidaan tämän ansiosta vähentää. Rinnakkaislaskennasta syntyy kuitenkin myös lisäkustannuksia, jotka täytyy ottaa huomioon operaatioiden kokonaiskustannuksia laskettaessa. Operaatioiden käynnistäminen usealla eri prosessorilla aiheuttaa *käynnistyskustannuksia* (startup cost). Käynnistyskustannukset voivat olla merkittävän suuria isoissa, jopa tuhansien operaatioiden rinnakkaisoperaatioissa. Useiden eri osaoperaatioiden *kilpailu resursseista* (interference) saattaa myös aiheuttaa viivästyksiä. Rinnakkain suoritettavat prosessit kilpailevat usein samoista jaetuista resursseista, kuten väyläohjain, jaetut levyt tai tietokannan lukot. Nämä jaetut resurssit saattavat olla jo jonkun toisen prosessin käytössä, joten resurssin vapautumista joudutaan odottamaan. Työkuorman jakelussa saattaa lisäksi esiintyä vinoumaa, joka aiheuttaa enemmän työmäärää joillekin prosessoreille. Tasaisesti jakautunut työkuorma voidaan laskea tehokkaammin rinnakkain, joten mahdollinen vinouma vaikuttaa negatiivisesti rinnakkaisen laskennan tehokkuuteen. Jos taas laskennan lopuksi pitää koota lopullinen tulos useammalta prosessorilta, se aiheuttaa *koontikustannuksia* (cost of assembling).

3.5 SQL-operaation laskenta ja optimointi

Tietokantaoperaatioita voidaan ohjelmoida hajautetussa tietokantajärjestelmässä SQL-kyselykielen avulla. Nämä operaatiot voivat hyödyntää myös rinnakkaislaskentaa. Tietokantajärjestelmän tietokantakyselynoptimoija valitsee jokaiselle operaatioille kulloinkin kokonaiskustannuksiltaan edullisimman laskentastrategian kaikista mahdollisista saman laskentatuloksen tuottavista strategioista [SKS11, s. 814-815]. Edullisimman laskentastrategia valinta on hajautetussa tietokantajärjestelmässä kuitenkin monimutkaisempaa kuin tavallisessa keskitetyssä tietokantajärjestelmässä. Valinnassa on otettava huomioon myös, miten tietokannan relaatiot on ositettu ja toisinnettu, sekä rinnakkaisoperaation kohdalla, miten rinnakkaislaskenta suoritetaan. Näitä rinnakkaislaskennan eri suoritustapoja käsiteltiin edellä. Rinnakkaislaskennasta voi aiheutua lisäkustannuksia, jotka pitää ottaa huomioon. Jotta operaatio voidaan laskea rinnakkain, pitää myös päättää millä järjestelmän resursseilla jokainen osaoperaatio lasketaan. Näiden resurssien määrittämistä kutsutaan myös *resursoinniksi* (scheduling). Kaikkia operaatioita ei kuitenkaan kannata laskea rinnakkain, jos laskentaan osallistuvien tahojen väliseen kommunikointiin kuluu enemmän kustannuksia kuin operaatioiden rinnakkain laskemisesta saadaan hyötyä.

Tietojen sijainti vaikuttaa hajautetussa tietokantajärjestelmässä merkittävästi tietokantaoperaation laskennan kustannuksiin. Tietojen siirtäminen solmusta toiseen lisää tiedon siirtämisestä aiheutuvia kustannuksia, kuten latenssia. Toisaalta operaation laskemisesta aiheutuu paikallisia kustannuksia kussakin solmussa, kuten levyoperaatioita [SKS11, s. 854-857]. Edullisimman strategian löytäminen voi olla siis hankalaa, jos operaatio kohdistuu tietokannan relaatioon, jonka monikot sijaitsevat useammassa kuin yhdessä palassa ja solmussa. Mikäli operaatio esimerkiksi vaatii tauluselauksen, alkuperäinen relaatio joudutaan rakentamaan kokonaan uudestaan kalliiden liitosoperaatioiden avulla. Kustannuksiin vaikuttaa myös, miten kalliiksi kukin tiedonsiirto tulee. Tähän vaikuttaa miten tietokantajärjestelmä on sirpaloitu ja ositettu, sekä onko saatavilla relaation toisinteita. Kyselynoptimoijan on siis löydettävä mahdollisimman edullinen balanssi näiden edellä mainittujen kustannuksien välillä, sekä päätettävä, mitä operaatioita kannattaa laskea solmussa paikallisesti ja mitä tietoja kannattaa siirtää eri solmujen välillä. Tietokantajärjestelmän fyysinen rakenne voi lisäksi vaikuttaa eri tavalla eri operaatioiden suorituskykyyn. Tietokannan sirpalointi, osittaminen ja indeksit on siis suunniteltava

niin, että erilaiset operaatiot voivat hyödyntää niitä mahdollisimman tehokkaasti.

4 Tietokannan laatuun vaikuttavat ominaisuudet

Tietokannan laatuun vaikuttavat tietyt ominaisuudet, joita tietokantajärjestelmä noudattaa toiminnassaan. Tietokannan luotettavuus voidaan taata esimerkiksi noudattamalla tietokannan transaktioiden ACID-ominaisuuksia. Hajautetut tietokantajärjestelmät eivät aina noudata kaikkia transaktioiden ACID-ominaisuuksia. Näistä ominaisuuksista luopumalla tähdätään parempaan skaalautuvuuteen, saatavuuteen ja suorituskykyyn. Tässä luvussa käydään läpi edellä mainitut ACID-ominaisuudet sekä esitellään joitakin vaihtoehtoisia malleja, joita usein käytetään hajautetun tietokannan laadun takaamiseen.

4.1 Transaktioiden ACID-ominaisuudet

Tietokannan transaktioiden ACID-ominaisuudet takaavat tietokannan luotettavuuden [SKS11, luku 14]. ACID-ominaisuudet koostuvat seuraavista neljästä ominaisuudesta:

- *atomisuus* (atomicity) (tai jakamattomuus),
- *oikeellisuus* (consistency) (tai eheys),
- *eristyvyys* (isolation) (tai erillisyys) ja
- *pysyvyys* (durability).

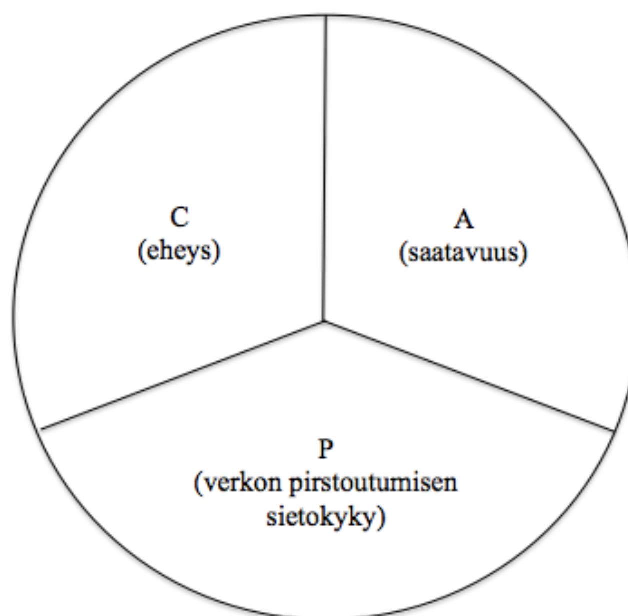
Atomisuus tarkoittaa, että kaikki tietokannan transaktion suorittamat tietokantaoperaatiot suoritetaan loppuun asti tai niitä ei suoriteta ollenkaan. Oikeellisuus taas tarkoittaa, että tietokannan tila säilyy eheänä transaktion suorituksen jälkeenkin. Eristyvyydellä tarkoitetaan, että muut tietokantatapahtumat eivät saa vaikuttaa yksittäisen transaktion suoritukseen. Transaktioiden eristyvyys taataan lukituksella, jossa transaktion sallitaan suorittaa tietokantaan kohdistuvan luku- tai kirjoitusoperaation vain, jos se on lukinnut operaation kohteen asianmukaisella lukolla. Pysyvyys taas tarkoittaa, että transaktion aikaansaamat muutokset tietokantaan ovat pysyviä.

Hajautettujen transaktioiden kohdalla käytetään usein *kaksivaiheista transaktioiden sitoutumiskäytäntöä, 2PC* (two-phase commit, 2PC) varmistamaan jokaisen hajautetun transaktion osapuolen ACID-ominaisuuksien mukainen luotettavuus ja atomisuus [SKS11, s. 832-836]. Käytännön toteutus tapahtuu nimensä mukaisesti kahdessa eri

vaiheessa. Ensimmäisessä vaiheessa transaktioita koordinoiva taho kysyy jokaiselta osapuolelta ovatko ne valmiita sitoutumaan. Jos kaikki osapuolet ovat valmiita sitoutumaan, koordinoiva taho pyytää toisessa vaiheessa jokaista osapuolta sitoutumaan. Jos joku osapuoli haluaa keskeyttää transaktion tai päätöstä ei saada määräajassa, koordinoiva taho keskeyttää ja peruuttaa koko transaktion. 2PC:n käyttöön liittyy kuitenkin merkittäviä kustannuksia. Transaktioon osallistuvien tahojen päätöksiä voidaan joutua odottamaan kauan aikaa. Esimerkiksi hajautetun tietokantajärjestelmän jokin piste ei ole välttämättä jostain syystä saatavilla. Sitä varten käytetään usein jotain tiettyä määräaikaa jossa vastaus on saatava tai transaktio keskeytetään.

4.2 CAP-teoreema

Californian Berkeley yliopiston professori Eric A. Brewer [Bre00] esitteli vuonna 2000 kehittämänsä CAP-teoreeman (CAP Theorem). CAP-teoreeman mukaan hajautettu tietojärjestelmä voi taata vain kaksi seuraavista ominaisuuksista: *eheys* (consistency), *saatavuus* (availability) ja *verkon pirstoutumisen sietokyky* (tolerance to network partitions). Kuvassa 1 CAP-teoreeman ominaisuudet on esitetty niiden englanninkielisistä nimistä johdettujen lyhenteiden C, A ja P avulla. Hajautetun tietokantajärjestelmän takaamat CAP-teoreeman mukaiset ominaisuudet esitetään yleensä näillä kirjainyhdistelmillä. Saatavuuden ja verkon pirstoutumisen sietokyvyn takaavalla järjestelmällä on esimerkiksi ominaisuudet AP.



Kuva 1: CAP-teoreeman esitys.

CAP-teoreemassa esitetyt ominaisuudet voidaan tulkita hajautettujen tietokantojen osalta seuraavalla tavalla. Eheyden saavuttamiseksi useaan eri pisteeseen kohdistuvat tietokantaoperaatiot tulee suorittaa loppuun saakka tai peruuttaa kokonaan [Sto10]. Tietokannan toisinteiden tulee myös olla eheitä. Tämän seurauksena kaikki hajautetun tietokantajärjestelmän solmut näkevät kaikki samat tiedot samaan aikaan. Saatavuus taas edellyttää, että tietokantajärjestelmän tulee pysyä aina toiminnassa eli kaikki normaalit luku- ja kirjoitusoperaatiot ovat mahdollisia virheistä huolimatta, esimerkiksi tietokannan toisinteiden avulla. Kolmas vaihtoehto eli verkon pirstoutumisen sietokyky tarkoittaa, että verkon pirstoutuessa tietojen käsittelyn tulee edelleen jatkua solmuissa.

4.3 BASE-oikeellisuusmalli

CAP-teoreemaa on usein käytetty perusteena tarpeelle tietokannan transaktioiden ACID-ominaisuuksia heikentävään luotettavuusmalliin [Pri08, Bre00]. Kuten edellä todettiin, CAP-teoreeman mukaan jaettu järjestelmä voi taata vain kaksi ominaisuuksista: eheys, saatavuus ja verkon pirstoutumisen sietokyky. ACID-ominaisuuksien mukaisen transaktioiden oikeellisuuden ja eristyvyyden takaaminen saattaa kuitenkin vaikuttaa negatiivisesti hajautetun tietokantajärjestelmän saatavuuteen. Kaksivaiheisessa transaktioiden sitoutumiskäytännössä saatavuuteen vaikuttaa esimerkiksi jokaisen transaktioon osallistuvan osapuolen saatavuus. Hajautetun tietokannan saatavuuteen voi myös vaikuttaa tietoalkioiden lukitseminen eristyvyyden takaamiseksi.

BASE-*oikeellisuusmallin* (BASE consistency model) avulla pyritään takaamaan hajautetulle järjestelmälle mahdollisimman korkea saatavuus ja tehokkuus [Pri08, Bre00]. Tämä saavutetaan heikentämällä tietokannan transaktioiden ACID-ominaisuuksista oikeellisuutta ja eristyvyyttä. BASE-oikeellisuusmallin ominaisuudet koostuvat seuraavista kolmesta eri ominaisuudesta:

- *periaatteessa käytettävissä* (basically available),
- *ei aina oikeellinen* (soft state) ja
- *lopulta oikeellinen* (eventually consistent).

BASE-mallia soveltava järjestelmä on periaatteessa käytettävissä, vaikka se ei ole aina oikeellinen. Järjestelmä päätyy silti oikeelliseen tilaan aina jossain vaiheessa. Tämä tarkoittaa, että tietokannan ei tarvitse olla oikeellisessa tilassa jokaisen transaktion jälkeen,

vaan riittää että tietokannan oikeellisuus saavutetaan jossain myöhemmässä vaiheessa. BASE-oikeellisuusmalli siis sallii tietokannan tietojen olevan jossain tilanteessa vanhentuneita. Edellä kuvattu menettely on transaktioiden ACID-ominaisuuksien vastainen.

5 Relaatiotietokantajärjestelmät

Relaatiotietokantajärjestelmät ovat yhä edelleen kaikkein yleisimmin käytettyjä tietokantajärjestelmiä. Kyseisiä järjestelmiä käytetään hyvin yleisesti kaiken kokoisten yritysten liiketoiminnan sovelluksissa. Tämän lisäksi relaationtietokantajärjestelmillä on paljon muitakin käyttökohteita. Laajan mittakaavan Web 2.0 -sovellukset ja massiiviset tietomäärät ovat kuitenkin asettaneet näille perinteisille tietokantajärjestelmille aivan uudenlaisia haasteita. Tässä luvussa käsitellään millaisiin tarkoituksiin relaatiotietokantajärjestelmiä on käytetty sekä millaisia haasteita laajan mittakaavan sovellukset niille asettavat. Luvun lopuksi vielä tutkitaan minkälaista kritiikkiä relaatiotietokantajärjestelmät ovat saaneet osakseen.

5.1 Relaatiotietokantajärjestelmien käyttökohteet

Alun perin E. F. Coddin kehittämään relaatiomalliin perustuvat relaatiotietokantajärjestelmät kehitettiin jo 1970-luvulla [VMZ10]. Sen jälkeen relaatiotietokantajärjestelmät ovat hallinneet kaupallisiin, akateemisiin ja tieteellisiin tarkoituksiin kehitettyjen tietokantajärjestelmien markkinoita aina näihin päiviin asti. Kaupallisia relaatiotietokantaohjelmien valmistajia ovat esimerkiksi Oracle [Ora12b], Microsoft [Mic12c] ja IBM [Ibm12b]. Avoimeen lähdekoodiin perustuvista relaatiotietokannan hallintajärjestelmistä tunnetuimpia ovat MySQL [Ora12e] ja PostgreSQL [Pos12]. Pienyrityksen ja henkilökohtaiseen käyttöön on kehitetty sen kaltaisia ohjelmia kuin Microsoft Access [Mic12b], kun taas suuren mittaluokan käyttöön on kehitetty tehokkailla tietokantapalvelimilla toimivia relaatiotietokannan hallintajärjestelmiä, kuten Oracle Database [Ora12a], Microsoft SQL Server [Mic12a] tai IBM DB2 [Ibm12a].

Relaatiotietokantajärjestelmien hyödylliset ominaisuudet ovat tehneet niistä suosittuja [Ler10]. Nämä järjestelmät ovat esimerkiksi suhteellisen vakaita ja turvallisia. Relaatiotietokantajärjestelmien tarjoamat tietokannan transaktioiden ACID-ominaisuudet pyrkivät takaamaan tietokannan luotettavuuden. Näiden järjestelmien tietokantatason tietoturvaominaisuudet ovat jokseenkin pitkälle kehittyneitä, esimerkiksi erillisien käyttä-

jänhallintatyökalujen ja hienorakeisien käyttö-oikeustasojen ansiosta. Relaatiotietokannan relaatioihin eli tauluihin on taas suhteellisen yksinkertaista mallintaa tietoja tavanomaisissa tapauksissa. Relaatiotietokantajärjestelmissä tietokantakyselyt kirjoitetaan SQL-kyselykielellä. SQL on korkeamman tason deklaratiivinen kyselykieli, joten sillä voidaan ilmaista yksinkertaisesti ja intuitiivisesti mitä halutaan tehdä sen sijaan, että joudutaan myös kertomaan miten kyseinen asia tehdään. SQL-kyselykielen intuitiivisen ilmaisukyvyn ansiosta SQL:n omaksuminen jonkun alemman tason ohjelmointikielen sijaan saattaa siis olla paljon helpompaa vähemmän ohjelmointikokemusta hallitsevalle. Näiden ominaisuuksien lisäksi relaatiotietokantajärjestelmiin on kehitetty uusia ominaisuuksia aina tarpeen vaatiessa. Näitä ovat esimerkiksi mahdollisuus tallentaa tietokantaan XML-muotoista tietoa sekä erilaiset analysointi- ja raportointiominaisuudet.

Relaatiotietokantajärjestelmät ovat osoittautuneet alusta lähtien toimiviksi ja luotettaviksi ratkaisuiksi erityisesti perinteisten *liiketoimintaan liittyvien tietojenkäsittelytoimintojen* (business data processing), kuten kirjanpidon, laskutuksen ja varastohallinnan sovelluksien käytössä [StC11]. Relaatiotietokantajärjestelmiä on silti kehitetty ja käytetty myös muihin tarkoituksiin. Näihin järjestelmiin on kehitetty uusia versioita, päivityksiä ja laajennososia, joiden avulla niiden käyttömahdollisuudet ovat entisestään laajentuneet. Relaatiotietokantajärjestelmiä on kehitetty esimerkiksi *tietovarastointiin* (data warehousing), tieteellisen tiedon käsittelyyn ja varastointiin sekä web-sivustojen, kuten sosiaalisen median ja pelisivustojen sovellusten käyttöön. Hajautettuun ja rinnakkaiseen laajan mittaluokan käyttöön on myös kehitetty relaatiotietokantajärjestelmiä. Oracle Real Application Clusters (RAC) [Ora12c] on esimerkiksi yhteislevyarkkitehtuuriin perustuva tietokantajärjestelmä, jonka avulla Oracle-relaatiotietokantaa [Ora12a] voidaan käyttää hajautetusti tietokonerypäissä. IBM DB2 Parallel Edition (PE) [BaF95] on taas yksityislevyarkkitehtuuriin perustuva, tietokantajärjestelmän loogisista solmuista koostuva relaatiomalliin perustuva rinnakkaistietokantajärjestelmä.

Tietokantajärjestelmiä hyödyntävät sovellukset voivat toimia monella eri tavalla, sekä vaatia järjestelmältä toisistaan poikkeavia ominaisuuksia. Tietokantajärjestelmiä hyödyntävät sovellukset voidaan jaotella esimerkiksi niiden suorittamien tietokantaoperaatioiden tyypin ja määrän mukaan. Stonebraker ja kumppanit [StC11] toteavat, että esimerkiksi tietovarastoinnin sovellukset ovat keskittyneet lukuoperaatioihin ja koostuvat enimmäkseen monimutkaisista tuhansien eri kohteiden luku- tai kirjoitusoperaatioista.

Liiketoiminnan operatiiviset tietojenkäsittelysovellukset, joita nykyään kutsutaan *OLTP –järjestelmiksi* (online transaction processing, OLTP), ovat taas keskittyneet tosiaikaisiin transaktioihin ja koostuvat enimmäkseen yksinkertaisista ja lyhyistä vain muutamien kohteiden luku- tai kirjoitusoperaatioista. Uusien Web 2.0 -sovellusten, kuten yhteisöpalvelujen sovelluksien operaatiot sijoittuvat edellä mainittujen sovellusten välimaastoon ja koostuvat enimmäkseen yksinkertaisista luku- ja kirjoitusoperaatioista. Stonebraker ja kumppanit [StC11] esittävät, että kaikkien merkittävien sekä kaupallisten että avoimeen lähdekoodiin perustuvien relaatiotietokantajärjestelmien toteutusten väitetään soveltuvan kaikkien eri operaatiotyyppejä ja määriä suorittavien sovellusten tarpeisiin. Stonebraker ja kumppanit [StC11] kutsuvat tämän kaltaisia järjestelmiä *yksi koko sopii kaikille -järjestelmiksi* (one-size-fits-all).

5.2 Relaatiotietokantojen haasteet laajan mittakaavan ympäristössä

Laajan mittakaavan Web 2.0 -sovellukset sekä massiiviset tietomäärät ovat asettaneet relaatiotietokantajärjestelmille isoja haasteita. Nämä haasteet liittyvät tehokkaaseen tietojenkäsittelyyn, tietokantaoperaatioiden rinnakkaistamiseen, järjestelmän skaalautuvuuteen sekä kustannuksiin [Tiw11]. Relaatiotietomalli edellyttää, että tietokantaan tallennettava tieto on rakenteellista, tiheää ja pääosin yhtenäistä. Tiedon ominaisuudet sekä tietokannan rakenne pitää olla mahdollista määritellä etukäteen. Eri tietokokonaisuuksiin liittyvien keskinäisten suhteiden tulee olla pysyviä. Relaatiotietokannan tietokokonaisuuksiin tulee näiden lisäksi olla mahdollista rakentaa oikeellisia indeksejä tietokantakyselyjen tehostamiseksi. Edellä mainitut vaatimukset eivät kuitenkaan välttämättä toteudu laajan mittakaavan Web 2.0 -sovelluksilla sekä massiivisilla tietomäärillä. Tieto on usein harvaa ja semirakenteellista tai jopa kokonaan rakenteetonta. Web 2.0 -sovellukset saattavat esimerkiksi usein vaatia paljon attribuutteja, joista vain osaa käytetään. Tiedon ja tietorakenteiden ominaisuuksiin voi myös tulla muutoksia jälkikäteen. Indeksien toteuttaminen ja ylläpito saattaa lisäksi olla hankalaa.

Relaatiotietokantajärjestelmä sijaitsee perinteisesti yhdellä tehokkaalla tietokantapalvelimella. Jos tarvitaan lisää tehoa, palvelimeen lisätään enemmän resursseja, kuten muistia, prosessoreja tai levyjä. Jos tämäkin kapasiteetti ylitetään, tarvitaan monta palvelinta joiden kesken tieto hajautetaan. Hajautetussa relaatiotietokantaympäristössä on usein monta erillistä tietokantapalvelinta, joille tiedot hajautetaan ja toisinnetaan. Hajautetun

tiedon hallinta on kuitenkin vaikeaa.

Laajan mittakaavan hajautetussa järjestelmässä relaatiomallin edellyttämien tietokannan taululiitosten toteuttaminen voi olla hankalaa. Taululiitosten laskenta voi olla tehotonta ja viedä liian kauan aikaa, jos käsiteltävänä on iso määrä ja paljon keskinäisiä liitoksia vaativia hajautettuja suuria tauluja [Lea10, VMZ10]. Semirakenteellisen tai rakenteettoman tiedon sovittaminen relaatiomalliin saattaa sen lisäksi tehdä tietokannan rakenteesta monimutkaisen ja sitä kautta tehottoman [Lea10]. Relaatioiden hajauttaminen ja tietokantaoperaatioiden rinnakkaislaskennan toteuttaminen voi myös olla vaikeaa, koska relaatiotietokantajärjestelmää ei oltu alun perin suunniteltu tiedon osittamista varten.

Relaatiotietokantajärjestelmät noudattavat operaatioissaan tietokannan transaktioiden ACID-ominaisuuksia. Kaikkien ACID-ominaisuuksien noudattaminen saattaa kuitenkin olla monimutkaista ja tehotonta, jos tiedot on varastoitu suurelle määrälle eri tietokoneita ja tavoitteena on tietokantajärjestelmän hyvä skaalautuvuus ja saatavuus [Aba10]. Tiedon oikeellisuutta on esimerkiksi hankalaa ylläpitää, jos samaa tietoa pitää ylläpitää monella eri koneella tiedon korkean saatavuuden varmistamiseksi.

5.3 Kritiikkiä relaatiotietokantajärjestelmiä kohtaan

Suosituimmat kaupalliset relaatiotietokannan hallintajärjestelmät perustuvat jo 25 vuotta sitten suunniteltuun arkkitehtuuriin [SMA07]. Oracle Database [Ora12a], Microsoft SQL Server [Mic12a] ja IBM DB2 [Ibm12a] ovat kaikki tavalla tai toisella saaneet vaikutteita *IBM System R -tietokantajärjestelmästä* [SMA07]. IBM System R on IBM:n [Ibm12b] jo 1970-luvulla kehittämä tietokantajärjestelmä [CAB81]. IBM System R oli samalla myös ensimmäinen merkittävä SQL-toteutus [SMA07]. IBM System R ei kuitenkaan ollut kaupallinen tietokannan hallintajärjestelmä. Kaupallisten relaatiotietokantaohjelmien nousu tapahtui varsinaisesti vasta myöhemmin. IBM System R:n jälkeen tietojenkäsittelytieteen alalla on tapahtunut huimaa kehitystä [SMA07]. Prosessorien nopeus sekä levyn ja muistin nopeus ja tallennuskapasiteetti ovat moninkertaistuneet. Tietokantajärjestelmien markkinat ovat myös kehittyneet. Ensimmäiset kaupalliset relaatiotietokantajärjestelmät oli kehitetty perinteisten liiketoimintaan liittyvien yritystoimintojen tarpeisiin. Tämän jälkeen tietokantajärjestelmien markkinat ovat kuitenkin kehittyneet ja rinnalle on tullut esimerkiksi *tietovarastoja* (data warehouse), *tekstin hallintaa* (text management), *tietovirtojen käsittelyä* (stream processing) sekä tieteellisiä

tietokantoja. 1970-luvulla tietokoneen ensisijainen käyttöliittymä oli yksinkertainen *terminaali-ikkuna* (terminal prompt), jonka kautta tietokoneelle annettiin käskyjä. Nykyään tietokannan operaatioita voidaan suorittaa Internetin ansiosta web-sivujen välityksellä. Relaatiotietokantajärjestelmiin on kehitetty vuosien varrella uusia ominaisuuksia vastaamaan näihin muuttuneisiin vaatimuksiin. Kyseisten järjestelmien arkkitehtuuri perustuu silti jossain määrin edelleen aivan eri aikakauden vaatimuksiin.

Vaikka kaupalliset relaatiotietokantajärjestelmät oli alun perin suunniteltu edellä mainittujen perinteisten yritystoimintojen tarpeisiin, on niitä sittemmin käytetty lähes kaikkiin tiedon käsittelyn ja varastoinnin tarpeisiin [HeJ11]. Näin on tehty vaikka kaikki varastoitavat tiedot eivät sovi kovin hyvin yhteen relaatiotietokantajärjestelmien soveltaman relaatiomallin kanssa. Stonebraker ja kumppanit [SMA07] esittävät artikkelissaan, että perinteiset kaupalliset relaatiotietokantajärjestelmät eivät välttämättä ole paras ratkaisu kaikkiin erilaisiin tiedon käsittelyn ja varastoinnin vaatimuksiin. Sen sijaan että yritetään ratkaista kaikkia uusia vaatimuksia vanhaan arkkitehtuuriin perustuvalla yksi koko sopii kaikille -tietokantajärjestelmällä, tulee kehittää kokonaan uusia vain tiettyihin tarkoituksiin erikoistuneita tietokantajärjestelmiä. Nämä uudet järjestelmät tulee kehittää täysin puhtaalta pöydältä ilman vanhaan arkkitehtuuriin perustuvia rasitteita, kuten esimerkiksi vanhaa koodia.

6 NoSQL-tietokantajärjestelmät

Laajan mittakaavan Web 2.0 -sovelluksien käyttöön on 2000-luvulla kehitetty uusia tietokantajärjestelmiä, joiden tietokannan tietomalli ei välttämättä perustu relaatiomalliin. Tässä luvussa käsitellään näiden tietokantajärjestelmien syntyä ja kehitystä. Sen lisäksi tutkitaan näiden tietokantajärjestelmien ominaisuuksia ja toimintaperiaatteita sekä miten näitä tietokantajärjestelmiä on luokiteltu. Google Bigtable -tietokantajärjestelmää [CDG06, CDG08] käytetään havainnollistavana esimerkkijärjestelmänä. Bigtable on vaikuttanut muiden NoSQL-tietokantajärjestelmien [LaM10, The12b, The12c, The12e] kehittämiseen. Tässä luvussa esitellään myös yksinkertainen MapReduce-ohjelmointiparadigma [DeG08], jonka avulla näissä tietokantajärjestelmissä voidaan hyödyntää laajan mittakaavan rinnakkaislaskentaa.

6.1 NoSQL käsitteenä ja ilmiönä

Perinteisten relaatiotietokantajärjestelmien saatavuus, skaalautuvuus ja suorituskyky on usein koettu riittämättömäksi laajan mittakaavan Internet-ympäristössä. Perinteisille relaatiotietokantajärjestelmille on viime vuosina kehitetty muita vaihtoehtoja. Näitä vaihtoehtoisia tietokantajärjestelmiä on kehitetty laajan mittakaavan Web 2.0 -sovelluksien tarpeita varten, ja niiden tietokannan tietomalli perustuu käsitteellisesti yksinkertaiseen avain-arvo-malliin. Avain-arvo-malli esitellään tarkemmin jatkossa. Näitä järjestelmiä on alettu kutsua ja luokitella *NoSQL-tietokantajärjestelmiksi* (NoSQL database systems). Näitä järjestelmiä saatetaan kutsua myös *NoSQL-tietokannoiksi* (NoSQL databases).

NoSQL-termiä käytettiin jo vuonna 1998. Carlo Strozzi [Str12, Pat99] kutsui kehittämäänsä avoimeen lähdekoodiin perustuvaa järjestelmää nimellä NoSQL. Kyseessä on kuitenkin relaatiotietokantajärjestelmä, jossa ei käytetä SQL:ää kyselykielenä. Termin NoSQL otti uudelleen käyttöön Eric Evans [Eva09, Ler10] vuonna 2009 järjestetyssä tapahtumassa, jossa käsiteltiin avoimeen lähdekoodiin perustuvia tietokantajärjestelmiä. NoSQL ei siis ole käsitteenä kaikkein paras mahdollinen käsitteen tulkinnanvaraisuuden vuoksi. Esimerkiksi on mahdollista tulkita, että NoSQL tarkoittaa *ei SQL* (no SQL) tai *ei SQL:lle* (no to SQL). Termistä käytetään joissain lähteissä tulkintaa *ei relaationaalinen* (non relational). Yleisesti käytetty määritelmä NoSQL-termille on kuitenkin, että NoSQL tarkoittaa *ei vain SQL* (not only SQL). Jollakin NoSQL-järjestelmäksi luokiteltavalla järjestelmällä saattaa nimittäin olla myös joitakin relaatiotietokantojen ominaisuuksia. Tällainen järjestelmä voi esimerkiksi hyödyntää myös SQL-kyselykieltä [ABK09, LLC12].

2000-luvun NoSQL-ilmiön syntyyn ovat vaikuttaneet vahvasti Internetin ja Web 2.0:n kehityksen lisäksi suuret Internetissä toimivat yritykset Google ja Amazon [Ama12a], jotka kehittivät omien palvelujensa perustaksi omat NoSQL-paradigmaan perustuvat tietokantajärjestelmänsä. Google Bigtable [CDG06, CDG08] on rakenteellisen tai puolirakenteellisen tiedon hallintaan ja varastointiin suunniteltu laajan mittakaavan hajautettu tietokantajärjestelmä. Bigtable on suunniteltu Googlen Internetissä toimivien laajan mittakaavan Web 2.0 -palvelujen perustaksi. Näitä tunnettuja palveluja ovat esimerkiksi Google-hakukone [Goo12l], YouTube [Goo12m], Google Earth [Goo12i] ja Google+ [Goo12n]. Google julkaisi vuonna 2006 sen omien työntekijöiden laatiman artikkelin

[CDG06, CDG08], jossa kuvaillaan Bigtablen rakennetta ja toimintaa. Bigtable-arkkitehtuurin julkistamisen jälkeen on kehitetty muita samankaltaiseen arkkitehtuuriin perustuvia tietokantajärjestelmiä [Tiw11]. Ensimmäisenä näiden uusien järjestelmien joukossa oli *Lucene* [The12d]. Lucene on Doug Cuttingin kehittämä avoimeen lähdekoodiin perustuva tiedonhakuun tarkoitettu ohjelmistokirjasto. Lucenen kehittäjät auttoivat tämän jälkeen Yahoo!ta [Yah12b] kehittämään *Hadoop-ohjelmistokehyksen* (Hadoop Software Framework), jonka avulla Google Bigtablea ja sen infrastruktuuria muistuttavassa hajautetussa ympäristössä voidaan suorittaa laajan mittakaavan rinnakkaislaskentaa [Tiw11]. Apache Hadoop [The12b] on Yahoo!n lisäksi ollut käytössä esimerkiksi Facebookilla [Fac12][BGS11].

NoSQL-tietokantajärjestelmien idea ja termi NoSQL nousivat jälleen pinnalle Hadoopin kehityksen yhteydessä [Tiw11]. Kiinnostus tämän kaltaisia järjestelmiä kohtaan kasvoi, kun myös Amazon päätti julkistaa oman uuden tietokantajärjestelmänsä arkkitehtuurin Googlen esimerkkiä seuraten. Amazon Dynamo [DHJ07] on Amazonin korkeaa saatavuutta edellyttävää Amazon-verkkokauppaa [Ama12a] varten kehitetty hajautettu tietokantajärjestelmä. Amazon julkaisi vuonna 2007 Dynamoa kuvaavan artikkelin [DHJ07]. Kumpikaan näiden isojen Internet-yritysten tietokantajärjestelmien tietokannoista ei siis perustunut relaatiomalliin. Näiden yritysten sekä niistä mallia ottaneiden muiden yritysten mahdollisesti relaatiomallista poikkeavat ratkaisut ovat herättäneet paljon kiinnostusta sovellusohjelmoijien sekä eri kokoisten yritysten keskuudessa. NoSQL-tietokantajärjestelmien tallennusratkaisun ja tiedon käsittelyn periaatteista sekä soveltuvuudesta eri käyttötarkoituksiin on haluttu saada enemmän tietoa. Lisäksi esille on noussut kysymys miten sovellukset ja yritystoiminta voivat hyötyä NoSQL-paradigmasta? NoSQL-tietokantajärjestelmien tietojenkäsittelyn ja varastoinnin toimintaperiaatteet ja ideat ovat levinneet laajalle hyvin lyhyessä ajassa. Pienempien yritysten lisäksi hyvin monet tunnetut yritykset, kuten Facebook [Fac12], Ebay [Eba12], IBM [Ibm12b] ovat kehittäneet NoSQL-paradigmaan perustuvia ohjelmien laajennoksia sekä aivan uusia ohjelmia [Git12, The12e, The12h, TSJ09]. Suuri osa näistä ohjelmista on julkaistu avoimena lähdekoodina.

6.2 Loogiset ominaispiirteet

NoSQL-tietokannat ovat usein niin erikoistuneita, ettei relaatiomallia ja SQL-kyselykielen koko ilmaisuvoimaa tarvita tai voida käyttää [PDG05]. Näiden tietokantojen looginen tietomalli perustuu avain-arvo-pariin [Cat10]. Avain-arvo-pari muodostaa tietokannan tietoalkion, jossa tietokantaan varastoitu rakenteellinen tai rakenteeton arvo on yksilöity indeksoitavan avaimen perusteella. Tätä mallia kutsutaan sen takia joskus myös *avain-arvo-malliksi* (key-value model) [Kat12]. Avain-arvo-malli perustuu siis avaimen ja arvon muodostamaan pariin, jossa yksilöllinen avain viittaa arvoon. Tästä käsitteellisesti yksinkertaisesta mallista johdettu tietokannan tietomalli saattaa kuitenkin olla monimutkaisempi. Esimerkiksi Google Bigtable -tietokannan [CDG06, CDG08] taulu koostuu riveistä, sarakkeista ja *aikaleimoista* (timestamp):

$$\text{(riviavain, sarakeavain, aikaleima)} \rightarrow \text{sisältö.}$$

Näiden yhdistelmä eli avain-arvo-mallin avain viittaa varsinaiseen tietosisältöön, joka on avain-arvo-mallin arvo. Samasta tiedosta voi olla varastoituna monta eri versiota, jotka erotellaan aikaleimojen perusteella. Riviavain voi olla esimerkiksi web-sivun käänteinen URL-osoite. Aikaleima voi taas olla esimerkiksi web-sivun indeksointihetken aika mikrosekunneissa. Bigtable-tietomallia käsitellään tarkemmin jatkossa.

NoSQL-tietokantajärjestelmien tietokannan skeema on usein hyvin joustava tai tietokanta saattaa olla kokonaan skeematon. Tietokannan rakennetta ei siis esimerkiksi tarvitse olla määritelty kokonaan etukäteen. Sovellusohjelmoija voi usein tehdä muutoksia skeemaan jälkikäteen, kuten lisätä tai poistaa tietokannan tauluja esimerkiksi *ohjelmointirajapinnan, API* (application programming interface, API) kautta. Ohjelmointirajapinta on lähdekoodiin perustuva määritelmä, jonka määrittelemän rajapinnan avulla ohjelmat voivat kommunikoida toistensa kanssa. Ohjelmointirajapinnan kautta myös sovellusohjelmoija voi suorittaa ohjelmaan liittyviä toimintoja. NoSQL-tietokantajärjestelmien ohjelmointirajapinnan kautta tietokannan käsittelyyn tarjottavat funktiot ovat kuitenkin usein rajoittuneet yksittäisten tietoalkioiden lukemiseen ja päivittämiseen [SKS11, s. 863]. Johonkin avaimeen liittyvä arvo voidaan varastoida tietokantaan esimerkiksi yksinkertaisella put(key, value) komennolla. Vastaavasti tietyllä avaimella varastoituun arvoon päästään käsiksi get(key, value) komennolla. Joissakin NoSQL-tietokannoissa arvoja voidaan hakea myös esimerkiksi avainten *arvovälillä* (range query). Sovellusoh-

jelmoijille tarjotaan yleensä hyvin pelkistetty ohjelmointirajapinta näiden yksinkertaisten tietokantaoperaatioiden suorittamiseen. Operaatiot voidaan yleensä ohjelmoida jollain yleisellä alemman tason proseduraalisella ohjelmointikielellä, kuten C++ tai Java. NoSQL-tietokannoille on myös kehitetty omia SQL:ää yksinkertaisempia ohjelmointi- ja kyselykieliä.

NoSQL-tietokantajärjestelmien tietokantaoperaatioiden laskentamenetelmät perustuvat hajautetun tietokantajärjestelmän pisteiden solmuissa rinnakkain suoritettaviin osaoperaatioihin, joiden tulokset lopuksi yhdistetään. Rinnakkaisessa laskennassa hyödynnetään samoja rinnakkaiskäsittelyn menetelmiä, joita käsiteltiin luvussa 3. NoSQL-tietokannan tietojen laajan mittakaavan rinnakkaislaskentaan voidaan usein hyödyntää yksinkertaista *MapReduce-ohjelmointiparadigmaa* (MapReduce programming paradigm) [DeG08]. MapReduce on ohjelmointimalli sekä siihen liittyvän kehyksen toteutus, jonka avulla voidaan suorittaa rinnakkaislaskentaa laajan mittakaavan hajautetussa ympäristössä. MapReduce-ohjelmointiparadigmaa käsitellään tarkemmin jatkossa. Bigtable-ohjelmointirajapinnan kautta sovellukset esimerkiksi voivat lukea, kirjoittaa ja poistaa taulun arvoja [CDG06, CDG08]. Ohjelmointirajapinnan kautta voidaan hakea suoraan yksittäisten taulujen arvoja rivi- ja sarakeavaimen sekä aikaleiman perusteella. Tietokantahaku voidaan kohdistaa myös johonkin määriteltyyn joukkoon. Aikaleimaa voidaan käyttää esimerkiksi vain tiettyyn aikaväliin sopivien tietojen hakemiseen. Haetaan esimerkiksi vain ne web-sivujen eri versiot, joiden aikaleima on viimeisen viikon sisällä. Hakua voidaan rajoittaa myös esimerkiksi *säännöllisen lausekkeen* (regular expression) avulla, jossa määritellään, mikä osa hakutuloksissa täsmää johonkin tiettyyn lausekkeessa määriteltyyn hakuehtoon. Haetaan esimerkiksi kaikki Ylen web-sivut fi.yle.www*. Edellä esitetty yksinkertainen ehto määrittelee, että hakutuloksissa pitää olla alkuosa fi.yle.www, mutta tähdellä merkitty loppuosa voi olla mikä tahansa. Ohjelmointirajapinnan kautta voidaan suorittaa myös erilaisia järjestelmän ylläpitotoimintoja, kuten edellä mainitut tietokannan skeemamuutokset sekä järjestelmän metatietojen, kuten käyttöoikeuksien muokkaaminen. Bigtable-taulu voi myös toimia MapReduce-ohjelmien syötetietojen lähteenä tai laskennan lopputuloksen kohteena.

Bigtable-tietokannan tietoja voidaan myös analysoida *Sawzall-ohjelmointikielellä* [PDG05]. Sawzall on Googlen kehittämä korkeamman tason proseduraalinen ohjelmointikieli isojen ja hajautettujen tietokokonaisuuksien analysointiin. Bigtablen käyttä-

jät voivat hyödyntää omia Sawzall-skriptejään [CDG06, CDG08]. Sawzall-ohjelmointikieltä voidaan käyttää esimerkiksi MapReduce-ohjelmien yhteydessä. Kielellä ei siis voi kirjoittaa tietoja suoraan Bigtable-tauluun, vaan sen käyttö rajoittuu tiedon muunnoksiin, suodatukseen sekä erilaisten yhteenvetojen laskentaan. Kieli on korkeamman tason ohjelmointikieli verrattuna johonkin alemman tason ohjelmointikieleen, kuten Java tai C++ [PDG05]. Sawzall on kuitenkin edelleen SQL-kyselykieltä yksinkertaisempi kieli. Sawzall ei ole SQL:n tapaan deklaratiivinen kyselykieli. Googlen asiakkailla on ollut tarvetta vielä Sawzall-ohjelmointikieltä ilmaisuvoimaisempaan ohjelmointi- tai kyselykieleen, koska yksinkertaisten kyselyjen kirjoittaminen Sawzall-kielellä saattaa edelleen olla kovin työlästä [CCF08].

NoSQL-tietokantajärjestelmät eivät yleensä noudata kaikkia tietokannan laadun takaavia tietokannan transaktioiden ACID-ominaisuuksia [SKS11, s. 866]. Näitä ominaisuuksia löyhentämällä tähdätään parempaan skaalautuvuuteen, saatavuuteen ja suorituskykyyn. Transaktioiden ACID-ominaisuuksien mukaista tietojen oikeellisuutta on löyhennetty korkean saatavuuden ja suorituskyvyn saavuttamiseksi. NoSQL-tietokantajärjestelmät noudattavat usein löyhempää BASE-oikeellisuusmallia jolloin tietokannan ei tarvitse olla aina oikeellinen vaan riittää, että tietokannan oikeellisuus saavutetaan jossain myöhemmässä vaiheessa. Tämä tarkoittaa, että kaikkien hajautetun tietokantajärjestelmän pisteiden tiedot eivät ole aina ajan tasalla. ACID-ominaisuudet voidaan silti taata jollekin tietylle tietokannan osalle. ACID-ominaisuudet voidaan esimerkiksi taata vain sirpale- tai rivikohtaisesti. NoSQL-tietokantajärjestelmät eivät usein voi ylläpitää toissijaisia oikeellisia tietokannan indeksejä, koska niitä saattaa olla hankalaa ja tehotonta ylläpitää kun tiedot on hajautettu ja ositettu jo avaimen perusteella. Toissijaisen indeksin muodostavaan attribuuttiin kohdistuvat lisäykset ja päivitykset pitää nimittäin indeksin päivittämistä varten levittää edelleen useampaan pisteeseen. Bigtablessa ei esimerkiksi voida suorittaa tavanomaisia transaktioita, jotka koskevat useaan eri riviavaimeen liittyviä tietoja [CDG06, CDG08]. Vain samaan riviavaimeen liittyviä tietoja koskevat transaktiot ovat *sarjallistuvia* (serializable). Peräkkäin suoritettavien transaktioiden ajoitus on sarjallistuva. Rinnakkain suoritettavien transaktioiden ajoitus on sarjallistuva, jos lopputulos eli tietokannan tila on transaktioiden suorituksen jälkeen sama, kuin jos samat transaktiot suoritettaisiin peräkkäin [SKS11, s. 641-646]. Bigtable tarjoaa kuitenkin käyttöliittymän, jonka kautta sovellusohjelmoija voi ohjelmoida monta eri rivejä koskevaa kirjoitusoperaatiota suoritettavaksi *eräajona* (batch operation) [CDG06, CDG08].

Eräajo on ohjelma jossa tietokone suorittaa monta operaatiota automaattisesti peräkkäin.

Kuvissa 2 ja 3 esitetään esimerkit C++ -kielellä toteutetuista kirjoitus- ja lukuoperaatioista. Kuvassa 2 tehdään saman rivin tietoihin lisäys ja poisto. Ensin avataan webtable-taulu. RowMutation-toiminto asettaa rivin fi.yle.www sarakkeen (ankkuri:hs.fi) arvoksi YLE. Sen jälkeen poistetaan toinen sarake (ankkuri:nelonen.fi). Lopuksi Apply-kutsu suorittaa edellä mainitut toiminnot atomisena operaationa. Kuvassa 3 Scanner-toiminto käy läpi rivin fi.yle.www kaikki ankkuri-sarakeperheen sarakkeet. Sarakeperheitä käsitellään tarkemmin jatkossa. Ohjelma tuottaa tuloksena rivin nimen, tietyntyyppisten sarakkeiden jokaisen sarakkeen nimen, aikaleiman ja arvon.

```
//Avaa taulu
Table *T = OpenOrDie("/bigtable/web/webtable");

//Kirjoita uusi ankkuri ja poista vanha
RowMutation r1(T, "fi.yle.www");
r1.Set("ankkuri:hs.fi", "YLE");
r1.Delete("ankkuri:nelonen.fi");
Operation op;
Apply(&op, &r1);
```

Kuva 2: Esimerkki kirjoitusoperaatiosta [CDG06, CDG08].

```
Scanner scanner(T);
ScanStream *stream;
stream = scanner.FetchColumnFamily("ankkuri");
stream->SetReturnAllVersions();
scanner.Lookup("fi.yle.www");
for (; !stream->Done(); stream->Next()) {
  printf("%s %s %lld %s/n",
         scanner.RowName(),
         stream->ColumnName(),
         stream->MicroTimestamp(),
         stream->Value());
  }
```

Kuva 3: Esimerkki lukuoperaatiosta [CDG06, CDG08].

6.3 Fyysiset ominaispiirteet

NoSQL-tietokantajärjestelmien ominaispiirteisiin kuuluu, että ne pyrkivät varmistamaan tietokantajärjestelmän hyvän saatavuuden skaalautumalla sirpaloimalla, osittamalla ja toisintamalla tietoa automaattisesti monelle eri koneelle. Näissä operaatioissa hyödynnetään samoja hajautettujen tietokantajärjestelmien menetelmiä, joita käsiteltiin luvussa 3. Ositus tapahtuu pääsääntöisesti vaakasuoraan osittamalla. Ositukseen käytettävää hajautusfunktiota ei usein ole mahdollista määritellä etukäteen [SKS11, s. 865]. Tämän takia tietokannan taulu ositetaan yleensä avaimen perusteella automaattisesti kun taulun

koko kasvaa tarpeeksi suureksi, suhteellisen pieniksi, vain muutamien satojen megatavujen taulujen paloiksi. Nämä ositetut palat hajautetaan ja varastoidaan hajautettuun tiedostojärjestelmään useista halvoista tietokoneista koostuvien tietokonerypäiden solmujen levyille. Nämä laajan mittakaavan tietokonerypäät ja niiden solmut on yleensä organisoitu yksityislevyjärjestelmään.

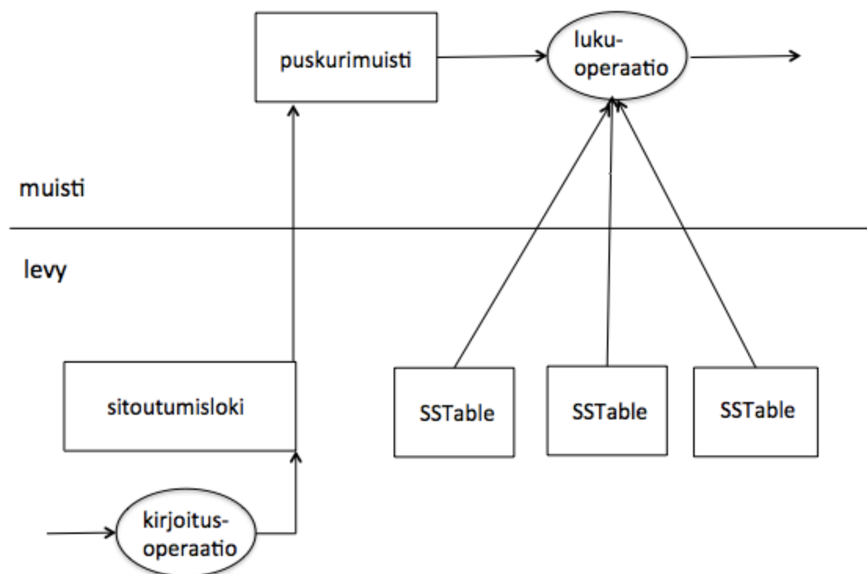
Google Bigtablen tietokannan tiedot esimerkiksi tallennetaan fyysisesti Google tietokonerypään yksityislevyjärjestelmään organisoituihin solmuihin *Google-tiedostojärjestelmään, GFS* (Google File System, GFS) [GGL03]. Kyseessä on hajautettu, skaalautuva ja vikasietoinen tiedostojärjestelmä, joka on tarkoitettu isoja tietokokonaisuuksia hajautetusti käsitteleville sovelluksille. Bigtable-taulu tallennetaan tiedostojärjestelmään muuttumattomiin *SSTable-tiedostoihin* (Sorted Strings Table) [CDG06, CDG08]. Tiedostomuoto koostuu levyllä olevasta pysyvästä, järjestetystä ja muuttumattomasta avain-arvo-bittijonopareja sisältävästä rakenteesta. Tiedostosta voidaan hakea tietoa tietyn avaimen perusteella tai käymällä läpi avain-arvo parit tietyllä avainvälillä. Tämä tiedosto muodostuu 64-kilotavun lohkosarjoista, jonka yksittäiset lohkot löydetään kunkin tiedoston lopussa sijaitsevan lohkoindeksin avulla. Kun tiedosto avataan, sen indeksi ladataan muistiin. Lohko haetaan indeksistä binäärihaulla, jonka jälkeen se luetaan levyltä. Vaihtoehtoisesti koko SSTable-tiedosto voidaan ladata muistiin, jolloin haussa ei tarvitse käyttää levyä ollenkaan. GFS ylläpitää lohkoista useampia toisinteita tietojen saatavuuden varmistamiseksi.

Bigtable-taulu ositetaan vaakasuoraan riveittäin paloiksi [CDG06, CDG08]. Taulu koostuu aluksi vain yhdestä palasta. Kun taulun tai palan koko kasvaa tarpeeksi suureksi, se jaetaan automaattisesti pienemmiksi oletusarvoisesti noin 1 gigatavun paloiksi. Yhteen riviavaimeen liittyvän rivin tieto sisältyy aina kokonaan yhteen palaan. Palaa ei siis voi jakaa pystysuoraan sarakkeittain kesken saman rivin. Palat tallennetaan fyysisesti SSTable-tiedostoihin. Yhden palan tiedot voivat olla useammassa SSTable-tiedostossa, ja sama tiedosto voi taas sisältää useamman palan tietoja. Tietokonerypäässä toimii yksi *isäntäpalvelin* (master server) ja monta sen hallinnoimaa *palapalvelinta* (tablet servers). Isäntäpalvelin ja palapalvelin ovat eri tyyppisiä rypään solmuja. Jokainen palapalvelin hallinnoi noin 10-1000 palaa. Palapalvelimet käsittelevät asiakassovellusten paloihin tekemät luku- ja kirjoituspyynnöt. Palapalvelimet hoitavat myös liian suureksi kasvaneiden palojen osituksen. Isäntäpalvelimen tehtäviin kuuluu esimerkiksi palojen

osoitus (assign) palapalvelimille, palapalvelimien seuranta, kuorman tasaus sekä tiedostojen poisto tiedostojärjestelmästä. Palojen osoitus tarkoittaa ilman palapalvelinta olevan palan määrittelemistä jollekin palapalvelimelle jolla on tilaa. Bigtablen *asiakaskirjasto* (client library) kommunikoi suoraan palapalvelimien kanssa. Asiakaskirjasto on komponentti, jonka kautta Bigtablen asiakassovellukset kommunikoivat palapalvelimien kanssa. Palojen paikantamiseksi asiakaskirjasto käy läpi kolmitasoiseen *sijaintihierarkian* (hierarchy). Sijaintihierarkiaan on tallennettu tieto palojen sijainnista. Asiakaskirjasto tallentaa välimuistiin kaikki palojen paikat, jotka se vain löytää. Hierarkian ensimmäisellä tasolla on pysyvä tiedosto, jossa kerrotaan erityisen *juuripalan* (root tablet) sijainti. Juuripalaan varastoidut sijaintitiedot viittaavat *metatietotaulun* (metadata table) paloihin. Kaikkien muiden palojen sijaintitiedot taas ovat metatietotaulun riviavaimissa. Riviavain koostuu palan taulutunnisteesta ja sen viimeisestä rivistä. Näiden tietojen avulla voidaan paikallistaa kukin pala.

Kuten relaatiotietokantajärjestelmätkin, monet NoSQL-tietokantajärjestelmät hyödyntävät toiminnassaan keskusmuistia sekä levyä. Tämän takia nämä järjestelmät ylläpitävät myös tietokannan transaktioiden käsittelyssä tarvittavia tietorakenteita, kuten puskurimuistia ja sitoutumislokia. Näitä käytetään hyvin samaan tapaan kuin relaatiotietokantajärjestelmissä. Esimerkiksi hyväksytty Bigtable-tietokannan kirjoitusoperaatio tallennetaan ensin levyllä olevaan sitoutumislokiin, jonne varastoidaan riveittäin transaktioiden toistotiedot [CDG06, CDG08]. Yhden palapalvelimen alaisuudessa olevien palojen kaikkien transaktioiden kaikki toistotiedot tallennetaan tiedostojärjestelmään yhteen fyysiseen sitoutumislokiin. Kun tietokannan kirjoitusoperaatio on sitoutunut, se viedään solmun keskusmuistissa olevaan rivitasolla järjestettyyn puskuriin. Puskurissa pidetään vain viimeisimmät päivitykset. Puskurin täyttyessä vanhemmat päivitykset siirretään solmun levylle muuttumattomiin SSTable-tiedostoihin. Tietokannan lukuoperaatio lukee tiedot puskurimuistista sekä SSTable-tiedostoista. Puskurimuisti on järjestelmän ainoa muuttuva tietorakenne, jota käsittelevät sekä luku- että kirjoitusoperaatiot. Tämä yksinkertaistaa Bigtablen samanaikaisuuden hallintaa. Tämän takia lukuoperaatioita SSTable-tiedostoihin ei tarvitse ajoittaa eli kaikki lukuoperaatioiden suoritusjärjestykset SSTable-tiedostoon on sallittu. Samanaikaiset luku- ja kirjoitusoperaatiot puskurimuistiin sallitaan tekemällä puskurimuistin riveistä *kirjoituskopioita* (copy-on-write) eri prosesseille. Kirjoituskopio on tarvittaessa muodostettu rivin kopio, johon muutokset suoritetaan. Tietokannan luku- ja kirjoitusoperaatiot voidaan siten suorittaa rinnakkain.

Kuvassa 4 esitetään Bigtable-solmuun tallennettuun palaan kohdistuvat tietokannan luku- ja kirjoitusoperaatiot. Kirjoitusoperaatio kirjoittaa muutokset ensin sitoutumislokiin. Tämän jälkeen muutos kirjoitetaan puskurimuistiin. Puskurin täyttyessä vanhemmat päivitykset siirretään SSTable-tiedostoihin. Lukuoperaatio lukee tiedot puskurimuistista ja muuttumattomista SSTable-tiedostoista.



Kuva 4: Bigtablen luku- ja kirjoitusoperaatioiden muistin ja levyn käytön esitys [CDG06, CDG08].

Palan metatietoihin on tallennettu lista niistä SSTable-tiedostoista, jotka muodostavat kyseisen palan fyysisesti sekä joukko *tarkistuspisteitä* (redo point) [CDG06, CDG08]. Tarkistuspisteet ovat *osoittimia* (pointers) tietokannan transaktioiden sitoutumislokeihin. Näissä osoittimissa on tieto mistä löydetään ne sitoutumislokit, joista voi löytyä palaa koskevaa tietoa. Jos palan tila pitää palauttaa eli elvyttää, tarkistuspisteiden perusteella löydetään ne sitoutumislokit joista löytyy palan tietoa. Palapalvelin lukee palan metatietojen perusteella palan muodostavien SSTable-tiedostojen indeksit muistiin ja luo puskurin uudestaan suorittamalla uudestaan kaikki tarkistuspisteiden jälkeen sitoutuneet päivitykset.

6.4 NoSQL-tietokantajärjestelmien luokittelu

Viime vuosina on kehitetty paljon uusia tietokantajärjestelmiä, joilla on joitain NoSQL-tietokantajärjestelmien ominaispiirteitä. Tietokantajärjestelmiä joita kutsutaan jollain perusteella NoSQL-tietokantajärjestelmäksi on tällä hetkellä olemassa useampia kym-

meniä ellei jopa satoja [TuB11]. Näillä tietokantajärjestelmillä on usein joitakin yhteisiä perusominaisuuksia, joita käsiteltiin edellä. Edellä esimerkkinä käytetty Bigtable-tietokantajärjestelmä on silti vain eräs NoSQL-paradigman toteutus. Laajan mittakaavan Web 2.0 -sovelluksia varten kehitetyillä NoSQL-tietokantajärjestelmillä on usein sen kaltaisia ominaisuuksia. Näiden eri järjestelmien tietokannan looginen tietomalli ja fyysinen rakenne saattavat kuitenkin poiketa hyvin paljon toisistaan. Tämän lisäksi on myös olemassa muunlaisia NoSQL-tietokantajärjestelmiä. Kaikkia näitä järjestelmiä ei ole edes kehitetty laajan mittakaavan Web 2.0 -sovellusten tarpeisiin. Erilaisten NoSQL-tietokantajärjestelmien luokitteluun ei kuitenkaan ole olemassa mitään vakiintunutta mallia. Luokitteluun on käytetty erilaisia toisistaan vaihtelevia malleja. Seuraavassa kuvataan muutamia erilaisia tapoja luokitella näitä järjestelmiä.

NoSQL-tietokantajärjestelmät voidaan luokitella esimerkiksi tietokannan noudattaman tietomallin mukaan. Usein käytetyssä luokittelumallissa [HeJ11] NoSQL-tietokantajärjestelmät on luokiteltu neljään eri ryhmään seuraavalla tavalla:

- *avain-arvo-varastot* (key value stores) (Voldemort, Redis ja Membase),
- *dokumenttivarastot* (document stores) (Riak, MongoDB ja CouchDB [The12a]),
- *sarakeperhevarastot* (column family stores) (Cassandra [LaM10, The12e], HBase [The12c] ja HyperTable) ja
- *verkkotietokannat* (graph databases) (Sesame, BigData, Neo4J, GraphDB ja FlockDB [Twi11]).

Avain-arvo-varastojen tietokannan tietomallia voidaan verrata hakemiston tai hajautustaulun kaltaiseen tietorakenteeseen, jossa arvot varastoidaan yksilöivän avaimen taakse [HeJ11]. Näiden järjestelmien tietokannan tietomalli perustuu yksinkertaiseen avain-arvo-pariin. Avain-arvo-parin varastoitu arvo löydetään suoraan indeksoitavan avaimen perusteella. Avain-arvo-varastojen arvot ovat kuitenkin yleensä *läpinäkymättömiä* (opaque) tietokantajärjestelmälle [Pop12]. Tämä tarkoittaa, että tietokantajärjestelmä ei voi tulkita arvoja mitenkään. Arvojen perusteella ei voi sen takia tehdä tietokantakyselyjä. Arvot ovat myös itsenäisiä ja eristettyjä toisistaan [HeJ11]. Arvojen mahdollisia keskinäisiä suhteita ylläpidetään sovelluslogiikassa. Tietomallin rakenne on hyvin joustava. Eri tyyppisiä arvoja voidaan lisätä tietokantaan ilman suuria muutoksia olemassa olevaan tietokannan rakenteeseen. Avain-arvo-varastojen tietokanta on siis usein täysin

skeematon. Tietokannan indeksointi- ja hakutoiminnot perustuvat yksilöivään avaimeen. Sovellusohjelmoijalle tarjotaan yleensä vain tähän avaimeen perustuvat yksinkertaiset put-, get- ja delete-operaatiot. Tämän kaltaisiin operaatioihin ei yleensä tarvita mitään kovin erikoista ohjelmointi- tai kyselykieltä. Avain-arvo-pareja voidaan kuitenkin yleensä myös ryhmitellä jollakin tavalla. Nämä avain-arvo-parit varastoidaan usein fyysisesti keskusmuistiin. Näin tietokannan tietoihin päästään käsiksi hyvin nopeasti. Tiedot kuitenkin saatetaan toisintaa ajoittain myös levylle. Tämä menettely takaa tietojen pysyvyyden. Avain-arvo-parit hajautetaan avaimen perusteella. Tietojen hajautus toteutetaan yleensä vaakasuoraan hajautusosituksella. Avain-arvo-varastot sopivat hyvin sovelluksille, jotka edellyttävät nopeita luku- ja kirjoitusoperaatioita. Avain-arvo-varastot tarjoavat kuitenkin tietokannan tietojen käsittelyyn vain yksinkertaiset operaatiot, joten ne eivät välttämättä sovi kovin hyvin monimutkaisia tietokantakyselyjä edellyttäville sovelluksille. Kuvassa 5 esitetään yksinkertainen avain-arvo-varastojen tietomalli. Jokaiseen avaimeen liittyy arvo. Jokaisella arvolla on taas yksilöllinen avain. Arvoilla ei ole tietokannassa mitään keskinäistä suhdetta.

Avain	Arvo
1	Janne
2	Anne
3	Mari
4	Liisa
01	Toyota
02	Volvo
03	Ford
04	BMW

Kuva 5: Avain-arvo-varastojen tietokannan tietomallin eräs esitys.

Dokumenttivarastojen tietokannan tietomalli on yksinkertaisia avain-arvo-varastoja monimutkaisempi. Tietomallin rakennetta kutsutaan joskus myös *semirakenteelliseksi malliksi* (semi structured model) [WIK12b]. Sillä voidaan mallintaa rakenteellisempia tietoja. Dokumenttivarastot koostuvat avain-arvo-pareja sisältävien dokumenttien kokoelmista [HeJ11]. Avain-arvo-parin avain on jokaisen arvon yksilöivä tunnus dokumentin sisällä. Sen lisäksi jokaisella dokumentilla on oma yksilöllinen tunnus. Dokumentti myös löydetään tämän indeksoitavan avaimen perusteella. Dokumenttivarastojen avain-arvo-parien arvot ovat yleensä *läpinäkyviä* (transparent) tietokantajärjestelmälle [Pop 12]. Tämä tarkoittaa, että tietokantajärjestelmä pystyy jotenkin myös tulkitsemaan arvoja. Tiedot voidaan esimerkiksi tallentaa tietokantaan sellaisessa formaatissa, jota tieto-

kantajärjestelmä voi tulkita. Tämän ansiosta tietokannan tietoja voidaan hakea pelkän avaimen lisäksi myös arvojen perusteella. Dokumenttivaraston tietokanta on yleensä skeematon [HeJ11]. Dokumenttien sisällä voi olla eri määrä toisistaan poikkeavia avain-arvo-pareja. Niitä voidaan myös lisätä tarpeen vaatiessa ilman skeemamuutoksia. Dokumenttivarastoihin voidaan yleensä kuitenkin tehdä monimutkaisempia hakuja kuin avain-arvo-varastoihin. Dokumenttivarastoissa on usein mahdollista ylläpitää tietokannan toissijaisia indeksejä, joten niitä voidaan hyödyntää tietokantakyselyissä [HeJ11, 10g12]. Dokumenttiin voidaan kohdistaa hakuja minkä tahansa dokumentin sisältönä olevan kentän perusteella. Arvoja voidaan hakea esimerkiksi arvovälillä. Käytössä on myös joitain esimerkiksi SQL-kyselykielen syntaksista tuttuja toimintoja, kuten and, or tai between vertailuoperaattorit. Näiden lisäksi hakua voidaan yleensä rajoittaa säännöllisten lausekkeiden avulla. Samaa ohjelmointi- tai kyselykieltä ei kuitenkaan yleensä voi käyttää useammassa dokumenttivarastossa. Kehitteillä on kuitenkin ollut yhteinen kyselykieli *UnQL* [Wil12]. UnQL on NoSQL-tietokantajärjestelmiä varten kehitteillä ollut SQL:ää muistuttava kyselykieli. UnQL:stä yritetään kehittää standardi kieli, jonka syntaksin avulla voidaan käsitellä itsekuvautuvaa rakenteetonta tietoa. Tämä kieli on kuitenkin vielä edelleen kehitysvaiheessa. Dokumentti voidaan yleensä toteuttaa jossain standardissa formaatissa, kuten *JSON* (JavaScript Object Notation, JSON) tai XML. JSON on tekstimuotoinen standardi tiedonsiirtoformaatti, jota käytetään usein samaan tarkoitukseen kuin XML-kieltä [SKS11, s. 864]. JSON on johdettu JavaScript-ohjelmointikielestä, mutta se on kuitenkin ohjelmointikielestä riippumaton formaatti. Formaatin tietotyyppejä ovat esimerkiksi taulukko tai *olio* (object). JSON olio on järjestämätön tietorakenne, joka voi koostua useammasta avain-arvo-parista. Tätä formaattia voidaan myös käyttää tietojen *sarjallistamiseen* (serialization). Tietojen sarjallistaminen tarkoittaa tietorakenteen muuntamista johonkin toiseen formaattiin tietojen helpompaa ja tehokkaampaa tallentamista tai siirtoa varten. Tietorakenne voidaan palauttaa alkuperäiseen muotoon saman sarjallistamisessa käytetyn formaatin avulla. Dokumenttivarastot ovat ilmaisuvoimaisia, koska niissä voidaan käyttää edellä kuvattuja formaatteja. Dokumentit hajautetaan dokumenttiavaimen perusteella [HeJ11]. Hajautus toteutetaan yleensä vaakasuoraan hajautus- tai osaväliosituksella. Dokumenttivarastojen tietomallin avulla voidaan usein mallintaa tehokkaasti kokoelmia, hierarkioita sekä sisäkkäisiä tietoja. Näistä dokumenteista koostuvia dokumenttivarastoja käytetään usein tosiaikaisessa analyysissä, lokeissa sekä Internet-blogeissa.

Kuvassa 6 esitetään dokumenttivarastojen tietokannan tietomalli. Kuvassa on kokoelma dokumentteja. Dokumentit sisältävät työntekijöiden tietoja. Jokaisella dokumentilla on dokumentin yksilöivä tunnus. Dokumenteissa voi olla eri määrä toisistaan poikkeavia avain-arvo-pareja. Mari on ainoa jolle on tallennettu työpari. Anne on ainoa jolle on tallennettu lempiväri. Jannelle ei ole tallennettu kumpaakaan edellistä. Marin työpari-avaimelle on tallennettu useampi työpari jollain sarjallistettavalla tietotyypillä.

Dokumenttikokoelma: työntekijät

Dokumenttiavain: 121214	
Avain	Arvo
Nimi	Mari
Ikä	28
Auto	Volvo
Lemmikki	Kissa
Työpari	(Nimi: "Anne", Nimi: "Liisa")

Dokumenttiavain: 121212	
Avain	Arvo
Nimi	Janne
Ikä	47
Auto	Toyota
Lemmikki	Koira

Dokumenttiavain: 121213	
Avain	Arvo
Nimi	Anne
Ikä	30
Auto	BMW
Lemmikki	Kultakala
Lempiväri	Musta

Kuva 6: Dokumenttivarastojen tietokannan tietomallin eräs esitys.

Sarakeperhevarastojen tietokannan tietomallin rakennetta voidaan verrata ison matriisin tai taulukon kaltaiseen tietorakenteeseen, joka koostuu riveistä ja sarakkeista. Avain-arvo-parit varastoidaan kyseisen tietokannan tauluun samaan tapaan kuin avain-arvo-mallin Bigtable-esimerkissä. Riviavaimen, sarakeavaimen sekä mahdollisen aikaleiman yhdistelmä eli avain-arvo-parin avain viittaa varsinaiseen tietosisältöön, joka on avain-arvo-parin arvo. Tietoja voidaan yleensä kysellä ainoastaan riviavaimen, tai joissain tapauksissa jonkin muun indeksoidun arvon perusteella [HeJ11]. Avain-arvo-varastojen tapaan sarakeperhevarastojen arvot ovat yleensä läpinäkymättömiä tietokantajärjestelmälle. Arvojen mahdollisia keskinäisiä suhteita ylläpidetään sovelluslogiikassa. Samasta tiedosta voi olla kuitenkin varastoituna monta eri versiota. Käytössä saattaa myös olla alemman tason ohjelmointikieliä ilmaisuvoimaisempia omia ohjelmointi- ja kyselykieliä [Goo12d, PDG05]. Samaa ohjelmointikieltä ei kuitenkaan yleensä voi käyttää useammassa eri sarakeperhevarastossa. Esimerkiksi Google Bigtable -tietokannan kyselymahdollisuuksia käsiteltiin edellä. Bigtable-tietokannan taulun rakenne on moniulotteinen,

järjestetty, harva, pysyvä ja hajautettu [CDG06, CDG08]. Taulun rivin muodostumista käsiteltiin edellä. Tietokannan tiedot indeksoidaan riviavaimien, sarakeavaimien ja aikaleimojen mukaan. Näiden yhdistelmä viittaa varsinaiseen tietosisältöön eli taulun yksittäiseen *soluun* (cell). Koko solun sisältö tallennetaan string-tietotyyppinä, jolla ei ole mitään tietokantajärjestelmän tulkittavissa olevaa merkitystä. Kunkin Bigtable-taulun rivit on järjestetty rivin avaimen mukaan leksikografiseen järjestykseen. Samaan asiaan liittyvät tiedot on siis mahdollista ryhmitellä lähelle toisiaan määrittelemällä riviavaimet sopivalla tavalla, kuten kuvan 7 esimerkissä. Näin toisiaan lähekkäin olevat tiedot voidaan lukea tehokkaammin. Rivillä voi olla vaihteleva määrä sarakkeita. Ainoastaan kunkin rivin käytössä olevat sarakkeet tallennetaan fyysisesti. Tämä tekee tietomallista harvan. Sarakeperhevarastoja kutsutaan sen takia joskus myös *laajennettavaksi arvovarastoksi* (extensible record stores) [StC11]. Laajennettavat arvo-varastot sisältävät arvojoukkoja, joiden leveys voi vaihdella. Nämä arvojoukot voidaan osittaa hajautetun tietokantajärjestelmän eri solmuille, joko vaakasuoraan tai pystysuoraan. Bigtable-taulun sarakkeet on esimerkiksi ryhmitelty saman tyyppistä tietoa sisältäviin *sarakeperhe* (column family) nimisiin sarakejoukkoihin [CDG06, CDG08]. Sarakeperhe on taulukohtainen. Rivejä ja sarakkeita voidaan lisätä koska tahansa, mutta sarakeperhe on usein myös määriteltävä aina ennen kuin sen sarakkeisiin voidaan tallentaa mitään. Tietokanta ei siis ole täysin skeematon. Tämän takia sarakeperhevaraston tietokannan tietomalli ei ole välttämättä yhtä joustava kuin avain-arvo-varasto tai dokumenttivarasto. Sarakeavain koostuu parista *sarakeperhe:määre* (family:qualifier). Määreen avulla erotellaan saman sarakeperheen eri sarakkeet. Esimerkiksi ankkuri:hs.fi-sarakkeen sarakeperhe on ankkuri ja määre eli yksittäinen sarake on hs.fi. Sarakeperhe on yksikkö jota voidaan käyttää hyväksi hajautetun tietokannan tietojen ryhmittelyssä, tallennuksessa ja sijoittelussa. Tietokannan transaktioiden suoritukseen ja tehokkuuteen sekä tietojen saatavuuteen ja oikeellisuuteen voidaan vaikuttaa määrittelemällä sopivia sarakeperheitä. Sovellusohjelmoija määrittelee tietokannan skeemassa halutut sarakeperheet. Skeemassa on sen jälkeen mahdollista määritellä näiden sarakeperheiden fyysiseen tallennukseen ja sijoitteluun vaikuttavia asetuksia. Näillä asetuksilla voidaan vaikuttaa hajautetun tietokannan sirpalointiin. Esimerkiksi ne sarakeperheet joihin kohdistuu tietokantaoperaatioita usein samaan aikaan, voidaan sijoittaa lähekkäin toisiaan. Nämä sarakkeet voidaan tallentaa omaan SSTable-tiedostoon. Tämä SSTable-tiedosto voidaan sijoittaa mahdollisimman lähelle sitä pistettä, josta sen sisältämien sarakkeiden tietoja useimmin käyte-

tään. SSTable-tiedosto voidaan myös määritellä ladattavaksi aina kokonaan muistiin. Näin toisiaan lähekkäin olevia tietoja voidaan hyödyntää mahdollisimman tehokkaasti. Sarakeperheitä voidaan käyttää lisäksi myös käyttöoikeuksien rajaamiseen sekä levytilan ja muistin määrän mittayksikkönä.

Sarakeperhevarastojen tietokannan tietomallissa voi olla myös kolmas ulottuvuus, nimittäin aika. Tietokannan tauluihin voidaan tallentaa samasta tiedosta useita eri versioita. Aina kun tietoja muokataan siitä tallennetaan uusi versio. Vanha versio jää myös talteen. Esimerkiksi Bigtable-tietokannan tietojen versiot indeksoidaan sekä erotellaan toisistaan aikaleiman perusteella [CDG06, CDG08]. Eri versiot tallennetaan ja indeksoidaan laskevassa aikajärjestyksessä taulun soluihin niin, että uusin tieto on luettavissa aina ensin. Bigtable voi määritellä aikaleiman tai vaihtoehtoisesti Googlen asiakassovellukset voivat määritellä sen itse. Käyttäjät voivat myös itse määritellä, kuinka monta eri versiota tai minkä ajanjakson eri versiot tiedoista halutaan säilyttää. Sarakeperheiden tietokannan tietomallia voidaan kutsua rivien, sarakkeiden ja aikaleimojen takia moniulotteiseksi. Google hyödyntää Bigtablea esimerkiksi sovelluksissa, jotka edellyttävät korkeaa suoritustehoa sekä alhaisia latenssikustannuksia [HeJ11]. Sarakeperhevarastojen tietokannan tiedot hajautetaan rivi- ja sarakeavaimen perusteella. Hajautus toteutetaan yleensä hajautus- tai osaväliosituksella. Tiedot ositetaan vaakasuoraan riveittäin ja pystysuoraan sarakeperheiden avulla. Sarakeperhevarastojen tietokantaan voidaan myös tallentaa vaihteleva määrä attribuutteja harvan tietomallin ansiosta. Sarakeperhevarasto soveltuu siis hyvin esimerkiksi laajan mittakaavan Web 2.0 -sovelluksien käyttöön.

Kuvassa 7 on esitetty web-sivuja varastoivan Bigtable-taulun siivu. Rivin avain on web-sivun URL-osoite käänteisessä järjestyksessä. Tämän ansiosta samaan domainiin ja alasivustoon kuuluvat web-sivut ryhmittyvät lähekkäin toisiaan peräkkäisille riveille. Sarakkeet sisältävät web-sivun (sisältö:) sekä kyseiselle sivulle viittaavat muut sivustot (ankkurit hs.fi, mtv3.fi ja nelonen.fi). Kaikilla riveillä ei ole tietoja samoissa sarakkeissa. Sivun www.yle.fi sisältösarake sisältää useita eri versioita samasta sivusta. Sen eri sivuversioiden aikaleimat on merkitty kuvassa t1, t2, ja t3. Saman rivin ankkurisarakeperheen eri sarakkeet hs.fi ja nelonen.fi sisältävät kuvassa vain yhden version. Niiden aikaleimat ovat t4 ja t5.

Riviavain	Sisältö:	Ankkuri:hs.fi	Ankkuri:mtv3.fi	Ankkuri:nelonen.fi
fi.yle.www	"<html>…" t3 "<html>…" t2 "<html>…" t1	"YLE" t4		"YLE" t5
fi.yle.www/saa	"<html>…"	"YLE sää"		"YLE sää"
fi.yle.www/urheilu	"<html>…"	"YLE urheilu"		
fi.yle.www/uutiset	"<html>…"		"YLE uutiset"	
fi.yle.www/uutiset/helsinki	"<html>…"			
fi.yle.www/uutiset/etela-karjala	"<html>…"			
fi.yle.www/uutiset/lappi	"<html>…"			
fi.yle.www/uutise/tampere	"<html>…"			

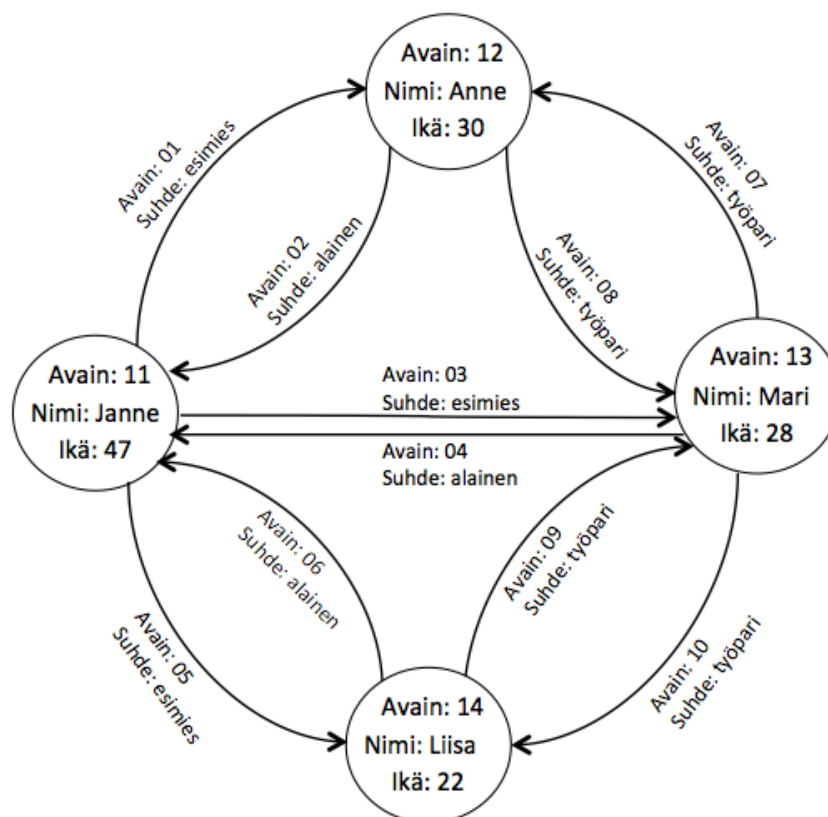
Kuva 7: Web-sivuja varastoivan Bigtable-taulun siivu [CDG06, CDG08].

Verkkotietokantojen tietokannan tietomallia voidaan verrata verkkomaiseen rakenteeseen. Tämän tietomallin avulla tietoihin voidaan liittää semantiikkaa. Tietomallissa mallinnetaan tietojen suhteita [HeJ11]. Tämän suunnatun verkon muodostavat solmut ja niitä yhdistävät kaaret. Solmusta voi taas olla kaari useampaan toiseen solmuun. Avain-arvo-parit on tässä tapauksessa varastoitu näihin verkon solmuihin ja kaariin. Sekä solmuun että kaareen voi olla varastoitu useampi avain-arvo-pari. Arvoihin ei kuitenkaan päästä käsiksi välttämättä suoraan avaimen perusteella. Arvoja joudutaan usein hakemaan verkon läpikäynnillä tutkimalla solmujen välisiä suhteita. Koska jokainen solmu tuntee vierussolmunsa, yksittäinen solmu voidaan löytää solmujen välisen polun avulla. Verkkotietokanta on yleensä skeematon. Solmuihin ja kaariin voidaan lisätä eri tyyppisiä avain-arvo-pareja ilman tietokannan skeemamuutoksia. Skeemaa voidaan kuitenkin hyödyntää. Avain-arvo-parien arvoaluetta voidaan usein rajoittaa skeemassa. Skeemassa voidaan myös määritellä, että vain tietyntyyppisten solmujen välillä voi olla vain tiettyjä kaaria. Verkkotietokantojen tietokannan tietomalli on hyvin tehokas tavanomaisissa verkko-operaatioissa, kuten verkon läpikäynnissä. Verkon läpikäynnissä vieraillaan kaikissa verkon solmuissa tai kaikissa lähtösolmusta saavutettavissa olevissa solmuissa. Verkon läpikäyntiin sovelletaan yleensä leveys- tai syvyyssuuntaista hakua. Verkosta on myös mahdollista löytää tehokkaasti lyhin polku lähtösolmusta maalisolmuun. Tähän voidaan usein soveltaa esimerkiksi jotain tunnettua verkkoalgoritmia, kuten Bellman-Ford, Dijkstra tai Floyd-Warshall. Verkosta voidaan hakea myös jotain tiettyä osaa. Esimerkiksi voidaan olla kiinnostuneita verkon aliverkosta, kuten jostain virittävästä puusta. Tähän voidaan soveltaa *hahmontunnistusta* (pattern matching). Hahmontunnistus on tekniikka, jolla etsitään jotain tietyn mallista merkkijonoa tiedostosta tai syötevirrasta. Hahmontunnistuksella voidaan etsiä joku verkon aliverkko, joka täsmää kyselyeh-

toon [Rod12]. Joissain verkkotietokannoissa voidaan käyttää myös yhteistä ohjelmointi- tai kyselykieltä, kuten *SPARQL* [W3C12] tai *Gremlin* [Tin12]. SPARQL on deklaratiivinen kyselykieli, jonka syntaksin avulla voidaan suorittaa hahmontunnistusta. Gremlin on ohjelmointikieli, jolla voidaan suorittaa verkon läpikäynti. Verkkotietokantoja voidaan hyödyntää tehokkaasti esimerkiksi navigointisovelluksessa [HeJ11]. Tietomallin avulla voidaan mallintaa myös monimutkaisia suhteita. Verkkotietokantojen tietokannan tietomalli sopii hyvin ihmisten verkostoitumista edistävien sosiaalisen median sovellusalueen mallinnukseen. Tämä sovellusalue voidaan mallintaa verkkona. Näissä sovelluksissa ihmiset muodostavat usein ystävistä koostuvan verkon, joka koostuu eri tyyppisistä suhteista. Ystävät voidaan jaotella esimerkiksi sukulaisiin, lähimpiin ystäviin ja ystävien ystäviin [Fac12]. Kuhunkin ryhmään voi kohdistua erilaisia ominaisuuksia. Nämä ryhmät voidaan mallintaa ystäväverkon aliverkkona. Usein tämänkaltaisissa sovelluksissa käyttäjät voivat myös jakaa, kommentoida, suositella tai *tykätä* (like) käyttäjien palveluun lataamaa sisältöä. Tykkääminen tarkoittaa, että käyttäjä voi jollain yksinkertaisella tavalla ilmaista pitävänsä palveluun ladatusta sisällöstä. Verkon solmujen ei kuitenkaan tarvitse mallintaa henkilöitä. Solmut voivat mallintaa myös muita entiteettejä, kuten esimerkiksi jotain yhteisöä johon henkilöt palvelussa kuuluvat.

Verkkotietokannan toteuttaminen laajan mittakaavan hajautetussa ympäristössä saattaa kuitenkin olla hankalaa. Verkkotietokannoissa ei usein hajauteta tietoja sirpaloimalla ja osittamalla [HeJ11]. Verkkomalliin perustuvan tietokannan tietojen osittaminen ei ole välttämättä kovin tehokasta. Solmujen joiden välillä on paljon keskinäisiä suhteita tulisi olla mahdollisimman lähellä toisiaan, etteivät verkon läpikäynnistä aiheutuvat latenssikustannukset kasva liian suuriksi. Toisaalta solmujen kokonaismäärä olisi myös hyvä olla jakaantunut mahdollisimman tasaisesti hajautetun tietokantajärjestelmän eri pisteisiin vinouman välttämiseksi. Paljon keskinäisiä suhteita sisältävää solmuryvästä kutsutaan verkon *kuumaksi pisteeksi* (hot spot). Edellä kuvatun kaltaisen ystäväverkon suhteet voivat kuitenkin muuttua jatkuvasti, joten kuumien pisteiden sijainti vaihtelee. Tietokannan tietojen osittaminen näiden kuumien pisteiden perusteella on siis vaikeaa. Vaikka hajautetut verkkotietokantajärjestelmät eivät yleensä hyödynnä tietojen osittamista, ne kuitenkin usein toisintavat tietoja tietokantajärjestelmän saatavuuden ja kestävyyden parantamiseksi.

Kuvassa 8 esitetään verkkotietokantojen tietokannan tietomalli. Kuvassa työntekijät sekä työntekijöiden väliset suhteet on mallinnettu suunnattuna yhtenäisenä verkkona. Verkon ei kuitenkaan tarvitse olla yhtenäinen. Yksittäisen työntekijän avain-arvo-parit ovat verkon solmuissa. Jokaisella työntekijällä on yksilöivä avain, nimi ja ikä. Työntekijöiden väliset suhteet ovat näiden solmujen välisiä kaaria. Jokaiseen suhteeseen liittyy avain-arvo-pareina yksilöivä avain sekä suhde. Työparit muodostavat oman ryhmän eli aliverkon verkon sisällä.



Kuva 8: Verkkotietokantojen tietokannan tietomallin eräs esitys.

Stonebraker ja kumppanit [StC11] ja Rick Cattel [Cat10] luokittelevat NoSQL-tietokantajärjestelmät tietokannan tietomallin mukaan merkittävimpiin ryhmiin seuraavasti:

- avain-arvo-varastot (Dynamo, Voldemort, Membase, Membrain, Scalaris, Riak, Redis ja Tokyo Tyrant),
- dokumenttivarastot (CouchDB [The12a], MongoDB, SimpleDB ja Terrastore) ja
- laajennettavat arvovarastot (extensible record stores) (Bigtable [CDG06, CDG08], Cassandra [LaM10, The12e], HBase [The12c], HyperTable ja

PNUTS).

Stonebraker ja kumppanit [StC11] kuvaavat avain-arvo-varastoja *oliokokoelmiksi* (collection of objects). Tämän kuvauksen mukaan jokaisella olioilla on avain ja arvo. Tietokantajärjestelmällä ei ole mahdollisuutta liittää arvoihin mitään erityistä merkitystä tai tulkita arvoja attribuuteiksi. Dokumenttivarastojen tietokannan tietomalli on kuvattu oliokokoelmaksi, joka tässä kuvauksessa koostuu olioista, joilla on useampia attribuutteja. Rakenne voi myös koostua sisäkkäisistä olioista. Laajennettavat arvo-varastot sisältävät artikkelin mukaan arvojoukkoja, joiden leveys voi vaihdella. Tätä käsiteltiin edellä.

NoSQL-tietokantajärjestelmät voidaan jakaa ryhmiin myös yleisluontaisemmin. Stefan Edlich [EDL12] jaottelee nämä järjestelmät yleisluontaisesti *ydin NoSQL -järjestelmiksi* (core NoSQL systems) ja *soft NoSQL -järjestelmiksi* (soft NoSQL systems). Ydin No SQL -järjestelmiä ovat enimmäkseen Web 2.0 -sovellusten tarpeisiin kehitetyt tietokantajärjestelmät. Näitä ovat edellä käsitellyt avainarvo-varastot, dokumenttivarastot, sarakeperhevarastot ja verkkotietokannat. Niiden lisäksi Edlich luokittelee ydin NoSQL -järjestelmiksi sellaiset järjestelmät, joilla on useampien erilaisten tietokantajärjestelmien ja työkalujen ominaisuuksia. Näitä hän kutsuu *multimallitietokannoiksi* (multimodel databases). Näitä ovat esimerkiksi OrientDB ja AlchemyDB. Tähän ryhmään hän luokittelee järjestelmät, joilla on esimerkiksi jokin yhdistelmä edellä mainittujen ydin NoSQL-järjestelmien ominaisuuksia. Tämän lisäksi multimallitietokannoilla voi olla relaatiotietokantojen ominaisuuksia ja ne voivat noudattaa tietokannan transaktioiden ACID-ominaisuuksia. Soft NoSQL -järjestelmiksi taas luokitellaan Edlichin mukaan muut mahdollisesti relaatiomallista poikkeavat tietokantajärjestelmät joita ei oltu enimmäkseen kehitetty Web 2.0 -sovellusten tarpeisiin, kuten esimerkiksi XML-tietokannat sekä tietyt *oliotietokannat* (object databases), kuten Db4o, Versant ja Objectivity. Oliotietokannoissa on olio-ohjelmointia helpottavia ominaisuuksia. Tiedot voidaan esimerkiksi tallentaa tietokantaan olioina. Näitä tietoja voidaan siten käsitellä suoraan olio-ohjelmointiin erikoistuneella ohjelmointikielellä.

Kuten luvun alussa todettiin NoSQL-tietokantajärjestelmien luokittelutapa ei ole vielä vakiintunut. Tätä kuvaa hyvin miten Internetin eri lähteissä, kuten blogikirjoituksissa [EDL12, HOF12, YEN09] ja Wikipediassa [WIK12a] on luokiteltu mahdollisesti relaatiomallista poikkeavia tietokantajärjestelmiä erilaisiin ryhmiin esimerkiksi tietokannan tietomallin, toimintatavan tai toteutuksen mukaan. Näitä järjestelmiä saatetaan kutsua

esimerkiksi NoSQL-tietokannoiksi tai *NoSQL-toteutuksiksi* (NoSQL implementations). Usein näitä erilaisia luokitteluryhmiä ja luokittelun syitä ei ole selostettu sen tarkemmin. Taulukkoon 1 on koottu näitä edellä mainittuja järjestelmiä eri luokitteluryhmiin.

Luokitteluryhmä	**Järjestelmä**
avain-arvo-varastot	Keyspace, Flare, Schema Free, RAMCloud
avain-arvo-varastot levyllä (key-value stores on solid state or rotating disk)	Bigtable [CDG06, CDG08], CDB, Keyspace, LevelDB, Membase, MemcacheDB, MongoDB, Virtuoso, Tarantool, Tokyo Cabinet, TreapDB, Tuple space
avain-arvo-välimuistit (key-value-cache)	Memcached [Dor12], Virtuoso, Coherence, Redis, Hazelcast, Tuple Space, Repcached, Infinispan, EXtreme Scale, Jboss Cache, Velocity, Terracoqa
hierarkkiset avain-arvo-varastot (hierarchical key-value store)	GT.M, InterSystems Cache
järjestetyt avain-arvo-varastot (ordered key-value stores)	Berkeley DB, IBM Informix C-ISAM, InfinityDB, MemcacheDB, NDBM, Tokyo Tyrant, Lightcloud, NMDB, Luxio, Actord
lopulta oikeelliset avain-arvo-varastot (eventually consistent key-value store)	Cassandra [LaM10, The12e], Dynamo, Hibari, Virtuoso, Voldemort, Riak, Dynomite, SubRecord, Mo8onDb, Dovetaildb
ylläpitopalvelut (hosted services)	Freebase, Virtuoso, Google Appengine Datastore [Goo12b]
verkkotietokannat (graph databases)	AllegroGraph, DEX, FlockDB [Twi11], InfiniteGraph, Neo4j, Virtuoso, OrientDB, Pregel, Sones GraphDB, InfoGrid, HyperGraphDB, Trinity, BrightstarDB, Big-

	data
tietorakennepalvelin (data-structures server)	Redis
oliotietokannat	Db4o, GemStone/S, InterSystems Cache, JADE, NeoDatis ODB, ObjectDB, ObjectivityDB, ObjectStore, Virtuoso, Versant Object Database, Wakanda, ZODB, ZopeDB, Shoal, Starcounter, Perst, NEO
dokumenttivarastot	Virtuoso, OrientDB, SimpleDB, Terrastore, CouchDB [The12a], MongoDB, Jackrabbit, XML-tietokannat, ThruDB, CloudKit, Perservere, Riak, Basho, Scalaris, BaseX, Clusterpoint, Exist, Lotus Notes, Lotus Domino, MarkLocig Server, RavenDB, SisoDB
laajat sarake-varastot (wide columnar store)	Bigtable [CDG06, CDG08], Hadoop [The12b], Hbase [The12c], Cassandra [LaM10, The12e], Hypertable, KAI, OpenNeptune, Qbase, KDI, Amazon SimpleDB
taulukko-varastot (tabular store)	Bigtable [CDG06, CDG08], Hadoop [The12b], Hbase [The12c], Hypertable, Mnesia, Virtuoso
monikko-varastot (tuple store)	Gigaspaces, Coord, Apache River, Virtuoso, Tarantool

Taulukko 1: NoSQL-tietokantajärjestelmien luokittelua.

Taulukossa 1 esiintyvistä luokitteluryhmistä ja järjestelmistä huomataan, että sama järjestelmä voi olla luokiteltuna useampaan eri ryhmään. MongoDB löytyy esimerkiksi avain-arvo-varastot levyllä sekä dokumenttivarastot ryhmistä. OpenLinkin Virtuoso löytyy myös monesta eri ryhmästä. Bigtable [CDG06, CDG08] on luokiteltu avain-arvo-varastoksi levyllä, laajaksi sarake-varastoksi ja taulukko-varastoksi. Joissakin läh-

teissä kaikkia Bigtablen kaltaisia järjestelmiä taas kutsutaan *Bigtable-toteutuksiksi* (Bigtable implementations) [WIK12a]. Osalla näistä järjestelmistä kuten Virtuosolla on myös relaatiotietokantojen ominaisuuksia, tai niitä ei ollut kehitetty alun perin laajan mittakaavan Web 2.0 -sovellusten tarpeisiin, kuten XML-tietokannat ja perinteiset oliotietokannat. Tästä tutkielmasta rajataan pois nämä aikaisemmin Stefan Edlichin [EDL12] Soft NoSQL -järjestelmiksi määrittelemät järjestelmät, ja keskitytään vain laajan mittakaavan Web 2.0 -sovellusten tarpeisiin kehitettyihin uudempiin tietokantajärjestelmiin.

Koska NoSQL-tietokantajärjestelmät ovat kovin heterogeenisiä järjestelmiä, niiden luokittelu on vaikeaa. Luokittelun hankaluutta kuvaa esimerkiksi luvussa 7 esiteltävä ylläpitopalveluna tarjottava multimallitietokantaan ja sarakeperhevarastoon perustuva hajautettu tietokantajärjestelmä, jossa on olio-ohjelmointia helpottavia ominaisuuksia sekä hierarkkinen tietokannan tietorakenne. Tämän kaltaista montaa eri tyyppistä tietokantajärjestelmää muistuttavaa järjestelmää saattaa olla siis hankalaa luokitella. Edellisten tietokannan tietomallien yleiskuvauksista kuitenkin huomataan, että kaikille yleisille NoSQL-tietokantajärjestelmien tietokannan tietomalleille yhteinen perusyksikkö on avain-arvo-pari. Yksinkertaisimmasta avain-arvo-pariin perustuvasta tietomallista on johdettu monimutkaisempia tietomalleja, jotka pystyvät käsittelemään yksinkertaisinta mallia rakenteellisempaa ja monimutkaisempaa tietoa. Näillä monimutkaisemmilla tietomalleilla on usein myös enemmän vaatimuksia. Tarvitaan esimerkiksi ilmaisuvoimaisempia ohjelmointirajapintoja ja ohjelmointi- tai kyselykieliä. Tietojen laajan mittakaavan osittaminen ja hajauttaminen voi olla myös vaikeampaa. Vaikka jokin tietomalli sinänsä sopii kohdealueen tietojen mallintamiseen, sen vaatimukset saattavat osaltaan myös siis rajoittaa kyseisiä tietokannan tietomalleja soveltavien tietokantajärjestelmien käyttökohteita.

NoSQL-tietokantajärjestelmät voivat kuitenkin poiketa toisistaan muutenkin kuin tietokannan tietomallin osalta. Kuten edellä on tullut esille, nämä järjestelmät voivat poiketa tietokannan tietomallin lisäksi myös ohjelmointirajapinnan, ohjelmointi- tai kyselykielen sekä fyysisen rakenteen osalta. Tämä tulee esiin myös useammassa lähdeartikkelissa, jossa vertaillaan muutamaa eri tyyppistä NoSQL-tietokantajärjestelmää keskenään [Cat10, HEL11]. Fyysisen toteutuksen osalta käytössä voi olla lisäksi esimerkiksi erilaisia tallennus, lukitus-, ositus- ja toisinnusmenetelmiä sekä transaktioiden samanaikai-

suuden hallintamenetelmiä. Useimmat näistä ominaisuuksista voivat vaihdella myös samaa tietokannan tietomallia noudattavien eri tietokantajärjestelmien välillä. Joitakin näistä ominaisuuksista käsiteltiin edellä. Näitä ominaisuuksia yhdistelevät toteutukset voivat olla myös hyvin monimutkaisia. Luvussa 7 tarkastellaan lisää erikoistuneita ratkaisuja. Hecht ja kumppanit [HeJ11] toteavat, että NoSQL-tietokantajärjestelmien nykyistä kehitysvaihetta voidaan verrata aikaan ennen SQL:ää. Ennen SQL:ää kehitettiin teknisesti heterogeenisiä tietokantajärjestelmiä. Silloin kehitys johti lopulta yhtenäiseen malliin ja standardiin. NoSQL-tietokantajärjestelmille yritetään myös kehittää yhteisiä standardeja, kuten yhteistä kyselykieltä [MeB11, Wil12]. Tällaisia kaikille yhteisiä ominaisuuksia ei kuitenkaan ole vielä otettu laajalti käyttöön. Yhteisten standardien kehittäminen on vaikeaa vaatimusten ja toisistaan poikkeavien erikoistuneiden toteutuksien takia. Joillakin NoSQL-tietokantajärjestelmillä on kuitenkin käytössään yhteinen ohjelmointimalli tietojen käsittelyyn. NoSQL-tietokantajärjestelmä on usein kehitetty toimimaan laajan mittakaavan infrastruktuurissa, jonne voidaan varastoida iso määrä tietoa [HeJ11]. Tavanomaiset tietokantakyselyt eivät ole riittäviä tällaisten tietokokonaisuuksien monimutkaiseen analyyttiseen käsittelyyn [PDG05]. Tämän takia esimerkiksi useilla dokumentti- ja sarakeperhevarastoilla [CDG06, CDG08, The12a] on käytössä yksinkertaista MapReduce-ohjelmointiparadigmaa hyödyntävä MapReduce-rajapinta [DeG08], jonka avulla voidaan suorittaa tehokkaasti laajan mittakaavan rinnakkaislaskentaa [HeJ11].

6.5 MapReduce-ohjelmointiparadigma

MapReduce on suurten tietokokonaisuuksien rinnakkaislaskentaan kehitetty ohjelmointiparadigma. Tämän ohjelmointimallin sekä siihen liittyvän kehyksen toteutuksen avulla voidaan käsitellä ja luoda suuria tietokokonaisuuksia samanaikaisesti rinnakkain laajan mittakaavan hajautetuissa tietokonerypäissä [DeG08]. MapReduce-ohjelmalle annetaan syötteenä joukko avain-arvo-pareja, joista tuotetaan tuloksena joukko avain-arvo-pareja. MapReduce-laskenta perustuu kahteen sovellusohjelmoijan määrittelemään funktioon. Map-funktiolle annetaan syötteenä avain-arvo-pari. Tämän jälkeen Map-funktio laskee välituloksena ohjelman syötteestä tietyt avain-arvo-parit. MapReduce-kehys hajauttaa sen jälkeen avaimen perusteella välituloksena saadut avain-arvo-parit Reduce-funktiolle. Reduce-funktio koostaa lopuksi samaan välituloksen avaimeen liittyvistä arvoista jonkin tuloksen. Tyypillisesti tuloksena on vain yksi arvo kutakin yksilöllistä avainta

kohti. MapReduce-laskentaa käsitellään tarkemmin jatkossa.

MapReduce ei siis ole kyselykieli, tietokanta eikä tietokannan hallintajärjestelmä. MapReduce-ohjelmointiparadigmaa voidaan kuitenkin hyödyntää esimerkiksi NoSQL-tietokantajärjestelmissä. Useimmissa uusissa NoSQL-tietokantajärjestelmissä on MapReduce-rajapinta laajan mittakaavan rinnakkaislaskentaa varten [BLS11]. Esimerkiksi Google Bigtable [CDG06, CDG08] voi hyödyntää MapReduce-ohjelmointiparadigmaa. Luvussa 7 esitetään myös toinen Googlen NoSQL-tietokantajärjestelmä, jossa on MapReduce-rajapinta. Apache Hadoop [The12b] käyttää myös MapReduce-ohjelmointiparadigmaan perustuvaa ratkaisua [CSD11, BGS11]. Hadoopiin perustuvat ratkaisut ovat esimerkiksi Yahoo!n [Yah12b] ja Facebookin [Fac12] käytössä.

Googlen palvelut, kuten hakukone [Goo12l] edellyttävät laajan mittakaavan raakatietokokonaisuuksien analysointia. Web-sivuihin liittyvistä dokumenteista ja lokeista joudutaan esimerkiksi laskemaan johdettua tietoa, kuten indeksejä sekä erilaisia tilastoja ja yhteenvetoja [DeG08, DeG10]. MapReduce-ohjelmointiparadigma sekä siihen liittyvä toteutus kehitettiin alun perin Google-hakukoneen [Goo12l] *käänteishakemistojen* (inverted index) rakentamisen yksinkertaistamiseen vuonna 2003. Käänteishakemistoja käytetään tehostamaan Google-hakukoneen hakuja. Tämän kaltaiset laskennat eivät usein ole käsitteellisesti välttämättä kovin monimutkaisia. Tietoja voidaan käsitellä yksinkertaisina avain-arvo-pareina. Valtavat tietomäärät asettavat kuitenkin laskennalle haasteita. Laskenta ei saa kestää liian kauan aikaa, joten tietojen laskenta hajautetaan sadoille tai jopa tuhansille tietokoneille suoritettavaksi samaan aikaan rinnakkain. MapReduce-rinnakkaislaskenta myös skaalautuu tarpeen mukaan. Laskentatehoa voidaan lisätä suorituskyvyn varmistamiseksi.

MapReduce-ohjelmointiparadigman ansiosta jollain yleisellä alemman tason ohjelmointikielellä voidaan kirjoittaa yksinkertainen ohjelma, jolla voidaan hyödyntää isoja resurssimääriä [DeG08]. MapReduce-kehyksen ajonaikainen järjestelmä hoitaa tietojen osittamisen, rinnakkaisen laskennan sekä mahdollisten virheiden käsittelyn. Sovellusohjelmoija kirjoittaa Map- ja Reduce-funktiot sekä määrittelee syöte- ja tulostiedostojen nimet ja valinnaiset parametrit.

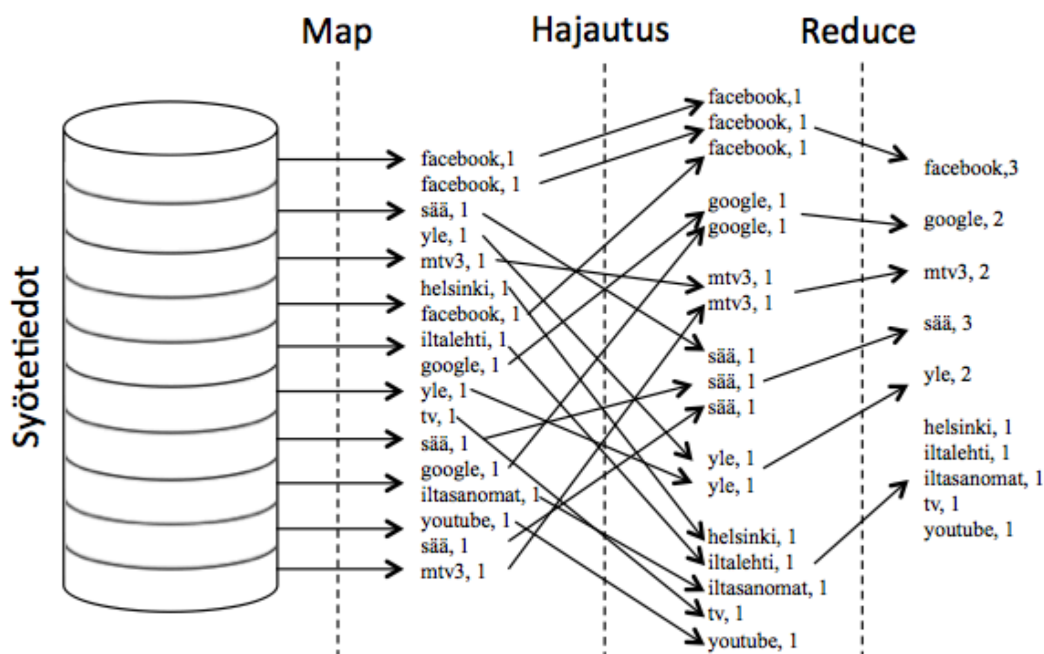
Sovellusohjelmoija kirjoittaa Map-funktion, jolla ilmaistaan syötetietojen avain-arvopari sekä haluttu joukko syötetiedoista laskettavia avain-arvo-pareja, kuten esimerkiksi avain ja siihen liitettävä tietty arvo [DeG08]. Map-funktiolle voidaan määritellä syöt-

teeksi esimerkiksi dokumentin nimi, joka edustaa avain-arvo-parin avainta sekä dokumentin muu sisältö, joka edustaa avain-arvo-parin arvoa. Map-funktio tuottaa sen jälkeen esimerkiksi jokaisen edellä määritellyn dokumentin sanan sekä siihen liittyvän sovellusohjelmoijan määrittelemän arvon. MapReduce-kehys hajauttaa avaimen perusteella nämä välituloksena saadut avain-arvo-parit sovellusohjelmoijan kirjoittamalle Reduce-funktiolle. Reduce-funktiolla ilmaistaan miten välituloksena saaduista avain-arvo-pareista lasketaan haluttu tulos. Tulos lasketaan usein koostamalla näistä avain-arvo-pareista jokin arvo, kuten esimerkiksi summattuna tietyn avaimen esiintymismäärä. Reduce-funktio voi esimerkiksi laskea yhteen tai koostaa jokaiseen yksilölliseen välituloksena saatuun sanaan liittyvät arvot. Tuloksena saadaan näin välituloksista muodostettu pienempi joukko arvoja. Syötetietoina olevat arvot ovat siis käsitteellisesti eri arvoalueesta kuin laskennan tuloksena saatavat arvot. Vastaavasti Map-funktion syötetiedoista laskemat arvot ovat taas samasta arvoalueesta kuin laskennan tuloksena saatavat arvot:

map (k1, v1) → list (k2, v2)

reduce (k2, list(v2)) → list (v2).

Tiedot välitetään Map- ja Reduce-funktioille jossain tietyssä muodossa, kuten esimerkiksi string-tietotyyppinä [DeG08]. Sovellusohjelmoijan tulee kuitenkin huomioida mahdolliset tietotyypin muunnokset funktioiden koodissa. Sovellusohjelmoijan on myös lisäksi määriteltävä syöte- ja kohdetiedostot sekä muut MapReduce-ohjelmaa koskevat parametrit. Kuvassa 9 esitetään erään MapReduce-ohjelman toiminta. Ohjelmassa lasketaan jonkun web-hakupalvelun avulla eniten haettuja sanoja web-hakupalvelun lokitiedostosta. Map-funktiolle annetaan avaimena hakupalvelun lokitiedoston nimi sekä siihen liittyvänä arvona lokitiedoston sisältö. Map-funktio tuottaa jokaisen lokitiedostosta löytyvän hakusanan sekä siihen liittyvän arvon, joka tässä tapauksessa on luku 1. MapReduce-kehys hajauttaa Map-funktion laskemat avain-arvo-parit. Sen jälkeen Reduce-funktiolla lasketaan yhteen kaikki tiettyyn sanaan liittyvät arvot, jolloin lopputuloksena saadaan kunkin sanan esiintymismäärä web-hakupalvelun lokitiedostossa.



Kuva 9: Erään MapReduce-ohjelman toiminta [Goo12e].

MapReduce-rinnakkaislaskennassa syötetiedot s ositetaan automaattisesti n *osaan* (split) s s_0, s_1,…, s_{n-1} [DeG08]. Map-funktion suoritus jaetaan yhtä moneen osaoperaatioon M M_0, M_1,…, M_{n-1}, jotka suoritetaan rinnakkain tietokonerypään solmuissa. Tämä useassa solmussa käynnistetty *Map-ilmentymä* (map invocation) M*i* käsittelee sille osoitetun osan s*i* syötetiedoista. Map-funktion laskemat väliaikaiset avain-arvo-parit (k2, v2) puskuroidaan muistiin, josta ne hajautetaan solmun yksityislevyjärjestelmän paikalliselle levylle ositettuna m *alueeseen* (region) r r_0, r_1,…, r_{m-1} avaimen k2 perusteella.

Reduce-funktion suoritus jaetaan m osaoperaatioon R R_0, R_1,…, R_{m-1} , jotka suoritetaan rinnakkain tietokonerypään solmuissa [DeG08]. Tämä useassa solmussa käynnistetty *Reduce-ilmentymä* (reduce invocation) R*i* käsittelee sille osoitetun alueen r*i* Map-funktion laskemista tiedoista. Tiedot luetaan solmujen paikallisilta levyiltä etäproseduurikutsuilla. Reduce-funktio koostaa samaan avaimeen k2 liittyvistä arvoista jonkin tuloksen ja kirjoittaa avaimen sekä siihen liittyvän tuloksen tiedostoon. Koko operaation lopputuloksena on yhtä monta m tiedostoa F F_0, F_1,…, F_{m-1}, kuin Reduce-ilmentymän suorituksia. Operaation lopputuloksena siis syntyy useampia tiedostoja. Jos laskennan lopputulos halutaan kuitenkin vain yhteen tiedostoon, on nämä F_{m-1} tiedostoa yhdistettävä tähän yhteen tiedostoon erillisenä prosessina.

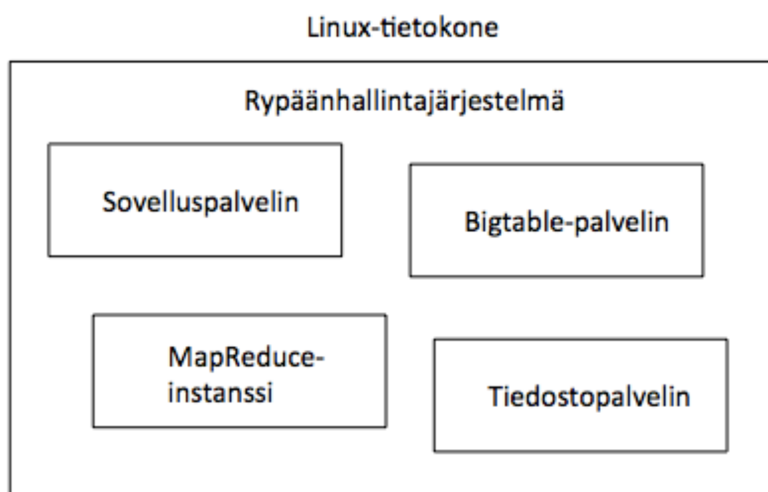
Ohjelman Map- ja Reduce-ilmentymiä kutsutaan *työläisiksi* (worker) [DeG08]. Sovellusohjelman käynnistämä MapReduce-ohjelman *isäntä* (master) ilmentymä koordinoi

ohjelman suoritusta, käynnistää työläiset sekä resursoi niille käsiteltävät tiedot. Isäntä pitää kirjaa jokaisen Map- ja Reduce-tehtävän tilasta sekä työläissolmun identiteetistä. Isäntä tallentaa Map-ilmentymien laskemien avain-arvo-parien levyalueiden r sijainti- ja kokotiedot. Nämä tiedot välitetään työläisille, joilla on käynnissä olevia Reduce-ilmentymän suorituksia. Sovellusohjelmoija voi myös seurata laskennan tilaa isännän ylläpitämien tilaraporttien avulla. Raporttien avulla voidaan esimerkiksi seurata työläisten tilaa, käsiteltävien tietojen kokoa ja prosessin vauhtia sekä valmistuneita ja keskeneräisiä Map- ja Reduce-tehtäviä. Näiden tietojen perusteella voidaan arvioida kuinka kauan aikaa laskenta kestää. Laskennan tilatietoja voidaan myös hyödyntää ohjelman mahdollisten virheiden jäljityksessä.

Laajan mittakaavan hajautetuissa tietokonerypäissä suoritettavan ohjelman tulee varautua erilaisiin virhetilanteisiin. Alkuperäinen Google MapReduce -toteutus soveltaa virheistä toipumiseen pääosin ohjelman *uudelleen suoritusta* (re-execution) [DeG08]. MapReduce-ohjelman isäntä-instanssi koordinoi ohjelman suoritusta tarkistamalla työläisten toimivuuden säännöllisesti. Jos työläinen ei vastaa isännän kutsuun tietyn ajan kuluessa, se hylätään. Kaikki hylätyllä työläisellä kesken olleet Map- ja Reduce-tehtävät sekä valmistuneet Map-tehtävät palautetaan niiden alkuperäiseen tilaan. Kaikki nämä tehtävät suoritetaan uudestaan jossain toimivassa solmussa. Valmistuneita Reduce-tehtäviä ei kuitenkaan tarvitse suorittaa uudestaan, koska niiden tulokset tallennetaan hajautettuun tiedostojärjestelmään. Isäntää koskevissa virhetilanteissa koko ohjelman suoritus hylätään, jolloin asiakassovelluksen täytyy käynnistää se uudelleen. Uudelleen suorituksen ansiosta pystytään toipumaan laajoistakin, useamman työläissolmun virhetilanteesta. Dean ja kumppanit [DeG08] artikkelissa on annettu esimerkki jopa 80 solmun hylkäämisestä kerrallaan. Tässäkin tapauksessa hylättyjen solmujen tehtävät vain suoritettiin uudestaan.

Alkuperäinen Google MapReduce -toteutus on riippuvainen muista Googlen järjestelmistä [DeG08]. *Google-rypäänhallintajärjestelmä* (Google Cluster Management System) hoitaa Google-tietokonerypäässä resurssien hallinnoinnin, seurannan ja ongelmien selvittelyn sekä tehtävien jakamisen, ajastamisen ja suorittamisen [CDG06, CDG08, DeG08]. Rypään tietokoneissa voi MapReduce-instanssin lisäksi toimia myös useiden muiden Googlen eri järjestelmien prosesseja. Kuvassa 10 esitetään Google-tietokonerypään yksittäisessä koneessa mahdollisesti toimivia Googlen eri järjestelmien prosesseja.

Kuvassa on yksittäinen Linux-tietokone, jossa toimii samaan aikaan sovelluspalvelin, Bigtable-palvelin, MapReduce-instanssi sekä hajautettu tiedostojärjestelmä.



Kuva 10: Google-tietokonerypään tietokoneessa toimivia prosesseja [CDG06, CDG08].

MapReduce-ohjelma voi hyödyntää esimerkiksi tietokantaa tai tiedostojärjestelmää syötetietojen lähteenä tai laskennan lopputuloksen kohteena. Luvussa 7 on esimerkki miten MapReduce-ohjelma voi hyödyntää tietokantaa. Alkuperäinen Google MapReduce -toteutus hyödyntää GFS-tiedostojärjestelmää, jonka avulla MapReduce-ohjelmien tiedostoja toisinnetaan useamman solmun levylle. GFS-tiedostojärjestelmää käsiteltiin edellä. Sen käyttö parantaa tietojen saatavuutta sekä järjestelmän kestävyyttä. Map-tehtävässä yritetään aina hyödyntää solmua, joka on lähinnä tehtävässä käytettävän syötetiedoston toisinnetta [DeG08]. Näin syötetietojen luku on aina mahdollisimman tehokasta. MapReduce-ohjelman suorituksen tulos toisinnetaan myös useampaan tiedostoon. Dean ja kumppanit [DeG08] artikkelissa mainitaan, että GFS:n käyttö vaikuttaa kuitenkin MapReduce-ohjelmien suorituskykyyn. Jos syötetiedot on jaettu moneen tiedostoon, näiden tiedostojen avaaminen ja käsittely aiheuttaa viivettä. Edellä mainittu tuloksen toisintaminen aiheuttaa myös viivettä. Googlen lisäksi ainakin Apache Hadoop [The12b] käyttää MapReduce-toteutuksensa taustalla GFS:n kaltaista hajautettua *HDFS-tiedostojärjestelmää* (Hadoop distributed file system, HDFS).

MapReduce-kehystä hyödyntävän järjestelmän kestävyyttä ja suorituskykyä parantaa Map- ja Reduce-vaiheiden osittaminen useampaan osaan, jotka voidaan suorittaa eri solmuissa samaan aikaan. MapReduce-tehtäviä ei resursoida tietyille solmuille etukäteen esimerkiksi missään laskentastrategiassa [DeG08]. Yksittäinen solmu voi suorittaa

eri tehtäviä, jotka resursoidaan sille ohjelman suorituksen aikana. Tämän ansiosta solmujen kuormaa voidaan tasata tosiaikaisesti. Näin myös hylätyn työläisen tehtävät voidaan jakaa kaikkien muiden työläisten kesken. Jos taas jokin työläissolmu suoriutuu sille osoitetusta tehtävästä hitaasti, sen keskeneräinen tehtävä osoitetaan suoritettavaksi samaan aikaan myös muille vapaille solmuille. Tehtävän suoritus on valmis, kun ensimmäinen näistä solmuista saa tehtävän suoritettua.

7 NoSQL-tietokantapalveluna

Monet suuret Internet-yritykset, kuten esimerkiksi Google ja Amazon [Ama12a] hyödyntävät hajautettua tietokantajärjestelmää ja sitä varten rakentamaansa laajan mittakaavan infrastruktuuria tarjoamalla sitä myös muiden yritysten Web 2.0 -sovelluksien alustaksi palveluna [SKS11, s. 861-862]. Tässä luvussa kerrotaan tästä palvelukonseptista sekä esitellään Googlen web-sovellusten ylläpitopalvelu [Goo12b], jonka kautta muut yritykset voivat hyödyntää esimerkiksi Googlen NoSQL-tietokantajärjestelmää sekä sen taustalla toimivaa laajan mittakaavan infrastruktuuria. Luvun lopuksi vielä analysoidaan kyseisen palvelukonseptin käyttömahdollisuuksia.

7.1 Pilvipalvelujen yleiskuvaus

Pilvilaskennan konsepti (cloud computing concept) tarkoittaa tietotekniikan, kuten tietokoneiden ja palvelimien tarjoamista asiakkaalle *pilvipalveluna* (cloud service) [SKS11, s. 861-862]. Asiakas voi palvelussa esimerkiksi itse määritellä kuinka paljon kapasiteettia, kuten laskentatehoa tai levytilaa tarvitsee. Pilven kapasiteettia voidaan myös lisätä tai vähentää asiakkaan oman tarpeen mukaan eli kyseessä on skaalautuva palvelu. Pilvilaskentaa tarjotaan erilaisina malleina. Asiakas voi hyödyntää pilven palveluja jonkin etukäteen kyseisiä palveluja tarjoavan yrityksen kanssa sovitun konseptimallin mukaan.

Pilvilaskennan konseptimallit voidaan jakaa kolmeen eri kategoriaan. *Infrastruktuuri palveluna, IaaS* (Infrastructure as a Service, IaaS) palvelussa asiakas voi hyödyntää palveluntarjoajan resursseja, kuten prosessointi- ja muistikapasiteettia, varastointia ja tietoverkkoja [Kim11, Lee10]. Asiakas voi asentaa tähän ympäristöön mitä tahansa näitä resursseja hyödyntäviä sovelluksia ja järjestelmiä, kuten esimerkiksi käyttöjärjestelmiä. Asiakas voi myös hallinnoida näitä asennettuja sovelluksia ja järjestelmiä. Asiakas

ei kuitenkaan pääosin hallinnoi palveluntarjoajan varsinaista infrastruktuuria. *Alusta palveluna, PaaS* (Platform as a Service, PaaS) palvelussa asiakas voi hyödyntää omia sovelluksiaan palveluntarjoajan infrastruktuurissa. Asiakas voi asentaa jonkun muun kuin palveluntarjoajan kehittämiä sovelluksia palveluntarjoajan infrastruktuuriin. Nämä sovellukset on kuitenkin kehitetty palveluntarjoajan hyväksymillä ohjelmointikielillä ja työkaluilla. Asiakas voi myös hallinnoida omia sovelluksiaan ja joitakin rajoitettuja sovelluksen ympäristöön vaikuttavia asetuksia. Asiakas ei kuitenkaan hallinnoi palveluntarjoajan infrastruktuuria. *Sovellus palveluna, SaaS* (Software as a Service, SaaS) palvelussa asiakas voi hyödyntää palveluntarjoajan sovelluksia, jotka hyödyntävät palveluntarjoajan resursseja. Näitä sovelluksia voidaan hyödyntää esimerkiksi asiakkaalle tarjottavan web-käyttöliittymän kautta. Asiakas ei kuitenkaan pääosin hallinnoi näitä sovelluksia. Asiakas ei myöskään hallinnoi palveluntarjoajan infrastruktuuria.

Pilvi voidaan toteuttaa useamman erilaisen toteutusmallin mukaan. Eräs sellainen on *Yksityinen pilvi* (private cloud), jossa pilven koko infrastruktuuria hyödyntää vain yksi taho [Kim11, Lee10]. Tämä taho voi myös hallinnoida pilven ohjelmia ja infrastruktuuria. Hallinnointi voi olla myös kokonaan ulkoistettu. Pilven infrastruktuuri voi olla hyödyntävän tahon oma tai ulkopuolisen omistuksessa. *Yhteisöpilvi* (community cloud) on taas toteutus, jossa pilven koko infrastruktuuria hyödyntää useampi taho joilla on joku yhdistävä tekijä, kuten päämäärä, vaatimukset tai aate. Nämä tahot voivat myös hallinnoida pilven ohjelmia ja infrastruktuuria. Hallinnointi voi olla myös kokonaan ulkoistettu. Pilven infrastruktuuri voi olla hyödyntävien tahojen oma tai ulkopuolisen omistuksessa. *Julkinen pilvi* (public cloud) on edellisistä toteutuksista poiketen jonkun yrityksen tai organisaation omistama infrastruktuuri, jota tarjotaan laajan heterogeenisen joukon hyödynnettäväksi palveluna. *Hybridipilvissä* (hybrid cloud) useampi jollakin edellä mainitulla toteutusmallilla toteutettu pilvipalvelu toimii jollakin tavalla yhdessä, esimerkiksi skaalautumalla tarpeen vaatiessa toiseen pilveen.

Monet pilvipalveluja tarjoavat yritykset tarjoavat SaaS- tai PaaS-palveluna esimerkiksi varastointia, karttapalveluja sekä muita palveluja, joita asiakas voi hyödyntää palvelun ohjelmointirajapinnan kautta. Tämän kaltaisia varastointia tarjoavia pilviperustaisia varastointijärjestelmiä kutsutaan Silberschatz ja kumppanit [SKS11, s. 861-862] kirjassa *pilviperustaisiksi tietokannoiksi* (cloud based databases). Yritykset voivat hyödyntää myös NoSQL-tietokantajärjestelmää jonkin muun yrityksen tarjoamassa pilvipalvelussa.

Yritykset voivat hyödyntää jo olemassa olevaa laajan mittakaavan hajautettua tietokantajärjestelmää, ja sen tietokonerypäiden laitteisto- ja ohjelmistoresursseja *NoSQL-tietokantapalveluna* (NoSQL database service) [Vog12]. Tässä palvelussa tarjotaan yrityksen sovellukselle hyvin skaalautuva ja kuormaa tasaava tietokantajärjestelmä ja sen infrastruktuuri, jonkin pilvilaskennan konseptimallin mukaisesti. Tämän tietokantajärjestelmän tietokannan tietomalli saattaa poiketa relaatiomallista, kuten palvelua kuvaava nimi viittaa. Näiden järjestelmien hinnoittelu perustuu usein sovelluksen käyttöön tarjottavien resurssien määrään, kuten tietojen tallennustilan kapasiteettiin. Joitakin palveluja voi myös käyttää ilmaiseksi tiettyjen rajoitteiden puitteissa. Tällaista palvelua tarjoavat usein sivutuotteenaan isot Internet-yritykset, kuten Google [Goo12b] ja Amazon [Ama12b], jotka ovat rakentaneet laajan mittakaavan infrastruktuurin omia palvelujaan varten.

7.2 Google App Engine

Google App Engine, GAE (Google App Engine, GAE) on pilvilaskennan PaaS-konseptimallin mukaan toimiva web-sovellusten ylläpitopalvelu [Goo12b]. Tämä palvelu tarjoaa sovellusohjelmoijalle web-käyttöliittymän, jonka kautta voidaan hyödyntää Googlen olemassa olevaa laajan mittakaavan infrastruktuuria, kuten tietokonerypäitä, Bigtable-tietokantajärjestelmää [CDG06, CDG08], ja MapReduce-laskentaa [DeG08]. Sovellukset ladataan GAE:n sovelluspalvelimelle palvelun omien työkalujen avulla. Palvelussa voidaan käyttää vain tiettyjä ohjelmointikieliä Googlen määrittelemien ohjelmointirajapintojen ja ohjelmistokehysten kautta [MKW11]. Näitä ohjelmointirajapintoja hyödynnetään yleensä jollain alemman tason ohjelmointikielellä. Palvelussa voidaan esimerkiksi käyttää Javaa ja Pythonia sekä niiden ajonaikaista ympäristöä [Goo12b]. GAE:n Java-ympäristö toimii rajoitetusti esimerkiksi *Spring-ohjelmistokehyksen* (Spring framework) [Spr12a] ja *Struts-ohjelmistokehyksen* (Struts framework) [The12f] kanssa [Goo12b]. Spring- ja Struts-ohjelmistokehykset ovat avoimen lähdekoodin ohjelmistokehyksiä, joiden avulla voidaan tehdä Java-ohjelmia [Spr12a, The12f]. GAE:n Python-ympäristö toimii useimpien Pythonin web-ohjelmistokehysten kanssa [Goo12b]. Sovellusohjelmoija voi myös hyödyntää Googlen muiden sovelluksien ja palvelujen ohjelmointirajapintoja, kuten Google Gmail-sähköpostipalvelua [Goo12j]. Jos sovelluksen käyttöä halutaan rajoittaa, käyttäjien tunnistautumiseen voidaan esimerkiksi käyttää *Google Accounts -tunnistautumispalvelua* (Google Accounts sign-in service) [Goo12a].

Google Accounts on Googlen keskitetty käyttäjien tunnistautumispalvelu. Palvelun käyttäjä voi avata palvelussa *Google käyttäjätilin* (Google Account), jonka käyttäjätunnuksen ja salasanan avulla käyttäjä voi kirjautua Googlen palveluihin, kuten Gmail [Goo12j], YouTube [Goo12m] ja Google+ [Goo12n]. Kehitteillä on myös testikäytössä oleva GAE:lle optimoitu MapReduce-ohjelmointirajapinta *App Engine MapReduce* [Goo12e]. App Engine MapReduce on eräs alkuperäisen Google MapReduce -kehyksen [DeG08] toteutus. Jotta sovellusohjelmoija voi kontrolloida sovelluksen ajonaikaisia kustannuksia, App Engine MapReduce -kehyksen suorituskykyä on kuitenkin rajoitettu. Googlen pilvipalvelussa voidaan käyttää useampaa tapaa tietojen varastointiin. Sovellus voi hyödyntää tietokantakerroksena *App Engine Datastore*, *Google Cloud SQL* tai *Google Cloud Storage* -varastorakenteita [Goo12b]. App Engine Datastore perustuu Google Bigtableen. App Engine Datastorea käsitellään tarkemmin jatkossa. Google Cloud SQL perustuu MySQL-relaatiotietokantajärjestelmään [Ora12e] Google Cloud Storage taas on GAE:n testikäytössä oleva varastorakenne, joka on tarkoitettu erittäin suurten tiedostojen tallentamiseen. Näiden lisäksi palvelussa voidaan hyödyntää keskusmuistissa toimivaa *Memcached-välimuistijärjestelmää* (Memcached caching system) [Dor12] väliaikaisten tietojen varastointiin. Memcached on tehokas keskusmuistissa sijaitseva välimuistin tapaan toimiva avoimeen lähdekoodiin perustuva avain-arvo-varasto. Memcached on tarkoitettu tehostamaan dynaamisten web-sivujen käyttöä. Tieto varastoidaan väliaikaisesti muistiin niin, että tietoja ei tarvitse hakea joka kerta pysyvästä tietokannasta.

GAE-pilvipalvelua on helppo käyttää [HoP10]. Palvelussa on yksinkertainen web-hallintakonsoli, jonne sovellusohjelmoija voi kirjautua selaimen kautta Google Account -käyttäjätunnuksella [Goo12a]. Käyttäjä voi kirjautua palveluun esimerkiksi omalla olemassa olevalla Gmail-sähköpostitilin [Goo12j] käyttäjätunnuksella ja salasanalla. Sovellusohjelmoijan tekemä ohjelma ladataan palveluun hallintakonsolin kautta [Goo 12b]. Tämän jälkeen sovellusta pääsee käyttämään esimerkiksi http://sovelluksen tunnus.appspot.com muotoisen URL-osoitteen kautta. Kaikki sovellusohjelmoijan palveluun lataamat ohjelmat näytetään hallintakonsolin käyttöliittymässä myös listana, jonka linkeistä pääsee hallinnoimaan yksittäisen sovelluksen asetuksia. Näitä asetuksia ovat esimerkiksi sovelluksen nimen muutokset, resurssien käyttö, käyttöoikeudet, lisäpalvelujen asetukset, domainmuutokset ja tietokantajärjestelmän asetukset. Hallintakonsolin kautta pääsee myös katsomaan sovelluksen käynnissä olevien *instanssien* (instances)

tietoja. GAE:n ladatut sovellukset skaalautuvat instanssien avulla. GAE käynnistää sovelluksesta useampia instansseja sitä mukaan kun tarve vaatii. Kaikki sovellukseen kohdistuvat pyynnöt jakautuvat näiden instanssien kesken. Google tarjoaa myös *sovelluskehityspaketin, SDK* (software development kit, SDK) kullekin tuetulle ohjelmointikielelle. Sovelluskehityspaketti sisältää yleensä työkaluja ja ympäristön jonkun tietyn tyyppisen ohjelman luomiseen. GAE:n sovelluskehityspaketti sisältää esimerkiksi web-palvelimen jonka avulla voidaan jäljitellä App Enginen palveluja sekä ympäristöä paikallisella tietokoneella. GAE:n sovelluskehityspaketti sisältää myös työkalun, jolla valmis sovellus voidaan ladata palveluun. GAE:n käyttöä ei ole sidottu paikkaan eikä aikaan. Palvelu on jatkuvasti saatavilla ja sitä voi käyttää miltä tahansa laitteelta, jossa on toimiva Internet-yhteys. Palvelun hyvän saatavuuden takaa GAE:n taustalla toimiva Googlen laajan mittakaavan skaalautuva infrastruktuuri, jossa tietoja toisinnetaan moneen tietokonerypään solmuun.

7.3 Resurssien hyödyntäminen

PaaS-konseptimallin mukaisessa pilvipalvelussa ylläpidettävät ohjelmat suoritetaan usein muista sovelluksista eristetyssä *hiekkalaatikkoympäristössä* (sandbox). Tässä ympäristössä sovelletaan hiekkalaatikkotekniikkaa, jossa tietokoneen prosessit suoritetaan kontrolloidusti toisistaan eristettynä [Goo12h]. Tässä tekniikassa yleensä rajoitetaan järjestelmän resursseja joita kukin prosessi saa käyttää. Hiekkalaatikko eristää sovelluksen paikasta, laitteistosta ja käyttöjärjestelmästä riippumattomaan luotettavaan ja turvalliseen ympäristöön. Kun ohjelmat ajetaan toisistaan eristettynä, ne eivät vaikuta toistensa suoritukseen. Hiekkalaatikkotekniikkaa käytetään usein koodin tai epäluotettavien ohjelmien testaamiseen. GAE-pilvipalvelussa tämän järjestelyn tarkoituksen on esimerkiksi parantaa järjestelmän tietoturvaa ja tehokkuutta [Goo12b]. Sovellukset eivät voi kirjoittaa tiedostojärjestelmään, vaan kaikki pysyvät tiedot on tallennettava GAE:n varastorakenteisiin. Tiedostoon kirjoittavien luokkien käyttö ei ole sallittu. Esimerkiksi Javan java.io.FileWriter -luokan käyttö ei ole mahdollista. Tietojen lukeminen tiedostojärjestelmästä on kuitenkin sallittu. Toista Internet-palvelinta ei voi käyttää suoraan, vaan kaikki pyynnöt välitetään GAE:n *pyyntöjenvälitysrajapinnan* (The URL Fetch Java API) kautta. Tämän ohjelmointirajapinnan avulla voidaan välittää http- ja https-pyyntöjä muille Internet-palvelimille. Sovelluksien pitää myös toimia tehokkaasti. Jos sovellus ei vastaa web-pyyntöön nopeasti, web-palvelin voi kuormittua liikaa. Sen takia

kaikki hitaasti toimivat sovelluksen prosessit lopetetaan automaattisesti. Tämän lisäsi myös muunlaisia sovelluksen tekemiä proseduurikutsuja on rajoitettu.

NoSQL-tietokantapalveluna tarjoaa asiakkaalle mahdollisuuden hyödyntää laajan mittakaavan resursseja. Käyttäjän ei tarvitse hankkia ympäristön käyttöön tarvittavia laitteita tai ohjelmia. Kustannuksia säästyy myös, koska käyttäjän ei tarvitse rakentaa käyttöympäristöä ennen kuin palvelua pääsee käyttämään. Laitteita tai ohjelmia ei tarvitse siis erikseen asentaa ja konfiguroida ennen käyttöä. Käyttäjän ei myöskään tarvitse huolehtia palvelun tai käyttöympäristön ylläpidosta. Käyttöympäristön päivitykset, varmistukset sekä ongelmien selvittely ja korjaukset kuuluvat palvelun ylläpitäjän vastuulle. Käyttöön tarjottava infrastruktuuri hoitaa tietojen hajautukseen, toisintamiseen ja kuorman tasaukseen liittyvät tehtävät. Sovellukset voivat esimerkiksi hyödyntää Googlen laajan mittakaavan infrastruktuurin ominaisuuksia, kuten kuorman tasausta ja automaattisesti skaalautuvia resursseja. GAE lisää käytettävissä olevia resursseja automaattisesti sovellukseen kohdistuvien web-pyyntöjen lisääntyessä [Goo12b]. Näin web-pyyntöjä voidaan käsitellä enemmän samaan aikaan rinnakkain. Kapasiteettia lisätään automaattisesti kuitenkin vain tehokkaasti toimiville sovelluksille, joiden vasteaika on alle yhden sekunnin. Hitaammin toimivia sovelluksia rajoitetaan palvelun tehokkuuden ja vakauden varmistamiseksi. Jos resursseja lisätään myös hitaiden prosessien kohdalla, hitaita prosesseja voidaan suorittaa enemmän samaan aikaan rinnakkain, joka entisestään hidastaa koko järjestelmää. Googlelta on kuitenkin mahdollista pyytää poikkeusmenettelyä, jos myös joillekin hitaille prosesseille tarvitaan automaattisesti skaalautuvia resursseja.

PaaS-konseptimallin mukaisessa pilvipalvelussa hyödynnettävillä resursseilla on usein käytössä myös *resurssikiintiöitä* (resource quotas), joita yksittäiset tai tietyn asiakkaan sovellukset eivät saa ylittää. Kiintiöiden avulla kontrolloidaan, että yksittäinen sovellus ei voi kuluttaa liikaa yhteisen infrastruktuurin resursseja. Muussa tapauksessa vapaita resursseja voi jäädä liian vähän muiden sovellusten käyttöön. GAE:n tietokantajärjestelmää koskevia yleisiä kiintiöityjä resursseja ovat esimerkiksi tietokantaan varastoitujen tietojen datamäärät ja tietokannan indeksien lukumäärät, sekä tietokantaan kohdistuvien kirjoitus- ja lukuoperaatioiden sekä pienten operaatioiden lukumäärät [Goo12b]. App Engine Datastore käyttöä varten on lisäksi omia kiintiöitä ja rajoituksia. Datastore-ohjelmointirajapintaa hyödyntävillä sovelluksen kutsuilla on resurssikiintiöt. Sovellus

voi kutsua Datastore-ohjelmointirajapintaa vain tietyn lukumäärän verran. Datastore-ohjelmointirajapinnan kautta voi myös siirtää tietoa vain tietyn datamäärän. Tietokantaan tallennetun yksittäisen rivin, sekä tietokantaan operoivan transaktion kokoa on lisäksi rajoitettu. Datastoreen voi myös määritellä vain tietyn luku- ja datamäärän yksittäisen tietokannan rivin tietoja koskevia indeksirivejä. Datastore-ohjelmointirajapinta ei ole ainoa ohjelmointirajapinta, jonka käyttöä rajoitetaan kiintiöillä. Kiintiöillä rajoitetaan useimpia GAE:n kautta käytettäviä resursseja.

GAE-palvelussa on käytössä kokonaiskiintiöt sekä päivittäiset ja minuuttikohtaiset kiintiöt [Goo12b]. Kokonaiskiintiöt koskevat GAE:n resurssien yli 24 tunnin käyttöä. Päivittäiset kiintiöt tarkoittavat vuorokauden aikana sovelluksen kuluttamia GAE:n resursseja. Minuuttikohtaiset kiintiöt tarkoittavat kuinka paljon sovellus voi kuluttaa GAE:n resursseja minuutin aikana. Minuuttikohtainen kiintiö estää sovellusta dominoimasta jonkin resurssin käyttöä, esimerkiksi kuluttamalla kaikki resurssinsa hyvin lyhyen ajan kuluessa. Jos sovellus ylittää jonkin sille määritellyn kiintiön, sen toiminta lakkaa kunnes saatavilla on taas lisää resursseja. GAE:ssa jokainen sovellus saa tietyn määrän ilmaisia resursseja. Sovellusohjelmoija voi esimerkiksi ladata palveluun ilmaiseksi enintään kymmenen sovellusta, joista kustakin voi olla kymmenen eri versiota. Sovellukset voivat kuluttaa ilmaiseksi 6.5 prosessorituntia vuorokauden aikana. Verkkoliikennettä saa käyttää tietyn määrän vuorokauden aikana. Vuorokauden aikana on mahdollista myös lähettää sähköpostia korkeintaan sadalle vastaanottajalle. Näiden lisäksi ilmaiseen käyttöön kuuluu yhteensä yksi gigatavu tallennustilaa, 200 tietokantaindeksiä, 50000 jokaista erityyppistä tietokantaoperaatiota sekä maksimissaan viisi miljoonaa sivuosumaa kuukaudessa palvelussa ylläpidettävän sovelluksen web-sivuille. Edellä mainittiin vain osa kaikista rajoituksista. Erilaisia resurssikiintiöitä ja rajoituksia on vielä paljon enemmän. Palveluun ladattujen sovellusten resurssikiintiöitä voi kuitenkin nostaa ostamalla isompia kiintiöitä, jos esimerkiksi tallennettava datamäärä tai tietokantaoperaatioiden määrä kasvaa. Maksetuilla resurssikiintiöillä on kuitenkin myös olemassa tietyt kattorajat resurssien kulutukselle.

7.4 Google App Engine Datastore

Google App Engine -pilvipalvelussa ylläpidettävä sovellus voi hyödyntää tietokantakerroksena *App Engine Datastorea* [Goo12b]. App Engine Datastore on Web 2.0 -sovellusten käyttöön kehitetty skaalautuva NoSQL-tietokantapalvelu. Datastore-tieto-kanta on skeematon. Tiedon rakennetta ylläpidetään sovelluslogiikassa. App Engine Datastore -tiedot on varastoitu Google Bigtable -tauluihin [CDG06, CDG08]. Kaikkien App Engine Datastorea käyttävien sovellusten tietosisältö varastoidaan yhteen Bigtable-tauluun [Goo12b]. Tietomallin toteutus on kuitenkin erilainen kuin alkuperäinen Bigtable-tietomalli. App Engine Datastore -taulun rakenne koostuu olioista joita kutsutaan *entiteeteiksi* (entity). Entiteettien rakennetta ylläpidetään sovelluslogiikassa. Taulu koostuu riveistä ja sarakkeista:

$$(\text{riviavain, sarakeavain}) \rightarrow \text{sisältö.}$$

Jokaisella rivillä eli entiteetillä voi olla yksi tai enemmän sarakkeita, joita kutsutaan *ominaisuuksiksi* (properties) [Goo12b]. Entiteetille määritellään entiteetin *laji* (kind). Entiteetit luokitellaan lajeihin esimerkiksi tietokantakyselyjä varten. Entiteetin laji voi olla esimerkiksi nimeltään työntekijä. Tämän lisäksi tietorakenne on hierarkkinen. Entiteetit voidaan järjestää tiedostojärjestelmän kaltaiseen puurakenteeseen. Entiteetti voidaan määritellä pysyvästi toisen entiteetin lapseksi. Sovellusohjelmoija voi esimerkiksi tehdä tämän määrityksen entiteetin luonnin yhteydessä. Jos entiteettiä ei määritellä toisen entiteetin lapseksi, kyseessä on juurientiteetti. Näin syntyy hierarkkinen puurakenne, jossa entiteetillä voi olla rekursiivisesti esivanhempia ja jälkeläisiä. Entiteetti ja sen jälkeläiset muodostavat *entiteettiryhmän* (entity group). Entiteetin laji- ja entiteettiryhmäsarakkeita kutsutaan taas entiteetin metatiedoiksi.

Entiteetin yksilöivä riviavain muodostuu sovellustunnisteen ja polun yhdistelmästä [Goo12b]. Sovellustunniste yksilöi entiteetin luoneen sovelluksen. Polku muodostuu katenoiduista entiteettiavaimista juurientiteetistä entiteettiin. Entiteettiavain taas muodostuu entiteetin lajista sekä tunnisteesta. Tunniste voi olla string tai int64 -tietotyyppiä. Sarakeavain muodostuu entiteetin ominaisuudesta, jolla on nimi. Riviavaimen ja sarakeavaimen yhdistelmä viittaa varsinaiseen tietosisältöön eli taulun yksittäisen soluun. Yksittäiseen soluun voidaan tallentaa yksi tai useampia arvoja. Kukin arvo vastaa yhtä tietotyyppiä. Tietotyyppinä voi olla esimerkiksi entiteettiavain, jolloin ominaisuuden arvo viittaa toiseen entiteettiin. Ominaisuudelle voidaan tallentaa useampi arvo jonkun

tietojen sarjallistamiseen tarkoitetun formaatin avulla. Kullakin entiteettirivillä on jokaisen ominaisuuden nimi ja arvot. Entiteettirivillä on myös metatietosarakkeet laji ja entiteettiryhmä.

Kuvassa 11 on esitetty App Engine Datastore -taulun siivu. Esimerkkisovellus varastoi tauluun työntekijöiden tietoja. Työntekijät on järjestetty esimies-alainen-hierarkiaan. Janne on työntekijöiden esimies sekä hierarkian juurientiteetti. Rivin avain on sovellustunnisteen ja polun yhdistelmä. Muut sarakkeet sisältävät ominaisuudet sekä metatiedot laji ja entiteettiryhmä. Entiteetin laji on esimies tai työntekijä ja entiteettiryhmä on /Esimies:Janne, joka on siis työntekijöiden esimies. Ominaisuudet ovat nimi, ikä ja työpari. Nimi on työntekijän etunimi. Ikä voidaan esittää kahdella tavalla eri tietotyyppien avulla. Työpari taas voi viitata toisiin entiteetteihin. Jannella ei ole työparia, Annella on yksi työpari ja Marilla on kaksi työparia.

Avain		Metatiedot		Ominaisuudet			
Sovellus tunniste	Polku	Laji	Entiteettiryhmä	Nimi	Ikä	Työpari	
1	/Esimies:Janne	Esimies	/Esimies:Janne	Janne	int:47		
1	/Esimies:Janne/ Työntekijä:Anne	Työntekijä	/Esimies:Janne	Anne	int:30	Avain:/Työntekijä:Mari	
1	/Esimies:Janne/ Työntekijä:Mari	Työntekijä	/Esimies:Janne	Mari	double: 28.5	Avain:/Työn tekijä:Anne	Avain:/Työn tekijä:Liisa

Kuva 11: App Engine Datastore -taulun siivu [Goo12b].

App Engine Datastoressa voidaan hyödyntää esimerkiksi tietokannan käsittelyyn tarkoitettuja standardeja rajapintoja, kuten *Java Data Objects, JDO* (Java Data Objects, JDO) ja *Java Persistence API, JPA* (Java Persistence API, JPA) [Goo12b]. JDO- ja JPA-rajapintojen kautta voidaan käsitellä erilaisia tietokantoja olio-ohjelmointimallin mukaisesti. Käsiteltävän tietokannan ei tarvitse olla oliotietokanta. App Enginen Datastorella on kuitenkin myös oma ohjelmointirajapinta, jonka kautta asiakassovellus voi lukea, kirjoittaa ja poistaa entiteettejä alemman tason ohjelmointikielellä. Nämä operaatiot voidaan toteuttaa yksinkertaisilla set-, put-, get-, delete- ja query-käskyillä. Entiteettiä voidaan käsitellä suoraan, jos entiteetin yksilöivä avain on sovelluksen tiedossa. Kaikki Datastoren tietokantakyselyt perustuvat tietokannan indekseihin. Jos entiteetin ominaisuutta ei ole indeksoitu, sitä ei voida hyödyntää kyselyissä. Indekseistä kerrotaan tar-

kemmin jatkossa. Kysely rajataan koskemaan aina tietyn lajisia entiteettejä. Kyselyn ehtoina voidaan käyttää entiteetin ominaisuuksien arvoja, esivanhempia tai avainta. Kyselylle voidaan määritellä ehtoja asettamalla *suodatin* (filter), jonka perusteella kysely rajaa kyselyn tuloksena palautettavat entiteetit. Suodattimeen määritellään esimerkiksi ominaisuuden nimi, vertailuoperaattori sekä ominaisuuden arvo. Jos kyselyä ei rajata lajin tai esivanhempien perusteella, kysely palauttaa kaikki sovelluksen entiteetit. Tälläiseen kyselyyn voidaan määritellä vain avaimeen perustuva suodatin. Kysely ei kuitenkaan palauta niitä rivejä, joiden suodattimeen määriteltyjä ominaisuuksien arvoja ei ole indeksoitu. Kyselyssä ei lisäksi voida käyttää liitos- tai koosteoperaatioita eikä suodattaa tietoja alikyselyn tuloksien perusteella. Samassa kyselyssä voidaan myös hyödyntää vain yhtä ominaisuutta kaikissa ehdoissa, joissa käytetään muita vertailuoperaattoreja kuin yhtäsuuruusvertailuoperaattoria.

Yleensä tietokantakyselyn tuloksena palautetaan entiteetin kaikki ominaisuudet. Kysely voi kuitenkin myös palauttaa vain tietyt entiteetin ominaisuudet, joista ollaan kulloinkin kiinnostuneita. Tämä Datastoren *projektiokysely* (projection query) on kysely, jossa määritellään ne entiteetin ominaisuudet, jotka tämän tyyppinen kysely palauttaa eli *projektoi* (project) [Goo12b]. Projektiokysely palauttaa tuloksenaan kyselyehtoa vastaavat entiteetit. Jokaisella palautetulla entiteettirivillä on vain niiden ominaisuuksien kohdalla arvo, jotka kyselyssä määriteltiin. Projektiokyselyihin liittyy kuitenkin rajoituksia. Samaa ominaisuutta voidaan esimerkiksi käyttää projektiossa vain kerran. Lisäksi vain niitä ominaisuuksia, jotka on indeksoitu voidaan projektoida. Jos taas projektoitava ominaisuus on esimerkiksi mukana kyselyn yhtäsuuruusehdossa, sitä ei voida projektoida. Seuraavan tyyppinen kysely ei esimerkiksi ole sallittu: SELECT ika FROM tyontekija WHERE ika = 30. Kyselyn tulokset voidaan myös järjestää ominaisuuden mukaan esimerkiksi nousevaan tai laskevaan järjestykseen. Kyselyssä määritellään ominaisuus, jonka perusteella kyselyn tulokset järjestetään. Tämä ominaisuus pitää kuitenkin olla indeksoitu. Jos ominaisuudella on useampia arvoja, ominaisuuden jokaista yksilöllistä arvoa kohdellaan kyselyssä yleensä omana rivinään. Projektiokyselyn tuloksena palautetaan esimerkiksi saman ominaisuuden kaikkiin kyselyn ehtoihin täsmäävät ominaisuus- ja arvokombinaatiot omana rivinään. Ominaisuus järjestetään myös sen mukaan, mikä sen yksittäinen arvo tulee ensimmäisenä vastaan järjestykseen perustuvassa tietokannan indeksissä. Kaikissa kyselyissä tämä ei kuitenkaan aina päde, vaan joissain tapauksissa kaikki ominaisuuden arvot otetaan huomioon. Tämä voi siis johtaa ennakoi-

mattomiin kyselytuloksiin.

Entiteettejä voidaan käsitellä myös eräajoina. Voidaan esimerkiksi tehdä ohjelma, jossa ensin määritellään useampi uusi entiteetti, jonka jälkeen kaikki määritellyt entiteetit lisätään tietokantaan yhdellä komennolla [Goo12b]. Eräajon käyttäminen säästää ohjelman kustannuksia ja resursseja, koska kuvatun kaltaisen eräajon käynnistäminen vaatii vain yhden tietokantakutsun sen sijaan, että jokainen entiteetti luodaan tietokantaan omalla kutsulla erikseen. Eräajo käsittelee lisättävät entiteetit myös tehokkaasti rinnakkain entiteettiryhmittäin. Kyselykielenä voidaan käyttää myös Googlen kehittämää SQL-kyselykielen kaltaista *Google kyselykieltä, GQL* (Google Query Language, GQL) [Goo12d]. GQL on Googlen kehittämä korkeamman tason kyselykieli. Kielen syntaksi muistuttaa jossain määrin SQL:ää. GQL on SQL:n tapaan helpompi ymmärtää ja omaksua, kuin jokin alemman tason ohjelmointikieli. GQL:n ominaisuudet kuitenkin poikkeavat SQL:stä. GQL:n avulla ei voida esimerkiksi suorittaa SQL:n liitos- ja koosteoperaatiota. App Engine Datastoressa GQL:n avulla voidaan ainoastaan kysellä entiteettejä ja niiden avaimia.

Sovellus voi hyödyntää App Engine Datastorea myös App Engine MapReduce -rajapinnan [Goo12e] kautta. App Engine MapReduce -ohjelmat voivat lukea tai kirjoittaa tietokantaan palvelun avoimeen lähdekoodiin perustuvien standardien luku- ja kirjoitusluokkien avulla. *Lukija* (writer) lukee tiedot esimerkiksi jostain GAE:n varastorakenteesta ja välittää tiedot laskennan syötetietoina MapReduce-kehyksen Map-funktiolle [Goo12g]. *Kirjoittaja* (writer) kirjoittaa MapReduce-kehyksen Reduce-funktion laskennan lopputuloksena tuottamat tiedot esimerkiksi johonkin GAE:n varastorakenteeseen. Lukija käy automaattisesti iteroiden läpi tietokantaan tallennettuja tietoja ja palauttaa joka iteraatiokierroksella aina tietyn ennalta määritellyn määrän tietoja. *DatastoreInputReader-lukija* esimerkiksi käy läpi ja palauttaa sovelluksen Datastoreen varastoitujen tietyn lajisten entiteettien instanssit. *DatastoreKeyInputReader-lukija* käy taas läpi ja palauttaa sovelluksen Datastoreen varastoitujen tietyn lajisten entiteettien avaimet. Standardille lukijalle voidaan siis määritellä palautettavien entiteettien laji. Muut mahdolliset tietokannan tietojen suodattimet joudutaan kuitenkin määrittelemään Map-funktiossa. Datastoreen kirjoittavasta standardista kirjoittajasta ei ole Googlen materiaalissa [Goo12g] kuvausta. Sovellusohjelmoija voi kuitenkin tarvittaessa tehdä myös oman muunnellun lukijan ja kirjoittajan.

Datastoren tietokantakyselyt voivat palauttaa tiedot kahdella *oikeellisuustasolla* (consistency level) [Goo12b]. *Vahvasti oikeelliset* (strongly consistent) kyselyt palauttavat aina ajan tasalla olevat tiedot, mutta voivat kestää pidempään. Samaa entiteettiryhmää koskevat esivanhempien perusteella suodatetut kyselyt ovat esimerkiksi vahvasti oikeellisia. *Lopulta oikeelliset* (eventually consistent) kyselyt taas palauttavat tiedot nopeammin, mutta voivat sisältää vanhoja tietoja. Kyselyt, joissa tietoja ei suodateta esivanhempien perusteella ovat esimerkiksi aina lopulta oikeellisia. Sovellusohjelmoija voi vaikuttaa sovellukselle näkyvien tietokannan tietojen oikeellisuuden tasoon määrittelemällä sovelluksen *tietojenlukupolitiikan* (read policy). Tietojenlukupolitiikassa voidaan määritellä eksplisiittisesti tietokantaoperaatioiden oikeellisuustaso, kuten esimerkiksi että kaikki lukuoperaatiot ja tietokantakyselyt ovat lopulta oikeellisia. Vaikka oikeellisuustaso määriteltäisiin eksplisiittisesti vahvasti oikeelliseksi, kyselyt joissa tietoja ei suodateta esivanhempien perustella ovat silti aina lopulta oikeellisia. Sovellusohjelmoija voi asettaa tietojenlukupolitiikan esimerkiksi Datastore-ohjelmointirajapinnan kautta Datastoren määrittelyyn tarkoitettujen Java-luokkien avulla.

Kuvassa 12 esitetään esimerkit Javalla toteutetusta Datastoren kirjoitusoperaatiosta ja tietokantakyselystä. Ensin lisätään juurientiteetiksi esimies Janne. Sen jälkeen lisätään kaksi työntekijää Anne ja Mari, jotka työskentelevät Jannen alaisuudessa. Lopuksi suoritetaan kysely, joka palauttaa kaikki Jannen alaisuudessa olevat työntekijät iän mukaan laskevaan järjestykseen järjestettynä. Näistä entiteeteistä palautetaan kaikki ominaisuudet.

```
import com.google.appengine.api.datastore.DatastoreService;
import com.google.appengine.api.datastore.DatastoreServiceFactory;
import com.google.appengine.api.datastore.Entity;
import com.google.appengine.api.datastore.Query;
import com.google.appengine.api.datastore.PreparedQuery;

DatastoreService datastore =
DatastoreServiceFactory.getDatastoreService();

//Lisää juurientiteetti
Entity esimies = new Entity("Esimies", "Janne");
esimies.setProperty("Nimi", "Janne");
esimies.setProperty("Ika", "47");
datastore.put(esimies);

//Lisää juuren alle entiteetti
Entity tyontekija1 = new Entity("Tyontekija", "Anne", esimies.getKey());
tyontekija1.setProperty("Nimi", "Anne");
tyontekija1.setProperty("Ika", "30");
datastore.put(tyontekija1);

//Lisää juuren alle entiteetti
Entity tyontekija2 = new Entity("Tyontekija", "Mari", esimies.getKey());
tyontekija2.setProperty("Nimi", "Mari");
tyontekija2.setProperty("Ika", "28");
datastore.put(tyontekija2);

//Kysellään juuriavaimen alla olevat työntekijät iän mukaan laskevaan
järjestykseen järjestettynä
Key juuriAvain = esimies.getKey();
Query Kysely = new Query("Tyontekija")
                         .setAncestor(juuriAvain);
                        .addSort("Ika", SortDirection.DESCENDING);
```

Kuva 12: Esimerkki Datastoren kirjoitusoperaatiosta ja tietokantakyselystä Javalla [Goo12b].

App Engine Datastore -tiedot varastoidaan *High Replication Datastore, HRD -varastoon* (High Replication Datastore, HRD) [Goo12b]. Kaikkien sovellusten tiedot tallennetaan tässä varastorakenteessa yhteen Bigtable-tauluun [CDG06, CDG08] [Goo12b]. Kaikkia entiteetin ominaisuuksia ei kuitenkaan tallenneta yksittäin taulun soluihin. Sen sijaan entiteetti sarjallistetaan vain yhteen soluun *Google-protokollapuskurin* (Google protocol buffer) [Goo12k] avulla. Google-protokollapuskuri on Googlen kehittämä tietojen sarjallistamiseen tarkoitettu formaatti. Google käyttää protokollapuskureita melkein kaikkiin tiedostomuotoihin ja etäproseduurikutsuihin. Entiteetin jokaisen ominaisuuden nimi ja arvot tallennetaan binäärimuodossa yhteen taulun soluun protokollapuskuriin. HRD-varasto perustuu Googlen aiemmin julkaisemaan *Megastore-varastojärjestelmään* (Megastore storage system) [BBC11, FuW11]. Google kehitti Megastoren Web 2.0 -sovellusten tarpeisiin [BBC11]. Relaatiotietokantojen heikko skaalautuvuus on liian rajoittava. Toisaalta NoSQL-tietokantajärjestelmien takaamat heikot tietokannan transaktioiden ACID-ominaisuudet sekä yksinkertainen ohjelmointirajapinta tekevät sovellusten kehittämisen hankalaksi. Megastore kuvataan multimallitietokantana, jonka tarkoituksena on yhdistää NoSQL-tietokantajärjestelmien hyvä

skaalautuvuus ja tehokkuus soveltuviin relaatiotietokantajärjestelmien ominaisuuksiin. Järjestelmässä sovellettavia relaatiotietokantajärjestelmien ominaisuuksia siis rajoitetaan tietyissä kohdissa. ACID-ominaisuudet taataan vain pienemmille mahdollisimman lähellä toisiaan sijaitseville tietokokonaisuuksille. Käytännössä tämä tarkoittaa esimerkiksi, että Megastore sirpaloidaan ja ositetaan hajautetun varastojärjestelmän eri pisteisiin. Tietokannan tiedot ositetaan entiteettiryhmittäin. Entiteettiryhmät toisinnetaan itsenäisinä kokonaisuuksina laajemmalle alueelle *synkronisesti* (synchronous). Synkroninen operaatio tarkoittaa, että prosessin pitää odottaa operaation valmistumista. Tietojen toisintamiseen hyödynnetään *Paxos-algoritmia* (Paxos) [CGR07, Lam98]. Paxos on yksimielisyys-algoritmi, jossa johonkin tehtävään osallistuvien prosessien välillä muodostetaan yksimielisyys hajautetussa epäluotettavassa ympäristössä [CGR07]. Jokainen osallistuja voi tuottaa jonkin tuloksen. Näistä tuloksista valitaan jokin, josta osallistujat ovat yhtä mieltä. Hajautetussa ympäristössä tämä on kuitenkin haasteellista, koska tehtävän osallistujat voivat sijaita eri solmuissa tai pisteissä. Kaikki pisteet eivät kuitenkaan ole välttämättä aina saatavilla. Paxos-algoritmia käytetään tämän ongelman ratkaisemiseen. Yksimielisyys voidaan saavuttaa vain, jos suurin osa tehtävään osallistujista on saatavilla jossain vaiheessa tarpeeksi pitkän aikaa ilman virheitä. Google esimerkiksi käyttää Paxosta tietokannan toisinteiden pitämiseen oikeellisina virhetilanteita varten. Algoritmin avulla voidaan saavuttaa yksimielisyys toisinnetun transaktioiden sitoutumislokin sisällöstä. Sitoutumislokia käsiteltiin tarkemmin luvussa 6. Tehtävään osallistujia ovat tietokannan toisinteet ja niiden tuottamat tulokset eli tietokannan päivitykset ovat sitoutumislokin rivejä. Tietokannan toisinteet pidetään oikeellisina tämän jokaisessa toisinteessa ylläpidettävän identtisen lokin avulla.

Megastoren sirpaloinnin tavoitteena on sijoittaa entiteettiryhmät sellaiseen pisteeseen, joka on mahdollisimman lähellä aluetta, josta entiteettiryhmään kohdistuu eniten operaatioita [BBC11]. Entiteettiryhmien toisinteet taas pyritään sijoittamaan mahdollisimman lähelle edellä mainittua pistettä. Tietokannan transaktioiden ACID-ominaisuudet voidaan taata vain tämän entiteettiryhmän sisällä. Tietojen oikeellisuutta sen sijaan heikennetään entiteettiryhmien välillä. Tämän takia kaikki transaktiot pyritään mahdollisuuksien mukaan kohdistamaan vain yhteen entiteettiryhmään. Entiteettiryhmien tiedot, metatiedot ja lokit tallennetaan kussakin pisteessä toimivaan Bigtable-tietokantaan [CDG06, CDG08] [BBC11]. Tietokantajärjestelmän toiminnassa, kuten esimerkiksi tietojen hajautuksessa voidaan siis hyödyntää Bigtablen tietomallia, infrastruktuuria ja

menetelmiä. Saman sovelluksen ja entiteettiryhmän tiedot myös esimerkiksi ryhmittyvät riviavaimen perusteella lähelle toisiaan Bigtable-tauluun. Näin toisiaan lähekkäin olevia tietoja voidaan hyödyntää tehokkaasti. Sovellusohjelmoija voi myös vaikuttaa tietokannan tietojen järkevään sirpalointiin ja osittamiseen määrittelemällä tiedot sopiviin entiteettiryhmiin.

Megastoressa on mahdollista määritellä myös tietokannan toissijaisia indeksejä mille tahansa entiteetin ominaisuudelle [BBC11]. Indeksit jaotellaan vain saman entiteettiryhmän tietoja sisältäviin *paikallisiin indekseihin* (local index) sekä *globaaleihin indekseihin* (global index), jotka sisältävät kaikkien entiteettiryhmien tietoja. Entiteettiryhmän tietoja sisältävä paikallinen indeksi varastoidaan entiteettiryhmän kanssa samaan pisteeseen. Paikallista indeksiä ylläpidetään ja päivitetään atomisesti ja oikeellisesti sitä mukaan, kun sen entiteettiryhmän entiteettien tietoja muutetaan. Globaalit indeksit sisältävät sen sijaan useiden entiteettiryhmien tietoja. Näiden indeksien avulla voidaan kysellä usean eri entiteettiryhmän tietoja. Vastaavasti globaalien indeksien tietokannan transaktioiden ACID-ominaisuuksia ei voidata taata, joten ne voivat sisältää vanhentuneita tietoja.

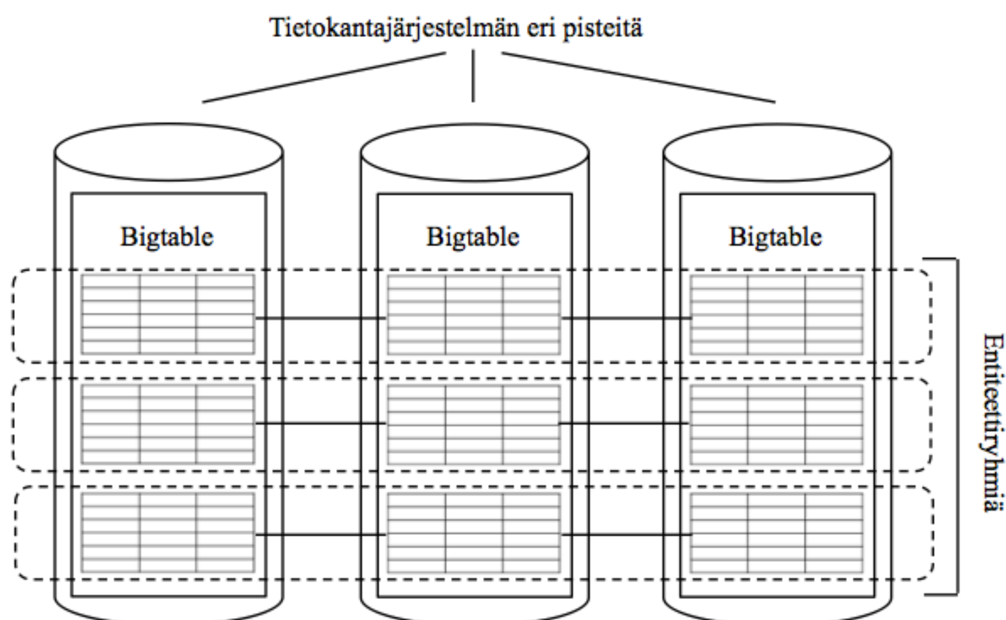
HRD perustuu Googlen mukaan Megastoreen [FuW11]. Koska HRD on yksi Megastoren toteutus erojakin on. Kaikkia eroavaisuuksia ei kuitenkaan ole selostettu tarkemmin Googlen dokumentaatiossa [Goo12b]. Indeksien toteutus esimerkiksi näyttää poikkeavan Megastore-indeksien toteutuksesta. HRD:ssä on myös mahdollista määritellä entiteettien ominaisuuksille tietokannan toissijaisia indeksejä. Kaikki varastoon kohdistuvat tietokantakyselyt perustuvat näihin ennalta määriteltyihin indekseihin. Sovelluksen indeksitietojen tallentamiseen käytetään viittä indeksitaulua, joissa ylläpidetään kaikkien App Engine Datastorea hyödyntävien sovellusten indeksejä. Googlen dokumentaatiosta [Goo12b] ei kuitenkaan selviä onko näitä indeksejä jaoteltu paikallisiin ja globaaleihin indekseihin. Kukin näistä tauluista varastoi eri tyyppisten indeksien tietoja. Entiteetit on järjestetty indeksitauluihin lajin, ominaisuuden ja arvojen mukaan. Entiteetti löytyy tietystä indeksistä vain, jos sen kaikkiin ominaisuuksiin joita käytetään kyseisessä indeksissä on asetettu arvo. Jos ominaisuudella on useampia arvoja, ominaisuudelle tulee jokaista sen yksilöllistä arvoa kohti oma rivi indeksitauluun. App Engine määrittelee oletuksena automaattisesti etukäteen jokaisen entiteettilajin jokaiselle ominaisuudelle indeksin. Sovellusohjelmoija voi kuitenkin määritellä ominaisuuden *indeksoimattomak-*

si (unindexed). Näin määriteltyä ominaisuutta ei indeksoida eikä sitä siten voi hyödyntää kyselyissä. Ominaisuuden määritteleminen indeksoimattomaksi kuitenkin säästää kirjoituskustannuksia, koska indeksoimatonta ominaisuutta ei tarvitse ylläpitää indeksitaulussa. Nämä automaattiset indeksit riittävät yleensä useimpien yksinkertaisten kyselyjen tarpeisiin. Kaikki muut indeksit on määriteltävä itse. Muita indeksejä tarvitaan esimerkiksi useampia ominaisuuksia ja arvoja hyödyntäviä kyselyjä varten. Sovellusohjelmoija määrittelee nämä muut indeksit sovelluskohtaisesti erillisessä indeksien määrittelyyn tarkoitetussa asetustiedostossa. Jos sovellus tekee tietokantakyselyn jonka tarvitsemaa indeksiä ei ole määritelty, syntyy virhetilanne joka pitää käsitellä sovelluslogiikassa. Edellä kuvattuja indeksejä ylläpidetään ja päivitetään *asynkronisen viestinnän* (asyncronic messaging) avulla sitä mukaan kun entiteettien tietoja muutetaan. Asynkronisessa viestinnässä operaatio ei jää odottamaan lähettämäänsä pyyntöön vastausta vaan voi jatkaa toimintaansa. Nämä järjestelmien tai prosessien väliset viestit tai pyynnöt jäävät jonoon odottamaan käsittelyä. Tämä tarkoittaa, että indeksit eivät ole aina oikeellisessa tilassa. Samaa entiteettiryhmää koskevat kyselyt eivät kuitenkaan palauta vanhentuneita tietoja [Goo12b]. Tästä kerrotaan vielä tarkemmin jatkossa.

Megastoren alkuperäisessä toteutuksessa kukin entiteettiryhmä on tavallaan pieni tietokantajärjestelmä, jolla on oma tietokannan transaktioiden sitoutumisloki [BBC11]. Tämä toisinnettu loki tallennetaan entiteettiryhmän juurielementin riville. Transaktiot ovat sarjallistuvia ja noudattavat ACID-ominaisuuksia tämän ryhmän sisällä. Megastore-transaktiot noudattavat *moniversioivaa samanaikaisuuden hallintaa, MVCC* (multiversion concurrency control, MVCC). MVCC on transaktioiden samanaikaisuudenhallintamenelmä, jonka avulla voidaan tehostaa tietokannan lukuoperaatioita [SKS11, s. 689-690]. MVCC:tä voidaan soveltaa tietokantajärjestelmässä, jonka tietokannan tietomalli tekee mahdolliseksi useamman eri monikkoversion tallentamisen. Tietokannan lukuoperaatio lukee esimerkiksi jollain tietyllä periaatteella viimeisimmän version tiedosta. Megastoressa MVCC hyödyntää moniulotteista Bigtable-tietomallia, joka sisältää tietokannan tiedoista aikaleimalla eroteltuja eri versioita [BBC11]. Aikaleimaa käytetään MVCC:n versiolaskurina. Lukuoperaatio käyttää hyväkseen viimeksi sitoutuneen transaktion aikaleimatietoa. Lukuoperaatio esimerkiksi varmistaa, että kaikkien keskeneräisten transaktioiden tiedot on kirjoitettu ensin levylle ja lukee vasta sen jälkeen viimeisimmän sitoutuneen version. Toinen vaihtoehto on, että lukuoperaatio lukee viimeisimmän sitoutuneen version joka on myös viety levylle, vaikka kaikkia muita sitoutuneiden

transaktioiden tietoja ei ole vielä viety levylle. Jos taas edellytetään parempaa suorituskykyä, lukuoperaatio ei välitä lokikirjauksista vaan lukee suoraan viimeisimmän version. Megastore tietokannan päivitykset levitetään tietokantajärjestelmän pisteisiin itsenäisesti ja synkronisesti saman entiteettiryhmän sisällä, mutta asynkronisesti eri entiteettiryhmien välillä. Saman entiteettiryhmän tiedot voivat sijaita toisinnettuina fyysisesti eri pisteissä. Eri entiteettiryhmien tietoja on taas myös samassa pisteessä. Tiedot toisinnetaan synkronisesti eri pisteiden välillä. Asynkronista viestinvälitystä käytetään siis loogisesti toisistaan erillä olevien entiteettiryhmien välillä. Megastoressa voidaan kuitenkin käyttää myös 2PC-sitoutumisprotokollaa päivitysten levittämiseen eri entiteettiryhmien välillä. 2PC:n käyttöön liittyvien kustannusten takia sitä ei kuitenkaan usein käytetä.

Kuvassa 13 esitetään Google Megastore -järjestelmäarkkitehtuuri. Kuvassa on kolme hajautetun tietokantajärjestelmän eri pistettä. Jokaisessa pisteessä on useampi toisistaan loogisesti erillinen entiteettiryhmä. Näiden entiteettiryhmien tiedot toisinnetaan synkronisesti muihin fyysisesti erillisiin pisteisiin. Loogisesti erillään olevien entiteettiryhmien välillä käytetään asynkronista viestintää myös saman pisteen sisällä. Kaikki entiteettiryhmät ja niiden sisältämät tiedot on varastoitu kussakin pisteessä toimivaan Bigtable-tietokantaan.



Kuva 13: Google Megastore -järjestelmäarkkitehtuuri [BBC11].

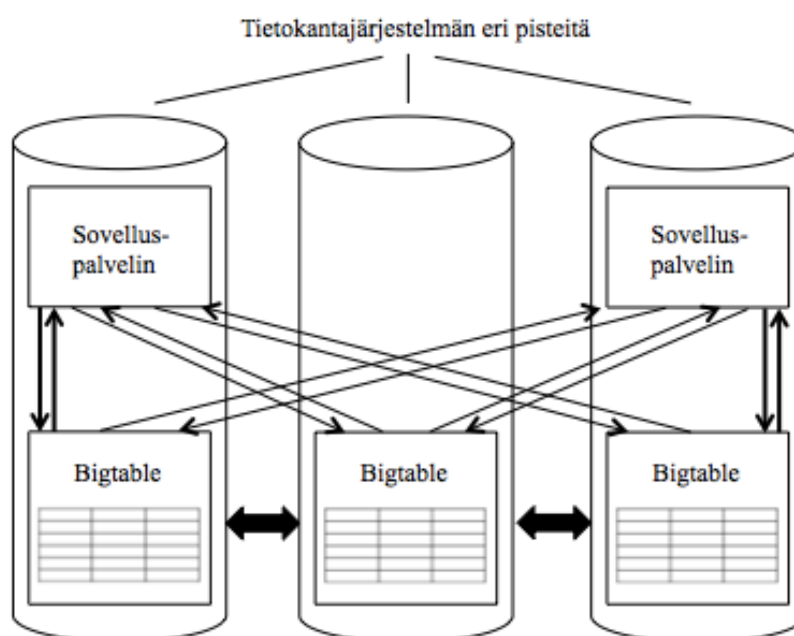
HRD-tietokannan transaktiot ovat myös sarjallistuvia saman entiteettiryhmän sisällä [Goo12b]. HRD:n transaktiot noudattavat *optimistista samanaikaisuuden hallintaa, OCC* (optimistic concurrency control, OCC). Optimistisessa samanaikaisuuden hallinnassa transaktion sallitaan tehdä rikkomuksia. Vasta transaktion sitoutumishetkellä tarkistetaan onko transaktio syyllistynyt rikkomuksiin [SKS11, s. 687-689]. HRD:n transaktion aloitushetkellä sekä juuri ennen transaktion sitoutumista katsottuja entiteettiryhmän aikaleimoja verrataan keskenään [Goo12b]. Transaktio keskeytetään jos päivitysaika ei ole sama. Jos samaa entiteettiä yrittää taas päivittää useampi sovelluksen instanssi, niin sen instanssin muutokset jonka transaktio sitoutuu ensin jäävät voimaan. Ne instanssit joiden transaktiot hylättiin, joutuvat yrittämään operaatioita uudestaan. HRD-kirjoitusoperaatio tehdään kahdessa eri vaiheessa. HDR tallentaa entiteettiä koskevan hyväksytyn kirjoitusoperaation tiedot ensin entiteettiryhmän toisinnettuun transaktioiden sitoutumislokiin, jonne varastoidaan riveittäin koko entiteettiryhmän entiteettien transaktioiden toistotiedot. Hyväksytty operaatio myös merkitään lokiin sitoutuneeksi. Sen jälkeen muutokset tallennetaan ja levitetään synkronisesti samaan aikanaan rinnakkain oikeiden palapalvelimien levyille tiedostoihin ja niiden toisinteisiin [FuW11, Goo12b]. Kaikki toisinteet eivät kuitenkaan saa päivityksiä välttämättä synkronisesti. Kaikki tietokantajärjestelmän pisteet eivät esimerkiksi ole välttämättä saatavilla. Päivitykset levitetään näihin toisinteisiin myöhemmin asynkronisesti, joten sovellus voi jatkaa toimintaansa välittömästi kun operaatio on sitoutunut. Ensin tallennetaan muutettu entiteetti. Tämän jälkeen HDR tarkistaa mitä kaikkia entiteettiä koskevia tietokannan indeksejä pitää päivittää. Jos sovelluksen käyttöön on esimerkiksi määritelty automaattisten indeksien lisäksi muita indeksejä, päivitettäviä indeksejä saattaa olla paljon. Kirjoitusoperaatioissa sattuu kuitenkin usein virheitä, kun taulun paloja jaetaan pienempiin osiin tai siirretään toisiin solmuihin. Jos entiteetin kirjoitus epäonnistuu sitoutumisvaiheessa, indeksejä ei päivitetä. Jos taas tietojen levitysvaihe epäonnistuu, muutokset voidaan levittää toisinteisiin myöhemmin seuraavan kerran, kun samaan entiteettiryhmään kohdistuu operaatioita. Tätä kutsutaan myös *tarvehakuiseksi päivitysten levittämiseksi* (demand-driven update propagation). Tarvehakuisessa päivitysten levityksessä päivitykset levitetään vasta siinä vaiheessa kun jokin prosessi tarvitsee niitä. Muutokset toteutetaan tietokannan toisinteisiin toisinnetun sitoutumislokin toistotietojen perusteella.

Koska muutokset levitetään myöhemmin asynkronisesti, kaikki tiedot eivät kuitenkaan välttämättä ole heti ajan tasalla kaikissa tietokantajärjestelmän pisteissä. Tietokanta-

kysely kohdistuu nopeimmin saatavilla olevaan tietokannan toisinteeseen, joka on yleensä paikallisessa pisteessä [FuW11]. Kyselyn tulokset voivat kuitenkin sisältää vanhentuneita tietoja, jos se koskee useampaa entiteettiryhmää [Goo12b]. Samaa entiteettiryhmää koskevat kyselyt palauttavat kuitenkin aina myös ajan tasalla olevat tiedot, koska kysely ei palauta tulosta ennen kuin kaikki saman entiteettiryhmän sitoutumislokiin merkityt sitoutuneet muutokset on levitetty johonkin saatavilla olevaan pisteeseen. Tämän odottaminen voi kuitenkin aiheuttaa viiveen. Saman entiteettiryhmän entiteettejä koskeville kirjoitusoperaatioille on jouduttu asettamaan vain yhden kirjoitusoperaation sekunnissa salliva kirjoitusraja. Kirjoitusraja tarvitaan, ettei kysely joudu odottamaan uusia päivitettyjä tietoja kohtuuttoman kauan. Google esittää, että oikeellisten tietojen levittämisestä aiheutuva viive ratkaistaan sovelluslogiikassa. Sovelluksen käyttäjälle voidaan näyttää viimeksi päivitetty tieto, esimerkiksi jonkin sovelluksen hallinnoiman välimuistitekniikan avulla. Sovellusohjelmoija voi siis vaikuttaa tietokannan transaktioiden suoritukseen ja tehokkuuteen sekä tietojen saatavuuteen ja oikeellisuuteen järjestämällä entiteettejä sopiviin entiteettiryhmiin. Sovellusohjelmoija voi myös määritellä tietyin rajoituksin näitä ryhmiä hyödyntävien lukuoperaatioiden ja kyselyjen oikeellisuustason. Sovellusohjelmoijan tulee siis päättää millaisten entiteettien tietoja on tarpeen käsitellä yhdessä eli mistä tiedoista koostuviin ryhmiin on tarpeen taata transaktioiden ACID-ominaisuudet ja mihin ei. Sovellusohjelmoijan tulee ottaa huomioon myös mahdollinen viive, joka aiheutuu ACID-ominaisuuksien noudattamisesta ryhmän sisällä. Jos tiedot jaotellaan semanttisesti useisiin pieniin ryhmiin, tarvitaan todennäköisesti paljon operaatioita, jotka koskevat useampaa ryhmää [BBC11]. Jos taas tiedot jaotellaan semanttisesti liian suurin ryhmiin jotka sisältävät paljon eri tyyppisiä tietoja, samaa ryhmää koskevien operaatioiden tehokkuus heikentyy.

Kuvassa 14 esitetään Google HRD:n järjestelmäarkkitehtuuri. Kuvassa on kolme hajautetun tietokantajärjestelmän eri pistettä. Jokaisessa pisteessä on Bigtable-tietokanta, jonne sovelluksen tiedot on varastoitu. Pisteessä voi toimia myös sovelluspalvelin, joka välittää sovelluksen luku- ja kirjoituspyynnöt tietokantaan. Nuolet kuvaavat näitä luku- ja kirjoitusoperaatioita. Kirjoitusoperaatio toteutetaan kahdessa vaiheessa. Muutokset tallennetaan ensin entiteettiryhmän sitoutumislokiin. Muutokset levitetään sen jälkeen synkronisesti tai asynkronisesti tietokannan toisinteisiin. Lukuoperaatio kohdistetaan nopeimmin saatavilla olevaan toisinteeseen, joka on yleensä paikallisessa pisteessä. Nopeimmin saatavilla olevassa pisteessä ei kuitenkaan välttämättä ole saatavilla kaikkia

ajan tasalla olevia tietoja. Samaa entiteettiryhmää koskevat tietokantakyselyt palauttavat kuitenkin aina myös ajan tasalla olevat tiedot, joten lukuoperaatio kohdistetaan sen sijaan hitaampaan pisteeseen, jonka tiedot ovat ajan tasalla. Tietoja luetaan niin kauan hitaammasta pisteestä, kunnes päivitetyt tiedot on levitetty myös nopeaan pisteeseen. Vasta tämän jälkeen tietoja aletaan lukea nopeammin saatavilla olevasta pisteestä.



Kuva 14: Google HRD:n järjestelmäarkkitehtuuri [FuW11].

7.5 Käyttömahdollisuuksien analysointi

Useimmat uudet NoSQL-tietokantajärjestelmät on kehitetty laajan mittakaavan Internet-sovellusten tarpeisiin. Näiden tarpeiden takia järjestelmän tulee olla skaalautuva, tehokas ja hyvin saatavilla. Kyseisten ominaisuuksien luonteen takia useimmat NoSQL-tietokantajärjestelmät, kuten erityisesti sarakeperhevarastot myös yleensä soveltuvat hyvin nimenomaan laajan mittakaavan Web 2.0 -sovellusten käyttöön. Laajan hajautetun tietokantajärjestelmän rakentaminen saattaa kuitenkin olla kallista ja haasteellista. Jos yritys päättää käyttää jotain NoSQL-tietokantaohjelmaa, sellaisen voi mahdollisesti ladata jopa ilmaiseksi Internetistä. Yrityksen web-sivusto voidaan sen jälkeen rakentaa hyödyntämään tällaista tietokantaa. Yrityksellä tulee kuitenkin olla käytettävissään myös tarvittava infrastruktuuri, jonka puitteissa tätä tietokantajärjestelmää käytetään. Sellaisen infrastruktuurin rakentaminen voi kuitenkin olla kallista ja haasteellista. Keskisuurilla tai isoillakaan yrityksillä ei usein ole omaa suurten Internet-yritysten kaltaista laajan mittakaavan skaalautuvaa infrastruktuuria. Tietokantajärjestelmän asentaminen,

määrittely ja ylläpito vaatii asiantuntijaosaamista. Varsinkaan aloittavalla yrityksellä ei välttämättä ole varaa ja muita resursseja hankkia, asentaa ja ylläpitää tällaista järjestelmää. Yrityksen web-sovellus ei lisäksi välttämättä ole heti valtavan käyttäjämäärän kiinnostuksen kohteena. Lisää resursseja saatetaan kuitenkin tarvita jatkossa, kun kävijämäärä kasvaa.

Aloittelevalle tai pienelle yritykselle voi olla houkutteleva vaihtoehto hyödyntää tietokantajärjestelmää NoSQL-tietokantapalveluna. Tällainen palvelu saattaa olla tiettyyn rajaan saakka jopa ilmaista. GAE-pilvipalvelua [Goo12b] esimerkiksi hyödynnetään yliopiston web-sovellusten ohjelmointikursseilla ohjelmointiympäristönä [HoP10]. Kurssilla opetellaan dynaamisten web-sivujen ohjelmointia *Java Server Pages, JSP* (Java Server Pages, JSP) sekä *Java Servletien* (Java Servlet) avulla. JSP ja Java Servlet ovat yleisiä tekniikoita joiden avulla voidaan tehdä Javalla dynaamisia web-sivuja. GAE:ssa on myös standardi Java Servlet rajapinta, jonka kautta voidaan hyödyntää myös JSP-tiedostoja. GAE:n ilmaiseen käyttöön määritellyt resurssikiintiöt riittävät hyvin opiskelijoiden Javalla tehtäviin pienimuotoisiin web-ohjelmoinnin harjoitustöihin. Kurssin aloittaminen on myös huomattavasti helpompaa ja nopeampaa, kun työssä vaadittu ympäristö on jo olemassa. Kurssin vetäjän esimerkiksi piti aikaisemmin ensin asentaa ja määritellä relaatiotietokantaympäristö, jota ohjelmoitava web-sovellus hyödyntää. Tällaisen harjoitustietokannan luomiseen sekä oppilaiden tilien ja käyttöoikeuksien määrittelyyn omalla palvelimella toimivaan MySQL- [Ora12e] tai DB2- [Ibm12a] tietokantajärjestelmään saattaa kulua paljon aikaa. Koulun ei lisäksi tarvitse huolehtia palvelun tai käyttöympäristön ylläpidosta. Laitteiden ja ohjelmistojen ylläpito kuuluu palvelun tuottajan vastuulle. Koulun palvelimella sijaitsevan tietokantapalvelimen käyttö saattaa myös olla hankalaa, jos työtä pitää tehdä koulun ulkopuolella. Sen sijaan GAE-hallintakonsoliin oppilaat pääsevät kirjautumaan mistä ja milloin tahansa Google Account -käyttäjätilinsä [Goo12a] välityksellä, joka usealla oppilaalla voi olla valmiina olemassa [HoP10]. Googlen laajan mittakaavan skaalautuva infrastruktuuri takaa tätä varten palvelun hyvän saatavuuden. GAE tarjoaa myös *Eclipse liitännäisen* (Eclipse plug-in). Eclipse on sovelluskehitysympäristö, jonka avulla voi kehittää ohjelmia useammalla eri ohjelmointikielellä, kuten Javalla. Eclipseä käytetään laajalti ohjelmoinnin opetukseen esimerkiksi tietojenkäsittelytieteen oppilaitoksissa. GAE Eclipse -liitännäisen avulla voi rakentaa ja testata GAE-sovelluksen Eclipsessä. Valmiin sovelluksen voi lopuksi ladata helposti suoraan Eclipse-ympäristöstä Googlen palveluun [Goo12b].

NoSQL-tietokantapalveluna näyttää siis soveltuvan hyvin ainakin pienimuotoiseen, väliaikaiseen ja ei-kriittiseen käyttöön. Jos yrityksen web-sovelluksen liikenne kasvaa, NoSQL-tietokantapalvelu myös skaalautuu tarpeen mukaan. Tarvittaessa yritys voi siirtyä palvelun maksulliseen käyttöön, jolloin lisäresursseista maksetaan tarpeen mukaan.

Yrityksen mahdollisesti jotain tietokantaa hyödyntävän sovelluksen siirtäminen pilvipalveluun ei kuitenkaan ole välttämättä kovin yksinkertaista. Olemassa olevan tiedon siirtäminen NoSQL-tietokantapalveluun saattaa olla hankalaa. Yrityksen sovellukseen voidaan joutua tekemään muutoksia. Sovellus ei esimerkiksi välttämättä noudata kaikkia palvelun vaatimuksia. Sovelluksen hyödyntämät tiedot saatetaan joutua myös esimerkiksi ensin muuntamaan sopivaan muotoon. Ison tietomäärän muuntaminen ja siirtäminen toiseen järjestelmään voi olla hidasta ja riskialtista. Tämä saattaa tarkoittaa käyttökatkoa olemassa olevan sovelluksen käyttöön. Pilvipalvelujen tietoturva ei ole välttämättä samalla tasolla yrityksen tietoturvavaatimusten kanssa. Yrityksen nykyisen järjestelmän käyttö voi olla hyvin rajattu ja suojattu, esimerkiksi palomuurilla sekä eri tasoisilla käyttöoikeuksilla. NoSQL-tietokantapalvelussa ei kuitenkaan välttämättä ole kaikkia vastaavia tietoturvaominaisuuksia, kuten esimerkiksi monipuolisia mahdollisuuksia käyttöoikeuksien rajoittamiseen tietokantatasolla. Tietoturvaa käsitellään lisää luvussa 8. Lainsäädäntö voi myös vaikuttaa tietojen sijoittamiseen. Pilvipalvelussa olevat tiedot saattavat olla paikallisen lainsäädännön vaikutuspiirissä. Tämä lainsäädäntö saattaa poiketa nykyisestä [Mur09]. Jos yritys siirtyy käyttämään jotain pilvipalvelua, se tulee yleensä samalla myös erittäin riippuvaiseksi pilvipalvelun toimittajasta. Yritys on siis jatkossa sidottu pitkäksi aikaa kaupalliseen järjestelmään. Tätä käsitellään lisää luvussa 8. Järjestelmän toimittaja saattaa myös tehdä muutoksia palveluun. GAE [Goo 12b] on esimerkiksi siirtynyt käyttämään HRD-varastoa aikaisemmin pääasiallisessa käytössä olleen erilaisen varastoratkaisun sijaan. Palvelun käyttäjät voivat edelleen käyttää myös vanhaa varastoa, mutta Google suosittelee vahvasti, että kaikki kyseisen varaston käyttäjät siirtyvät käyttämään uutta HRD-varastoa [Goo12f]. Sovellusten tietojen siirtämiseen uuteen järjestelmään tarjotaan kyllä työkalu, mutta työkalun ohjeessa mainitaan, että työkalun optimaalista käyttöä varten yritys voi joutua tekemään sovellukseen muutoksia. Pilvipalvelujen käyttöön liittyy myös monenlaisia rajoituksia. Nämä rajoitukset saattavat vaikuttaa siihen onko yrityksen mahdollista edes harkita jotain pilvipalvelua. GAE:ssa esimerkiksi voidaan käyttää vain tietyillä ohjelmointikielillä ohjelmoituja sovelluksia. Yritykselle voi siis soveltua paremmin joku yksityinen pilvi,

jossa yritys voi hyödyntää itse koko pilven infrastruktuuria vapaammin. NoSQL-tietokantapalvelua kuitenkin tarjotaan myös osana muunlaisia pilvilaskennan konseptimalleja. Amazon [Ama12a] esimerkiksi tarjoaa *Amazon EC -palveluaan* (Amazon Elastic Compute Cloud, Amazon EC) [Ama12b] IaaS-konseptimallin mukaan. Amazonin Amazon EC -pilvipalvelussa yritys voi asentaa oman sovelluksen suoraan Amazonin infrastruktuurin päälle. Amazon EC:ssä on mahdollista käyttää myös useampaa ohjelmointikieltä. Myös pilvipalvelujen hinnoittelu saattaa olla sekavaa. Hinnoittelu voi olla erittäin hienorakeista. Yrityksen saattaa olla vaikeaa hahmottaa mistä kaikesta palvelun kokonaishinta oikein muodostuu. Kyseisiä hinnoittelumalleja on lisäksi erilaisia, mikä tekee pilvipalvelujen vertailemisesta hankalaa. Google esimerkiksi tarjoaa ilmaisia resursseja vain tiettyyn rajaan saakka. Amazon taas tarjoaa vuoden ajan tietyn määrän ilmaisia resursseja [Ama12d]. Pilvipalvelut voivat tarjota lisäksi toisistaan poikkeavia *palvelutasosopimuksia, SLA* (service level agreement, SLA). SLA on yrityksen ja palveluntarjoajan välinen sopimus, jossa määritellään tarjottavan palvelun taso. Google esimerkiksi tarjoaa hyvitystä, jos sen palvelu ei toimi tai ole saatavilla sovitusti [Goo12c]. Amazon taas tarjoaa hienorakeisempaa SLA:ta, jossa on määritelty esimerkiksi *aluekohtainen saatavuus* (region unavailability) [Ama12c]. Se tarkoittaa palvelun saatavuutta sillä alueella, josta pilvipalvelussa ylläpidettävän sovelluksen instanssia ajetaan.

8 Vertailua ja analysointia

Laajan mittakaavan Web 2.0 -sovelluksia varten kehitetyt uudet tietokantajärjestelmät ja työkalut ovat siis haastaneet perinteiset relaatiotietokantajärjestelmät joillain osa-alueilla. NoSQL-tietokantajärjestelmillä ja työkaluilla on kuitenkin relaatiotietokantajärjestelmistä poikkeavia lähestymistapoja, ominaisuuksia ja käyttökohteita. Tässä luvussa tarkastellaan ensin MapReduce-ohjelmointiparadigman ja relaatiotietokantajärjestelmien toisistaan poikkeavia lähestymistapoja laajan mittakaavan analyyttiseen rinnakkaislaskentaan. Sen jälkeen vertaillaan NoSQL-tietokantajärjestelmiä relaatiotietokantajärjestelmiin. Lopuksi pohditaan vielä useamman erilaisen tietokantajärjestelmän ja työkalun hyödyntämistä yhdessä.

8.1 MapReduce-ohjelmointiparadigman ja relaatiotietokantajärjestelmien lähestymistavat laajan mittakaavan analyyttiseen rinnakkaislaskentaan

MapReduce-ohjelmointiparadigma [DeG08] on herättänyt viimeaikoina laajaa kiinnostusta kaupallisissa, akateemisissa ja tieteellisissä piireissä [AnT10, DeG10]. Paradigmaan perustuvaa ratkaisua hyödynnetään nykyään monessa laajan mittakaavan Web 2.0 -sovelluksessa [BGS11, Wei10]. Relaatiotietokantajärjestelmät kykenevät kuitenkin myös käsittelemään suuria tietokokonaisuuksia samanaikaisesti rinnakkain laajan mittakaavan hajautetussa ympäristössä. Väitetään jopa, että lähes mikä tahansa rinnakkaislaskentaa edellyttävä tehtävä voidaan kirjoittaa joko joukkona MapReduce-tehtäviä tai joukkona tietokantakyselyitä [PPR09]. Nämä relaatiomalliin perustuvat rinnakkaistietokantajärjestelmät ovat olleet käytössä jo ennen Googlen [DeG08] vuonna 2003 esittelemää MapReduce-kehyksen toteutusta [SAD10]. Näiden paradigmojen sekä niistä johdettujen toteutusten lähestymistavat laajan mittakaavaan analyyttiseen rinnakkaislaskentaan poikkeavat kuitenkin toisistaan.

MapReduce-ohjelmointiparadigma kehitettiin alun perin Google-hakukoneen [Goo12l] tarpeisiin. Tämän jälkeen Google on hyödyntänyt tätä paradigmaa esimerkiksi laajan mittakaavan verkkojen käsittelyyn, tekstinkäsittelyyn, koneoppimiseen sekä *staattiseen konekääntämiseen* (statistical machine translation) [DeG10]. Staattinen konekääntäminen tarkoittaa tietokoneella tehtävää kielenkääntämistä staattisten mallien avulla. MapReduce-ohjelmointiparadigman avulla voidaan analysoida laajoja tietokokonaisuuksia [OSG10] sekä hyödyntää tietokantakyselyjä [DeG10]. MapReduce-ohjelmointiparadigmaa voidaan hyödyntää näihin erilaisiin tarpeisiin myös avoimeen lähdekoodin perustuvien MapReduce-toteutusten avulla. Facebook on esimerkiksi rakentanut Apache Hadoopiin [The12b] perustuvan *Hive -tietovarastojärjestelmän* (Hive warehouse system) [The12h, TSJ09], jonne varastoituja tietoja voidaan hyödyntää MapReduce-ohjelmissa. Hive on isoille tietomäärille tarkoitettu avoimen lähdekoodin *tietovarastojärjestelmä* (data warehouse system), jonka tiedot varastoidaan Hadoopin kaltaiseen tiedostojärjestelmään.

MapReduce on koettu jossain määrin relaatiotietokantajärjestelmän tietojen analysointikyvyt haastavaksi paradigmaksi. Tiede- ja tietokantayhteisössä on kyseenalaistettu MapReduce-ohjelmointiparadigma sekä siitä johdetut toteutukset. De Witt ja kumppanit

[DeS08] katsovat MapReduce-ohjelmointiparadigman sopivan lähinnä yleisluontoiseen laskentaan, mutta olevan tietokantayhteisön näkökulmasta iso askel taaksepäin laajan mittakaavan tietointensiivisten sovellusten ohjelmointiparadigmana. Kirjoituksessa todetaan, että MapReduce edustaa melkein 25 vuotta sitten kehitettyjen hyvin tunnettujen tekniikoiden erästä toteutusta, josta puuttuu suurin osa relaatiotietokantajärjestelmien pitkälle kehitetyistä ominaisuuksista. MapReduce-ohjelmointiparadigman katsotaan olevan myös yhteensopimaton relaatiotietokantajärjestelmien työkalujen kanssa sekä käyttävän tietojen indeksoinnin sijasta raakaa voimaa.

De Witt ja kumppanien [DeS08] kritiikkiä on sen jälkeen arvosteltu. Vasta-argumenttien mukaan MapReduce-ohjelmointiparadigman uutena ideana on suorittaa laajan mittakaavan rinnakkaislaskentaa halvoista ja epäluotettavista tietokoneista koostuvissa suurissa tietokonerypäissä [CHU08, TYP08]. Arvostelussa tuodaan esille, että MapReduce ei ole tietokantajärjestelmä vaan algoritmitekniikka laajan mittakaavan tietojen käsittelyyn. MapReducea ei siis ole tarkoitettu korvaamaan tietokantajärjestelmiä. Samalla todetaan, että MapReduce ei ole varasto- tai hakujärjestelmä. Sen sijaan MapReducea käytetään laajan mittakaavan rakenteettoman tiedon käsittelyyn, josta tuotetaan jollakin tavalla jäsenneltyä tietoa, kuten esimerkiksi indeksejä. Tämän tiedon ei tarvitse olla relaatiomallin mukaista. Indekseistä ei lisäksi välttämättä ole edes hyötyä kaikissa isoissa laskentatehtävissä, joissa MapReducea nimenomaan usein käytetään.

Näiden molempien paradigmojen sekä niistä johdettujen toteutuksien ominaisuuksia ja tehokkuutta on silti vertailtu keskenään [DeG10, PPR09, SAD10]. MapReduce-ohjelmointiparadigman ja relaatiomallia noudattavien rinnakkaistietokantajärjestelmien sekä niihin liittyvien järjestelmien toteutuksien lähestymistapa laajan mittakaavan rinnakkaislaskentaan ovat kuitenkin erilaisia. Relaatiotietokantajärjestelmät edellyttävät tietojen vastaavan etukäteen selkeästi määriteltyä tietokannan skeemaa. MapReduce kuitenkin sallii myös täysin rakenteettoman tiedon. MapReduce-ohjelmointiparadigman ja relaatiotietokantajärjestelmien ohjelmointimallit ovat erilaisia. Näiden järjestelmien optimointi, tietojen hajautus sekä tehtävien laskenta poikkeavat lisäksi toisistaan. Useat näistä toisistaan poikkeavista piirteistä liittyvät kyseisistä paradigmoista johdettuihin toteutuksiin [SAD10].

MapReduce ei ole kyselykieli, tietokanta eikä tietokannan hallintajärjestelmä vaan ohjelmointimalli sekä siihen liittyvä kehyksen toteutus, joka on kehitetty suurten tietoko-

konaisuuksien analyyttiseen rinnakkaislaskentaan. Sovellusohjelmoija kirjoittaa Map- ja Reduce-funktiot, mutta MapReduce-kehyksen ajonaikainen järjestelmä hoitaa rinnakkaisen laskennan sekä siihen liittyvät yksityiskohdat hajautetussa ympäristössä [DeG08]. MapReduce-ohjelmointiparadigma on hyvin joustava ohjelmointimalli [DeG10, LLC12]. MapReduce-kehyksen funktiot voidaan kirjoittaa jollain yleisellä proseduraalisella ohjelmointikielellä. MapReducea ei ole sidottu mihinkään tietokannan tietomalliin eikä järjestelmään ladattavien tietojen välttämättä tarvitse noudattaa etukäteen määriteltyä skeemaa. Tämä ansiosta voidaan käsitellä harvaa, puolirakenteellista tai jopa täysin rakenteetonta tietoa. MapReduce-kehyksessä ei myöskään ole valmiita sisäänrakennettuja indeksejä. MapReduce-ohjelmointiparadigma on lisäksi täysin riippumaton tietojen varastointiin käytettävästä järjestelmästä. MapReduce-laskenta skaalautuu hyvin, koska laskentakapasiteettia voidaan lisätä tarpeen vaatiessa. MapReduce kykenee myös toipumaan virhetilanteista hyvin eli järjestelmällä on hyvä kestävyys. Edellä kuvatut ominaisuudet sopivat hyvin laajan mittakaavan Web 2.0 sovellusten vaatimuksiin. Näitä ominaisuuksia käsitellään tarkemmin seuraavaksi.

MapReduce-ohjelmointiparadigma on silti vain hyvin yksinkertainen ohjelmointimalli, jossa ei ole laskentaa helpottavia valmiita operaatioita, kuten esimerkiksi SQL-kyselykielen liitos- tai koosteoperaatiot [PPR09]. Näiden operaatioiden laskeminen MapReduce-ohjelmalla voi olla hankalaa ja tehotonta. Esimerkiksi koosteen laskeminen kahden tai useamman eri tietokokonaisuuden liitoksesta vaatii MapReduce-ohjelman suorituksen jakamista useaan toisistaan riippuvaan vaiheeseen, joiden tulokset putkitetaan aina seuraavan operaation syötteeksi. Useat MapReduce-tehtävät ovat hyvin yksinkertaisia ja ne voidaan ilmaista myös SQL-kyselykielellä [DeG10]. SQL:n ja sen liitos- ja koosteoperaatioiden ilmaisuvoima ei kuitenkaan välttämättä riitä kaikkiin monimutkaisiin analyyttisiin laskentatehtäviin. Näissä tehtävissä vaaditaan usein saman tiedon käymistä läpi useampaan kertaan. Silloin taas nimenomaan tarvitaan monimutkaisempi algoritmi, jossa ohjelman jonkin laskennan välituloksia voidaan myös putkittaa toisen prosessin syötetiedoiksi. Esimerkiksi jos joukosta dokumentteja etsitään tietyn kaltaiset yksilölliset merkkijonot, joiden kunkin kaikki esiintymät lasketaan sen jälkeen joukosta dokumentteja. MapReduce-ohjelmointiparadigmaa käytetään hyvin usein juuri tämän kaltaisiin laskentatehtäviin. Joissakin relaatiotietokantajärjestelmissä on silti myös mahdollista esimerkiksi hyödyntää *käyttäjien määrittelemiä funktioita, UDF* (user defined function, UDF) edellä kuvattujen laskentatehtävien ohjelmoimiseen. UDF on käyttäjän itse

määrittelemä toiminto, jolla voidaan laajentaa ohjelman toimintaa. UDF voi olla esimerkiksi SQL-kyselyyn liitettävä aliohjelma. UDF voidaan ohjelmoida esimerkiksi SQL-kyselykielellä tai joissain tapauksissa jollain alemman tason ohjelmointikielellä. Pavlo ja kumppanien [PPR09] artikkelissa tulee kuitenkin ilmi, että näiden monimutkaisten analyyttisten laskentatehtävien ohjelmoiminen relaatiotietokantajärjestelmissä voi olla hankalaa ja suoritus tehotonta.

Sovellusohjelmoija voi ohjelmoida MapReduce-ohjelman jollain yleisellä proseduraalisella ohjelmointikielellä. Google MapReduce -toteutuksessa voidaan esimerkiksi käyttää C++ -kieltä [PPR09]. Apache Hadoopissa [The12b] taas voidaan käyttää Javaa. SQL-kyselykielen etuna on sen deklaratiivisuus. Algoritmien kirjoittaminen alemman tason ohjelmointikielellä vaatii yleensä myös enemmän ohjelmakoodia. Toisaalta mahdollisuus käyttää jotain yleistä alemman tason ohjelmointikieltä tekee MapReduce-kehyksestä yleisluonteisemman ja joustavamman, jolla voidaan ohjelmoida esimerkiksi edellä kuvattuja monimutkaisempia analyyttisiä laskentatehtäviä. MapReduce-kehykseen on sittemmin kehitetty uusia rajapintoja korkeamman tason kieliä varten, jotta MapReduce-ohjelmien ohjelmointi olisi helpompaa ja vaatisi vähemmän ohjelmointikoodia. Google on esimerkiksi kehittänyt korkeamman tason proseduraalisen Sawzall-ohjelmointikielen tätä tarkoitusta varten. Edellä kuvattuja rajapintoja hyödyntäviä projekteja ovat myös *Pig* [ORS08, The12g], Hive [The12h, TSJ09], Scope [CJL08] ja Dryad/Linq [IBY07]. Pig on esimerkiksi alun perin Yahoo!n [Yah12b] kehittämä työkalu laajan mittakaavan rinnakkaiseen MapReduce-laskentaan [ORS08, The12g]. MapReduce-tehtävien ohjelmointiin voidaan työkalun avulla hyödyntää Apache Hadoop -ympäristössä [The12b] korkeamman tason proseduraalista *Pig Latin -ohjelmointikieltä* (Pig Latin programming language) [ORS08, The12g]. Pig Latin on SQL:n tapaan helpompi ymmärtää ja omaksua, kuin jokin alemman tason ohjelmointikieli. Pig Latin ei kuitenkaan ole SQL:n tapaan deklaratiivinen kyselykieli. Pig Latin -ohjelmia voidaan myös silti laajentaa UDF-funktioilla, jotka on tehty jollain alemman tason ohjelmointikielellä.

Mikäli skeemaa ei ole, MapReduce-ohjelman syötetiedot joudutaan jäsentämään vasta niiden käsittelyn yhteydessä [PPR09]. Syötetiedot voidaan ladata yksinkertaisimmillaan muotoilemattomana tekstinä suoraan syötetiedostosta järjestelmään. Tiedot kopioidaan MapReduce-kehystä hyödyntävän järjestelmän solmujen paikallisilta levyiltä samaan aikaan rinnakkain suoraan kuhunkin hajautetun tiedostojärjestelmän samassa solmussa

sijaitsevaan instanssiin. Tiedot tallennetaan tiedostojärjestelmään samassa tekstimuodossa kuin ne olivat syötetiedostossa. Tämän jälkeen tiedostojärjestelmä kopioi tiedostolohkon mahdolliset toisinteet muihin solmuihin. MapReduce-ohjelma joutuu silloin jäsentämään tiedot sekä muuntamaan ne oikeaan tietotyyppiin vasta varsinaisen laskentaoperaation aikana. Näin syötetietojen lataaminen MapReduce-kehystä hyödyntävään järjestelmään on tehokasta, mutta ladattujen tietojen jäsentäminen vasta laskennan yhteydessä voi vaikuttaa negatiivisesti varsinaisen laskentaoperaation suorituskykyyn.

Jos relaatiotietokannassa ei ole valmiina kaikkia laskentatehtävässä hyödynnettäviä tietoja, syötetietojen lataaminen relaatiotietokantaan saattaa vaatia useamman vaiheen [PPR09, SAD10]. Relaatiotietokantajärjestelmässä pitää olla ensinnäkin etukäteen määritelty tietokannan skeema ennen tietojen lataamista. Tiedot joudutaan jäsentelemään jo lataamisen yhteydessä skeeman mukaisesti oikein tietokannan tauluun. Tiedot ladataan yleensä jokaisessa solmussa rinnakkain suoritettavalla SQL-komennolla, jossa määritellään esimerkiksi miten syötetiedot erotellaan. Syötetiedot luetaan tällä komennolla rinnakkain solmujen paikallisilta levyiltä. Jokaisen syötetiedoston rivin erotinmerkillä erotetuista tiedoista muodostetaan monikko, joka sijoitetaan tietokannan tauluun. Tämän jälkeen tiedot joudutaan vielä järjestämään uudelleen jokaisessa solmussa. Tietojen uudelleen järjestäminen tehdään samaan aikaan rinnakkain kaikissa solmuissa joko automaattisesti tai manuaalisesti erillisellä komennolla riippuen käytettävästä järjestelmästä. Syötetietojen lataaminen relaatiotietokantaan on siis hitaampaa, mutta tietokantaoperaation aikana voidaan hyödyntää tehokkaasti suoraan valmiiksi tietokannan tauluihin jäsenneltyjä tietoja, kuten monikkojen attribuutteja.

Relaatiotietokantajärjestelmät hyödyntävät tietokantakyselyissä tietokannan skeemassa määriteltyjä tietokantaan muodostettuja indeksejä [PPR09]. Kyselynoptimoija esimerkiksi valitsee kyselylle indeksien perusteella kulloinkin edullisimman laskentastrategian. MapReduce-kehyksessä ei kuitenkaan usein ole valmiiksi määriteltyjä indeksejä. MapReduce-ohjelmissa voidaan kuitenkin hyödyntää *luonnollisia indeksejä* (natural indices) [DeG10]. Tällainen luonnollinen indeksitieto voi esimerkiksi olla lokitiedoston nimeen generoitu aikaleima. Tätä tietoa voidaan hyödyntää esimerkiksi luomalla MapReduce-ohjelma, joka hyödyntää vain niitä lokitiedostoja joiden aikaleima on tietyllä välillä. MapReduce on myös täysin riippumaton tietojen varastointiin käytettävästä järjestelmästä. Tämän ansiosta MapReduce-ohjelmalla voidaan hyödyntää erilaisia varas-

tojärjestelmiä, kuten hajautettuja tiedostojärjestelmiä, relaatiotietokantojen kyselyjen tuloksia tai NoSQL-tietokantoja. Luvussa 7 tuli esille miten MapReduce-ohjelma voi esimerkiksi hyödyntää NoSQL-tietokantaa. MapReduce-ohjelma voi myös käyttää syötetietoinaan esimerkiksi jo valmiiksi suodatettuja tietoja, kuten indeksejä hyödyntävien tietokantakyselyjen tuloksia. Jos tiedot on ryhmitelty tietokantaan rivien tai sarakkeiden mukaan, voidaan lukea vain ne tietokantarivialueet tai sarakkeet joista ollaan kulloinkin kiinnostuneita.

Jossakin yhteisessä tietohakemistossa sijaitsevan skeeman ja indeksitietojen puuttuminen voi kuitenkin olla myös ongelmallista, jos useamman sovelluksen pitää hyödyntää samaa käsiteltävää tietoa. Sovellusohjelmoijan määrittelemää tiedon rakennetta pitää voida tulkita jollain tavalla. Mahdolliset indeksit sekä niiden käyttötapa pitää olla myös kaikkien tietoa hyödyntävien tahojen tiedossa. Jos taas tiedolle on määritelty jotain rajoitteita, niitä pitää noudattaa. Jos samaa MapReduce-ohjelmaa hyödyntää useampi sovellus, edellä kuvatun toiminnallisuuden toteuttaminen jää yleensä sovellusohjelmoijan vastuulle. MapReduce-ohjelma voi kuitenkin hyödyntää myös jotain rakenteellista itsekuvautuvaa formaattia, kuten JSON, Google Protocol Buffer tai XML [LLC12].

MapReduce-laskentaa voidaan suorittaa laajan mittakaavan tietokonerypäässä, jonka laskentakapasiteettia voidaan lisätä tarpeen mukaan. MapReduce-järjestelmän yksittäisiä Map- ja Reduce -laskentasolmuja lisätään järjestelmään laskentaoperaatioiden määrän kasvaessa. Tämän ansiosta MapReduce skaalautuu hyvin. Yahoo! [Yah12b] on esimerkiksi ilmoittanut vuonna 2008, että sen Hadoopiin [The12b] perustuvan MapReduce-toteutuksen [Ana08] laskentakapasiteetti skaalautuu jopa yli 4000 yksittäiseen solmuun [Ana08, LLC12].

Laajan mittakaavan hajautettu laskenta edellyttää kuitenkin järjestelmältä myös hyvää kestävyyttä. Erilaiset viat ovat tämän kaltaisissa ympäristöissä yleensä hyvin todennäköisiä. MapReduce käyttää virhetilanteista toipumiseen uudelleen käynnistystä. Tätä käsiteltiin luvussa 6. Virheen takia hylättyjä laskentavaiheita suoritetaan uudestaan. MapReduce tallentaa laskennan välituloksia levylle. Virheistä toipumiseen voidaan hyödyntää myös näitä levylle tallennettuja tietoja. MapReduce-ohjelman Map-instanssit tallentavat käsitellyt tulokset solmun paikalliselle levylle. Tämän jälkeen Reduce-työläinen lukee eli vetää Map-työläisen prosessoimat tiedot etäproseduurikutsuilla Map-solmun paikalliselta levyltä käyttäen jotain tiedonsiirtoprotokollaa. Laskennan lopuksi

Reduce-vaiheen tulokset tallennetaan hajautettuun tiedostojärjestelmään. Näitä valmiita tuloksia voidaan hyödyntää, kun tehtävää jatketaan mahdollisen virheen jälkeen. Tämä menettelytapa voi kuitenkin hidastaa ohjelman suoritusta [PPR09]. Jos välitulokset ovat suurikokoisia, niiden tallentaminen solmun paikalliselle levylle voi olla kallista. Jos taas useampi Reduce-instanssi yrittää vetää tietoja samasta solmusta, se voi aiheuttaa kilpailua solmun resursseista.

Relaatiomallia noudattavien rinnakkaistietokantajärjestelmien tietokannan transaktiot pyrkivät sen sijaan välttämään laskennan välituloksien tallentamista levylle, joten vikatilanteessa yleensä koko transaktio joudutaan suorittamaan uudestaan [SAD10, PPR09]. Relaatiotietokantajärjestelmän kyselynoptimoija resursoi ohjelman suorituksen aluksi operaation yksittäiset tehtävät jokaiseen solmuun tehtäväjonoon. Kun solmun tehtävänä on lähettää tietoa toiseen solmuun, tietoa ei tallenneta käsittelevän solmun paikalliselle levylle vaan se työnnetään vastaanottavaan solmuun. Laskennan välituloksia ei siis tallenneta levylle. Kysely voidaan suorittaa tehokkaasti, koska välituloksia ei tallenneta levylle eikä kilpailua resursseista välttämättä siten synny. Toisaalta tämä menettelytapa heikentää relaatiotietokantajärjestelmän kestävyyttä. Laskennan välituloksia ei voida hyödyntää virheistä toipumiseen, koska niitä ei tallenneta levylle. Relaatiotietokantajärjestelmissä saattaa siis olla tarvetta transaktiotasoa hienorakeisempaan järjestelmän kestävyyteen, kun hajautetun järjestelmän mittakaava kasvaa ja mahdolliset virheet sen myötä todennäköisesti yleistyvät.

Relaatiotietokantajärjestelmän kyselynoptimoija laskee operaatioiden suoritukselle tehokkaimman laskentastrategian. Koska suoritettavat tehtävät ovat solmujen tiedossa tehtäväjonossa etukäteen, järjestelmän kyselynoptimoija pystyy tasaamaan hajautetun järjestelmän solmujen laskentakuormaa tehokkaasti niin, että solmujen välinen verkkoliikenne pysyy mahdollisimman vähäisenä [SAD10, PPR09]. Koostekyselyssä tarvittavien tietojen esisuodatus esimerkiksi voidaan tehdä paikallisesti jo niissä solmuissa, joissa tiedot sijaitsevat. Näin vain tarvittavat valmiiksi suodatetut tiedot lähetetään verkon yli kyselyn lopullisen koosteen laskentaa varten. MapReduce-isäntä taas resursoi kullekin solmulle suoritettavat tehtävät vasta MapReduce-ohjelman suorituksen aikana. Yksinkertaisen MapReduce-ohjelman toteutuksessa esimerkiksi vastaava tietojen suodatus saatetaan tehdä vasta Reduce-vaiheessa, jolloin Reduce-työläiset ovat jo lukeneet Map-työläisten käsittelemät tiedot. Näin verkon solmujen välillä voi liikkua tietoja, joita

ei enää välttämättä tarvita. Tehtävien suorituksen ajastamisesta vasta ohjelman suorituksen aikana voi kuitenkin olla hyötyä, jos yksittäisten solmun suorituskyvyssä tai laskentakuormassa havaitaan poikkeavuuksia. Tämä mahdollistaa solmujen tosiaikaisen kuorman tasauksen sekä reagoimisen hitaasti toimiviin solmuihin.

Hajautetun järjestelmän tehokkuuteen vaikuttaa myös yksittäisten laskentasolmujen tehokkuus. MapReduce-ohjelman käynnistämiseen useammassa solmussa saattaa aiheuttaa käynnistyskustannuksia, jotka hidastavat ohjelman suoritusta [PPR09]. Käynnistyskustannus liittyy jokaiseen yksittäisessä solmussa käynnistettävään työläisprosessiin. Käynnistyskustannukset saattavat rajoittaa erityisesti useista lyhytkestoisista laskentatehtävistä koostuvan ohjelman suoritusta, koska jokaisen pienen osatehtävän käynnistämiseen liittyy oma käynnistyskustannus. Tämän tyyppisten laskentatehtävien suoritukseen kuluu myös enemmän aikaa, koska MapReduce-isäntä joutuu käynnistämään ja koordinoimaan enemmän Map-instansseja useammassa solmussa. MapReduce-toteutuksen käynnistyskustannuksia on kuitenkin mahdollista vähentää. Työläisinstansseja voidaan esimerkiksi pitää käynnissä jonkin MapReduce-ohjelmaan kuuluvan päättyneen laskentaprosessin jälkeen, jolloin käynnistyskustannuksia ei enää synny kun MapReduce-isäntä resursoi työläiselle seuraavan tehtävän [DeG10].

MapReduce-kehystä hyödyntävien järjestelmien ja relaatiotietokantajärjestelmien ohjelmointirajapintojen ominaisuudet kuitenkin lähentyvät toisiaan. MapReduce-kehyksen toteutuksiin on kehitetty uusia rajapintoja korkeamman tason ohjelmointikielille [PPR09, SAD10]. Vastaavasti relaatiotietokantajärjestelmiin kannattaa kehittää ominaisuuksia, joiden avulla on tulevaisuudessa helpompi toteuttaa monimutkaisiin analyyttisiin laskentatehtäviin tarvittavia funktioita. Molempien järjestelmien ominaisuuksia yhdistäviä järjestelmiä on jo olemassa. MapReduce-ohjelmointiparadigman ja SQL:n ominaisuuksia yhdistäviä multimallijärjestelmiä ovat esimerkiksi *HadoopDB* [ABK09], Hive [The12h, TSJ09], Asterdata, Greenplum, Cloudera ja *Vertica* [Ver12]. Vertica on alunperin Michael Stonebrakerin ja Andrew Palmerin kehittämä sarakeperustainen relaatiotietokantajärjestelmä, jossa on Hadoop- [The12b] ja Pig- [ORS08, The12g] integraatio. MapReduce-ohjelmissa voidaan hyödyntää Verticaan tallennettuja tietoja näiden rajapintojen kautta. HadoopDB on taas Apache Hadoop -ympäristössä [The12b] toimiva hajautettu avoimen lähdekoodin multimallijärjestelmä, joka koostuu useammasta yhden solmun relaatiotietokantajärjestelmästä [ABK09, LLC12]. Tässä hajautetussa tietokan-

tajärjestelmässä pyritään hyödyntämään relaatiotietokantajärjestelmien SQL-kyselyjen suorituskykyä sekä MapReduce-kehyksen laskennan skaalautuvuutta ja kestävyyttä. Suurin osa kyselynkäsittelystä pyritään siis suorittamaan yksittäisten solmujen relaatiotietokantajärjestelmissä. SQL-kysely hajautetaan MapReduce-kehyksen avulla useammaksi tehtäväksi, jotka suoritetaan rinnakkain näissä yksittäisissä solmuissa. Kukin solmu suorittaa sille resursoidun osan SQL-kyselystä ja palauttaa tuloksen avain-arvopareina MapReduce-kehykselle mahdollista jatkokäsittelyä varten.

MapReduce-kehystä hyödyntävät järjestelmät voivat myös olla houkutteleva vaihtoehto käyttäjille, joilla on vähäiset vaatimukset järjestelmälle sekä pieni hankintabudjetti. Useat MapReduce-kehystä hyödyntävät järjestelmät perustuvat avoimeen lähdekoodiin, joka on saatavilla ilmaiseksi [PPR09, SAD10]. Relaatiomalliin perustuvat rinnakkaistietokantajärjestelmät taas ovat usein kalliita ja järeitä kaupallisia järjestelmiä. MapReduce-kehystä hyödyntävän järjestelmän asentaminen nopeasti toimintakuntoon esimerkiksi kertaluontoisia laskentatehtäviä varten saattaa olla myös yksinkertaisempaa kuin ison relaatiotietokantajärjestelmän. Asennuksen yhteydessä tarvitsee vähintään määritellä solmujen tietohakemistot, järjestelmän kirjasto sekä kokoonpanoasetukset. Tämän jälkeen MapReduce-ympäristö on valmis ohjelman suoritukseen. Relaatiotietokantajärjestelmän asentaminen ja määrittely käyttövalmiuteen oikein voi olla paljon monimutkaisempaa. Kaikki oletusasetukset eivät välttämättä toimi ja niiden muuttaminen voi olla hankalaa. Optimaalisten asetusten löytämiseen saatetaan tarvita myös järjestelmäntoimittajan apua.

Stonebraker ja kumppanit [SAD10] esittävät artikkelissaan, että MapReduce-ohjelmointiparadigma soveltuu parhaiten yksinkertaisiin *uuta-muunna-lataa, ETL-tehtäviin* (extract-transform-load, ETL) sekä monimutkaisiin analyyttisiin tehtäviin. ETL-tehtävässä tiedot poimitaan jostain syötetiedostosta, muunnetaan sopivaan muotoon sekä ladataan johonkin varastojärjestelmään. Tämän mukaan MapReduce lähinnä muuntaa raakatietoa jonkin varastojärjestelmän kulutettavaksi. MapReduce-kehys voisi sen takia toimia myös relaatiotietokantajärjestelmän yhteydessä yleiskäyttöisenä rinnakkaislaskentaan tarkoitettuna ETL-työkaluna.

MapReduce ja relaatiotietokantajärjestelmät näyttävät siis täydentävän toisiaan. Stonebraker ja kumppanit [SAD10] tulevat lopulta siihen johtopäätökseen, että MapReduce ja relaatiotietokantajärjestelmät eivät ole kumpikaan hyviä siinä mitä toinen osaa hyvin.

Tämän katsotaan johtuvan näiden järjestelmien keskittymisestä hajautetun ja rinnakkaisen laskennan eri tehtäviin, kuten myös edellä tuli ilmi. MapReduce-laskenta sopii esimerkiksi ETL-tehtäviin sekä monimutkaiseen analyyttiseen laskentaan. MapReduce myös skaalautuu kestävällä tavalla laajan mittakaavan ympäristössä. Relaatiotietokantajärjestelmien pitkälle kehitetyt kyselyominaisuudet ovat taas esimerkiksi intuitiivisia ja hyvin tehokkaita. Molempien järjestelmien vahvuuksia tarvitaan monimutkaisten analyyttisten ongelmien ratkaisemiseen. Silloin looginen ratkaisu olisi antaa kummankin järjestelmän hoitaa suosiolla ne tehtävät joissa on toista parempi eikä yritetä kehittää vain yhtä yksi koko sopii kaikille -järjestelmää, jonka odotetaan hoitavan kaikki nämä tehtävät hyvin.

8.2 NoSQL-tietokantajärjestelmien ja relaatiotietokantajärjestelmien vertailu

NoSQL-tietokantajärjestelmät ovat vielä aika uusia järjestelmiä. Web 2.0 on suhteellisen tuore kehityssuunta. Sitä varten kehitetyt ensimmäiset NoSQL-tietokantajärjestelmät esiteltiin vasta 2000-luvun alussa. Näitä järjestelmiä on sittemmin kehitetty ja julkaistu vasta viime vuosina. NoSQL-paradigmaan kohdistuu tällä hetkellä paljon huomiota [Cat10, HEL11]. Paradigma herättää keskustelua ja sitä tutkitaan useilla tahoilla. Uusia NoSQL-järjestelmiä kehitetään kiihtyvällä vauhdilla. Aikaisemmin julkaistuihin järjestelmiin julkaistaan myös päivityksiä ja lisäosia. NoSQL-paradigmaan perustuvien järjestelmien ohjelmakoodi ei kuitenkaan välttämättä vielä ole riittävän kehittyneellä tasolla [Cat10]. NoSQL-tietokantajärjestelmien ohjelmakoodia ei ole ehkä ehditty testata tarpeeksi hyvin, joten ne eivät ole välttämättä vielä kovin luotettavia. Liian aikaisessa vaiheessa uuteen tekniikkaan siirtyvällä web-sivustolla saattaa esimerkiksi esiintyä uudesta tekniikasta johtuvia käyttökatkoksia. Perinteisillä relaatiotietokantajärjestelmillä on sen sijaan takanaan jo pitkä kehityshistoria. Pitkä kehityshistoria tekee järjestelmästä todennäköisesti myös paljon luotettavamman koska sitä on ehditty testata, käyttää ja kehittää pidemmän aikaa.

NoSQL-tietokantajärjestelmissä ei ole useita perinteisten relaatiotietokantajärjestelmien valmiita pitkälle kehitettyjä ominaisuuksia [HEL11, RYS11]. NoSQL-tietokantajärjestelmien käyttöä perusteellaan usein sillä, ettei kaikkia raskaiden relaatiotietokantajärjestelmien ominaisuuksia edes tarvita [Lea10]. Tietokannan skeemaa ei välttämättä ole ja tietojen indeksointitoiminnot ovat rajoittuneita, esimerkiksi tietokannan toissijaisia in-

deksejä ei usein voida käyttää. Toissijaisille indekseille on silti ollut tarvetta. Tietokannan yhteisen skeeman ja indeksitietojen puuttuminen voi vaikuttaa suorituskykyyn ja johtaa ongelmiin useamman sovelluksen ympäristössä, kuten edellä tulee ilmi. Chang ja kumppanien [CDG06, CDG08] artikkelissa esimerkiksi todetaan, että Bigtableen ollaan kehittämässä mahdollisuutta käyttää toissijaisia indeksejä. Googlen uudemmissa NoSQL-tietokantajärjestelmissä [BBC11, Goo12b] on mahdollisuus käyttää toissijaisia indeksejä. GAE [Goo12b] esimerkiksi edellyttää, että kaikki tietokantakyselyt perustuvat ennalta määriteltyihin indekseihin.

Laajan mittakaavan Web 2.0 sovellusten tieto on usein harvaa ja semirakenteellista tai jopa kokonaan rakenteetonta. Jokin NoSQL-tietokantajärjestelmien erikoistunut tietokannan tietomalli saattaa siis sopia hyvin tämän kaltaisten tietojen tallentamiseen ja käsittelyyn. Esimerkiksi sarakeperhevarastojen tietomalli on harva. Rivillä voi olla vaihteleva määrä sarakkeita. Ainoastaan kunkin rivin käytössä olevat sarakkeet tallennetaan fyysisesti. Tavalliseen relaatiotietokantaan voidaan myös tallentaa harvaa tietoa, joka sisältää yhtä riviä kohden vaihtelevan määrän attribuutteja [Aba07, SAD10]. Tässä tapauksessa jokaisen puuttuvan attribuutin arvoksi tulee NULL-arvo. NULL-arvo joudutaan usein tallentamaan fyysisesti levylle, joten operaatioiden suoritus hidastuu ja tietokannan koko kasvaa. Toisaalta on olemassa myös uusia relaatiotietokantajärjestelmiä, kuten esimerkiksi Vertica [Ver12], jotka sallivat tällaisen harvan tiedon tehokkaan käsittelemisen. Vertica esimerkiksi käsittelee vain operaatiossa kulloinkin tarvittavia tietoja sekä ohittaa kaikki NULL-arvot automaattisesti [Aba07, SAD10].

NoSQL-tietokantajärjestelmien tietokannan taululiitosoperaatiot ovat rajoittuneita verrattuna relaatiotietokantajärjestelmiin. Tiettyjen monimutkaisempien kyselyoperaatioiden laskenta on tehokasta relaatiotietokantajärjestelmissä niiden valmiiden ja optimoitujen laskentaa yksinkertaistavien liitos- ja koosteoperaatioiden ansiosta. NoSQL-tietokantajärjestelmien tietokantakyselyissä ei sen sijaan usein voi toteuttaa liitos- tai koosteoperaatioita. Liitosoperaation laskenta pitää yleensä toteuttaa jotenkin sovelluslogiikassa [Pok11]. Taulujen liitosoperaatioiden toteuttaminen on hankalaa laajan mittakaavan ympäristössä, jossa tiedot ositetaan vaakasuoraan. Tämä koskee myös relaatiotietokantajärjestelmiä. Edellä mainittujen SQL-operaatioiden suoritus pitää siis myös hajauttaa. Useita eri solmuja koskevien tietokannan transaktioiden koordinointi on kuitenkin kallista. Tietojen siirrosta eri solmujen välillä syntyy kustannuksia.

Väitetään, että NoSQL-tietokantajärjestelmät ovat tehokkaampia ja skaalautuvat paremmin kuin relaatiotietokantajärjestelmät laajan mittakaavan ympäristössä. Relaatiotietokantajärjestelmien saatavuus, skaalautuvuus ja suorituskyky on koettu riittämättömäksi. Rick Cattellin [Cat10] mielestä relaatiotietokantajärjestelmien ei ole vielä osoitettu saavuttavan yhtä tehokasta skaalautuvuutta, kuin esimerkiksi Google Bigtable. NoSQL-tietokantajärjestelmä taas väitetään skaalautuvan tietojen osittamistavan ansiosta lähes lineaarisesti verrattuna solmujen määrään [Pok11]. NoSQL-tietokantajärjestelmät hyödyntävät yksityislevyarkkitehtuuria sekä löyhentävät tai jättävät kokonaan pois relaatiotietokantajärjestelmien ominaisuuksia. Nämä ovat syitä joiden ansiosta ne saavuttavat hyvän saatavuuden, skaalautuvuuden ja suorituskyvyn. Edellä mainitut relaatiotietokantajärjestelmien ominaisuudet kuluttavat hajautetun tietokantajärjestelmän resursseja ja vaikuttavat negatiivisesti saatavuuteen, skaalautuvuuteen ja suorituskykyyn. Stonebraker ja kumppanit [StC11] toteavat kuitenkin, että kaikki vakavasti otettavat viimeisen 10 vuoden aikana kehitetyt relaatiotietokantajärjestelmät pystyvät skaalautumaan hyödyntämällä yksityislevyarkkitehtuuria. Relaatiotietokantajärjestelmien suorituskyky riippuu myös näiden järjestelmien toteutuksesta [Cat10, StC11]. Viimeaikoina on kehitetty tehokkaammin skaalautuvia uusia relaatiotietokannan hallintajärjestelmiä, kuten VoltDB ja Clustrix [Cat10]. Lisäksi olemassa olevien järjestelmien, kuten MySQL Clusterin [Ora12d] suorituskykyä on parannettu. Relaatiotietokantajärjestelmät saattavat päästä parempaan skaalautuvuuten, jos tietokantaan kohdistuvat operaatiot ovat pieniä ja suppeita [StC11]. Suppeammat operaatiot kohdistuvat todennäköisesti myös pienempään määrään solmuja, joten tietojen siirrosta eri solmujen välillä ei aiheudu yhtä paljon kustannuksia kuin laajemmissa operaatioissa. Tietokantajärjestelmän ei todennäköisesti lisäksi tarvitse kuluttaa yhtä paljon resursseja tietokannan transaktioiden 2PC-sitoutumiskäytännön koordinoimiseen, kun koordinoitavia osapuolia on vähemmän. Tietokantajärjestelmä tulee tätä varten sirpaloida järkevästi. Tietokantajärjestelmää hyödyntävät sovellukset tulee taas suunnitella niin, että ne voivat hyödyntää näitä tietokantajärjestelmän sirpaleita mahdollisimman tehokkaasti. Tietokantaoperaatioiden tulee siis koskea mahdollisimman pientä solmu ja palamäärää. Paljon levyä käyttävien tietokantaoperaatioiden suorituskykyä voidaan taas parantaa tehokkaammalla tallennustekniikalla. Tämä koskee myös yhteislevyarkkitehtuuria. Yhteiset levyt voivat olla järjestelmän tehokkuutta rajoittava tekijä. Levyn käyttöä voidaan kuitenkin tehostaa hyödyntämällä perinteisiä kiintolevyjä tehokkaampaa *SSD-massamuistia* (solid state drive) [Pok11].

SSD on tiedontallennusväline, jossa ei ole pyöriviä tai muuten liikkuvia mekaanisia osia toisin kuin perinteisessä kiintolevyssä. SSD-tallennusvälineet voivat olla jopa 100 kertaa parhaita perinteisiä kiintolevyjä tehokkaampia satunnaisissa luku- ja kirjoitusoperaatioissa. Jos sovellus edellyttää hyvää skaalautuvuutta, myös yhteislevyarkkitehtuurista voi siten tulla varteenotettava vaihtoehto. Näiden vaihtoehtojen lisäksi Michael Rys [Rys10] antaa esimerkin miten myös relaatiotietokantajärjestelmä voi skaalautua tehokkaasti sovelluslogiikkaan rakennetun asynkronisen viestinnän avulla. Relaatiotietokantaa hyödyntävä sovellus voi esimerkissä näyttää ajan tasalla olevat tiedot nopeammin paikallisesti, mutta muihin tietokannan sirpaleisiin tiedot levitetään viiveellä asynkronisesti. Tämä tarkoittaa tietysti sitä, että tietokanta ei ole koko ajan oikeellisessa tilassa, joten myös relaatiotietokantajärjestelmän transaktioiden ACID-ominaisuuksia löyhennetään.

NoSQL-paradigman ominaispiirteisiin kuuluu, että tietoa varastoidaan useista halvoista tietokoneista koostuvien tietokonerypäiden solmujen levyille. Nämä järjestelmät skaalautuvat esimerkiksi lisäämällä raakaa laskentatehoa eli solmujen määrää tarvittaessa. Useissa lähteissä korostetaan kuitenkin myös yksittäisen solmun suorituskyvyn merkitystä [AnT10, LLC12, StC11]. Hajautetun järjestelmän tehokkuuteen vaikuttaa myös yksittäisen laskentasolmun tehokkuus. Tämä on ei-triviaalia kuitenkin myös siksi, koska mitä tehokkaammin yksittäinen solmu pystyy käsittelemään tietoja, sitä vähemmän solmuja tarvitaan saman tehtävän suorittamiseen. Jos solmuja tarvitaan vähän, ei tietokonerypäidenkään tarvitse olla suuria. Näiden pienempien tietokonerypäiden etuna on, että niissä on vähemmän laitteita. Mitä vähemmän laitteita tarvitaan, sitä enemmän säästetään hankintakustannuksia, energiaa sekä muita palvelinsalin resursseja. Jos taas on vähemmän laitteita, niin laitevikojen kokonaismäärä vähenee. Laitteiden korjaamiseen käytettäviä resursseja on siten mahdollista säästää. Laadukkaat tietokoneet kaatuvat silti joskus, joten hajautetun tietokantajärjestelmän tulee edelleen joka tapauksessa varautua laitevikoihin. Relaatiotietokantajärjestelmät taas perinteisesti toimivat tehokkaammissa palvelimissa, joita yleensä tarvitaan vähemmän. Pavlo ja kumppanit [PPR09] toteavat, että relaatiomalliin perustuva rinnakkaistietokantajärjestelmä voi usein olla monissa laskentatehtävissä aivan yhtä tehokas kuin tuhansista solmuista koostuva tietokoneryväs, vaikka sama tehtävä suoritetaan pienemmässä määrässä solmuja. Tämän takia on parempi keskittyä ohjelmoimaan parempia ja suorituskykyisempiä algoritmeja, kuin vain lisätä raakaa laskentatehoa.

NoSQL-tietokantajärjestelmien tietoturvaominaisuudet saattavat olla puutteellisia. Kehittymätön tiedostojen salaus, yksinkertaiset tunnistautumiskäytännöt sekä puutteelliset käyttöoikeustasot voivat johtaa tietoturvaongelmiin. Laajan mittakaavan Web 2.0 -sovelluksissa voidaan käsitellä myös arkaluonteista tietoa. Vaikka esimerkiksi sosiaalisen median sovellusten luonteeseen usein kuuluu, että käyttäjät jakavat tietojaan laajalle käyttäjäkunnalle, kaikkea näiden sovellusten käsittelemää tietoa ei kuitenkaan aina ole tarkoitettu koko maailman katsottavaksi. Sosiaalisen median sovelluksiin ladatun tiedon käyttäjiä halutaan usein rajoittaa jollain perusteella. Tiettyjen arkaluonteisten asioiden ei haluta joutuvan vääriin käsiin. Osa tiedoista saattaa olla tarkoitettu vain omalle lähipiirille, kuten sukulaisille tai lähimmille ystäville. Tämän lisäksi käytössä on sovelluksia sähköpostiin verrattavaan henkilökohtaiseen viestintään. Facebookilla [Fac12] on esimerkiksi toiminto yksityisviestejä varten, jota monet käyttävät sähköpostin sijasta henkilökohtaisten viestin välitykseen.

Relaatiotietokantajärjestelmissä on usein tietokantatasolla jokseenkin pitkälle kehitettyjä tietoturvaominaisuuksia. Okman ja kumppanit [OGG11] toteavat, että NoSQL-tietokantajärjestelmillä taas on paljon kehittämisen varaa ennen kuin ne voivat tarjota Web 2.0 -sovelluksille turvallisen ympäristön arkaluontoisten tietojen suojaamiseen. NoSQL-tietokantajärjestelmät ovat keskittyneet tietojen hyvään saatavuuteen ja suorituskykyyn. Tietoturvaominaisuudet on usein jätetty vähemmälle huomiolle, vaikka saatetaan käsitellä hyvin yksityisiä ja luottamuksellisia tietoja. Tietokannan levyllä sijaitsevia tiedostoja ei usein ole suojattu automaattisesti mitenkään, joten niissä sijaitseviin tietoihin voi päästä suoraan käsiksi. Sovelluksien ja tietokannan välistä tietoliikennettä taas ei välttämättä salata lainkaan, joten verkossa liikkuvia tietoja voi vakoilla. Tietokonerypäiden sisäistä liikennettä ja syötetietojen lataamista solmun levylle ei lisäksi yleensä ole suojattu tai salattu lainkaan. Rypäiden välisen verkkoliikenteen salaaminen voi myös olla puutteellista. Tietokantajärjestelmiä ja sinne tallennettuja tietoja hyödyntää usein monta prosessia, sovellusta ja käyttäjää. Käyttöoikeuksien ja käyttäjän tunnistaminen eivät kuitenkaan välttämättä ole NoSQL-tietokantajärjestelmissä kovin kehittyneellä tasolla. Käyttäjät ja käyttöoikeudet tallennetaan yksinkertaiseen tiedostoon riittämättömästi suojatussa muodossa, esimerkiksi jopa aivan selväkielisenä tekstinä. Tämä tiedosto pitää lisäksi olla tallennettuna oikeellisesti järjestelmän jokaiseen solmuun, muuten käyttäjällä voi olla voimassa vääriä oikeuksia. Käyttöoikeuksien rakeisuus saattaa myös olla karkeatasoinen. Vaihtoehtoina saattaa olla antaa joko pelkät lukuoikeudet tiettyihin tie-

tokannan tietoihin tai vaihtoehtoisesti täydet oikeudet tietokannan kaikkiin tietoihin. NoSQL-tietokantajärjestelmissä käytettävien erilaisten ohjelmointi- ja kyselykielten välityksellä on mahdollista tehdä myös *injektiohyökkäyksiä* (injection attacs), jolla järjestelmälle haitallisia komentoja liitetään sovelluksen kautta välitettävään tietokantakutsuun. Sovelluksen pitäisi siis osata tarkistaa kaikkien tietokantakyselyjen oikeellisuus ennen niiden välittämistä järjestelmälle. Edellä mainittuja puutteellisia tietoturvaominaisuuksia voidaan kuitenkin yleensä parantaa NoSQL-järjestelmän omien rajapintojen muunnelluilla toteutuksilla, sovelluslogiikassa tai joillain muulla varsinaisesta NoSQL-järjestelmästä riippumattomalla toimenpiteellä. Ylipäätään minkään tietokantajärjestelmän tietokantatason tietoturvaominaisuudet eivät ole aina kaikissa tilanteissa riittäviä, vaan osa tietoturvaan liittyvistä asioista joudutaan silti usein hoitamaan sovelluslogiikassa.

NoSQL-tietokantajärjestelmistä voi lisäksi puuttua lisäominaisuuksia ja työkaluja, kuten raportointityökalut. Edellä kuvattujen relaatiotietokantojen pitkälle kehitettyjen ominaisuuksien puuttuminen tekee järjestelmästä toisaalta joustavan ja yleisluontoisen, mutta samalla se tarkoittaa, että sovellusohjelmoija joutuu itse ohjelmoimaan sovelluslogiikkaan kaiken mahdollisesti tarvittavan lisätoiminnallisuuden, joka relaatiotietokantajärjestelmissä on yleensä oletuksena valmiina olemassa. Tietokannan loogisen mallin taas on tarkoitus olla riippumaton fyysisestä toteutuksesta. Ohjelmointirajapinnan ja deklaratiivisen kyselykielen tarkoituksena on tarjota looginen rajapinta tietokannan hyödyntämiseen niin, ettei sovellusohjelmoijan tarvitse välittää tietokannan fyysisestä rakenteesta. NoSQL-tietokantajärjestelmissä loogisen mallin ja fyysisen rakenteen välinen raja kuitenkin välillä hämärtyy sovellusohjelmoijan näkökulmasta. Sovellusohjelmoijan on usein otettava kantaa fyysiseen toteutukseen, kuten hajautetun tietokannan tietojen fyysiseen sijoitteluun. Tämä tulee esille luvuissa 6 ja 7. Tietokannan käsittelyyn taas tarjotaan yleensä joku alemman tason ohjelmointikieli, josta puuttuu SQL-kyselykielen deklaratiivinen ilmaisukyky. Tällaisella kielellä ohjelmointi saattaa kuitenkin olla hyvin työlästä.

NoSQL-järjestelmien ylläpito saattaa olla hankalaa. Tätä käsiteltiin sovellusohjelmoijan näkökulmasta edellä. Ongelmien selvittely voi lisäksi jäädä täysin sovellusohjelmoijan vastuulle, joten siihen voi kulua paljon aikaa [Lea10, WMH11]. Useat NoSQL-tietokantajärjestelmät perustuvat avoimeen lähdekoodiin. Tällaiset järjestelmät ovat yleensä

ilmaisia. Kalliiden kaupallisten tietokantajärjestelmien mukana tulee kuitenkin koko joukko valmiita apuvälineitä ja työkaluja. Järjestelmän toimittajalta voi olla myös helpompi saada käyttäjätukea ja lisäpalveluja. Avoimen lähdekoodin järjestelmiin tukea voi etsiä lähinnä erilaisen käyttäjä- ja kehitysyhteisöjen kautta. Järjestelmän mahdolliset virheet voi joutua myös korjaamaan itse. Stonebraker ja kumppanit [StC11] katsovat asiaa kuitenkin toisesta näkökulmasta ja toteavat, että esimerkiksi monet verkkokaupat haluavat käyttää pelkästään avoimeen lähdekoodiin perustuvia järjestelmiä. Avoimen lähdekoodiin perustuvien järjestelmien käyttö ei ole yhtä sitovaa ja antaa enemmän mahdollisuuksia. Huonoimmassa tapauksessa kalliin kaupallisen järjestelmän hankkinut yritys voi olla jatkossa sidottu pitkäksi aikaa kalliisiin järjestelmäpäivityksiin, korkeisiin ylläpitokustannuksiin sekä mahdollisesti riittämättömään käyttäjätukeen, koska siirtyminen johonkin toiseen tuotteeseen saattaa käydä liian kalliiksi. Yritys ei ole niin riippuvainen järjestelmän toimittajasta, koska tukea voi saada eri tahoilta muualta. Tämän lisäksi järjestelmään voi kehittää itse korjauksia.

NoSQL-paradigmaan perustuvat järjestelmät saattavat olla vielä täysin tuntemattomia yrityksille. Yritykset eivät välttämättä vielä tunne näiden järjestelmien ominaisuuksia, mahdollisuuksia tai rajoitteita [Lea10, Sto11]. Saattaa olla myös epäselvää miten nämä järjestelmät eroavat perinteisistä relaatiotietokantajärjestelmistä. Laajan mittakaavan hajautettuun ympäristöön tarkoitettuja relaatiotietokantajärjestelmiä on myös olemassa. NoSQL-järjestelmät ovat lisäksi erikoistuneita ja siksi kovin heterogeenisiä. NoSQL-tietokantajärjestelmät saattavat poiketa ominaisuuksiltaan ja toteutustavoiltaan hyvin paljon toisistaan. Näiden järjestelmien tietokannan tietomallit sekä toteutukset poikkeavat toisistaan. NoSQL-tietokantajärjestelmiä ei ole edes luokiteltu kovin täsmällisesti. Yritysten pitäisi kuitenkin jollain perusteella voida vertailla näiden järjestelmien ominaisuuksia helposti sekä toisiinsa että relaatiotietokantajärjestelmiin, jotta voidaan hahmottaa sopivatko ne omiin vaatimuksiin. NoSQL-tietokantajärjestelmien ohjelmointirajapinta, käyttöliittymä sekä ohjelmointi- tai kyselykieli saattavat olla erilaisia. Sovellusohjelmoijalle voi olla haastavaa omaksua näiden sekä toisistaan että mahdollisesti relaatiotietokantajärjestelmistä poikkeavien erilaisten mallien käyttö. Relaatiotietokantajärjestelmät ja SQL-kyselykieli taas tarjoavat yleisessä käytössä olevan standardin tietokannan tietomallin ja kyselykielen sekä sitä kautta paljon tutumman käyttöympäristön.

Yrityksen sovellus ja sen hyödyntämä tietokantajärjestelmä saattaa olla hyvin kriittinen

yrityksen liiketoiminnan menestyksen kannalta. Kaikkien tietojen tulee olla mahdollisimman oikeellisia ja pysyviä. Pienikin virhe järjestelmän toiminnassa saattaa tulla hyvin kalliiksi. Jos tietokannan laatua ei voida taata tietokannan transaktioiden ACID-ominaisuuksilla, virheitä voi esiintyä todennäköisemmin. Tällaista riskiä ei yleensä voida ottaa. ACID-ominaisuuksiin luotetaan myös liiketoiminnan kannalta kriittisissä transaktiokeskeisissä OLTP-järjestelmissä [Sto11]. OLTP-tyyppisten transaktioiden voisi ajatella sopivan hyvin NoSQL-paradigman periaatteisiin, koska nämä yksinkertaiset luku- tai kirjoitusoperaatiot saattavat joissain sovelluksissa koskea aina vain yhtä riviä. ACID-ominaisuuksista ei kuitenkaan haluta välttämättä luopua, koska useampaa riviä koskeviin transaktioihin saattaa olla tarvetta joskus tulevaisuudessa.

NoSQL-tietokantajärjestelmät noudattavat yleensä tietokannan laadun takaavia tietokannan transaktioiden ACID-ominaisuuksia löyhempää BASE-oikeellisuusmallia, jossa tietojen oikeellisuus saavutetaan vasta jossain myöhemmässä tilassa. Tämä oikeellisuuden saavuttamisen taso saattaa kuitenkin vaihdella NoSQL-tietokantajärjestelmien toteutuksissa [BeT11]. Yrityksen pitää valita tietokantajärjestelmä, jonka ACID- ja BASE-ominaisuudet vastaavat sovelluksen vaatimuksia. Näiden ominaisuuksien mittaaminen ja arviointi saattaa kuitenkin olla vaikeaa, esimerkiksi monien erilaisten toteutusten takia. Oikeellisuusominaisuudet voidaan myös jakaa eri luokkiin. Oikeellisena tilana voidaan pitää esimerkiksi tietokannan nykytilaa, jossa kaikki tietokannan toisinteet sisältävät samat tiedot. Toisaalta oikeellisena tilana voidaan pitää tilaa, jolloin sovelluksen käyttäjä näkee oikeat tiedot. Tämä tila voi taas poiketa edellä kuvatusta tietokannan nykytilasta. NoSQL-tietokantapalvelunakaan ei ratkaise tätä ongelmaa. Yritys joutuu myös tällaisessa palvelussa arvioimaan oikeellisuuden tasoa. Sovellusohjelmoijalla voi lisäksi olla mahdollisuus vaikuttaa jossain määrin sovelluksen tietojen oikeellisuuden tasoon. Bermbach ja kumppanit [BeT11] esittelevät kuitenkin tavan, jonka avulla sovellusohjelmoija voi arvioida tietokantajärjestelmiä myös niiden oikeellisuusominaisuuksien perusteella. Yrityksen omat vaatimukset hajautetun tietokantajärjestelmän luotettavuuden takaaville ominaisuuksille eivät kuitenkaan välttämättä ole vielä täysin selvillä [Rys11]. Saatavuuden ja tehokkuuden sekä oikeellisuuden ja eristyvyyden välillä on siis tehtävä kompromissi. Tämä tarkoittaa, että yrityksen on priorisoitava toimintojaan ja selvitettävä sovelluksen sekä käsiteltävän tiedon vaatimukset. Pitääkö yrityksen tietojen olla mahdollisimman hyvin saatavilla vai onko tärkeämpää pitää tieto koko ajan oikeellisessa tilassa? Vai riittääkö, että tietojen oikeellisuus saavutetaan vasta jossain myö-

hemmässä vaiheessa? Jos tietojen ei tarvitse olla aina oikeellisia, pitää myös selvittää oikeellisuuden taso eli kuinka vanhoja tietoja mahdollisesti sallitaan. Näiden vaatimuksien lisäksi tietokantajärjestelmän valintaan vaikuttavat monet edellä käsitellyt ominaisuudet. Näitä ovat esimerkiksi sovelluksen tietokantaoperaatioiden määrä ja tyyppi, sovellusalueen tietojen rakenne ja rakenteen vaatima tietokannan joustavuus sekä järjestelmältä edellytettävä skaalautuvuus, saatavuus ja tehokkuus. Tietokantajärjestelmän luotettavuudella, kestävyydellä ja tietoturvalla saattaa myös olla suuri merkitys. Näiden seikkojen lisäksi yrityksen tulee huomioida tietokantajärjestelmän käytettävyys ja ylläpidettävyys.

Yritykset voivat olla haluttomia siirtymään käyttämään jotain NoSQL-ratkaisua edellä mainituista syistä. Useat merkittävät yritykset käyttävät edelleen perinteisiä relaatiotietokantajärjestelmiä ydintoimintojensa taustalla [Rys11, Sto11]. Näihin kuuluu myös hajautettuja ja hyvää skaalautuvuutta edellyttäviä elektroniseen kaupankäyntiin tai pankkitoimintaan liittyviä sovelluksia. Nämä järjestelmät voivat toimia myös transaktiokeskeisten OLTP-järjestelmien tapaan. Jopa tunnetut laajan mittakaavan Web 2.0 -sovellukset, kuten Facebook [Fac12], Twitter [Twi12] ja MySpace [Mys12] ovat käyttäneet edelleen pitkälle NoSQL-ilmiön aikanakin ainakin osittain myös relaatiotietokantajärjestelmiä toiminnassaan. Myspace käyttää tietojen varastoimiseen Microsoft SQL Serveriä [Mic12a] [Mic09]. Facebook ja Twitter ovat käyttäneet MySQL-relaatiotietokannan hallintajärjestelmää [Ora12e] [Sob07, Twi10]. Twitter on esimerkiksi varastoinut Tweet-viestit edelleen NoSQL-ilmiön aikana MySQL-tietokantaan [Twi10]. Twitter kuitenkin käyttää useita eri tyyppisiä tietokantajärjestelmiä ja työkaluja toiminnoissaan. Twitter käyttää NoSQL-tietokantajärjestelmiä sellaisiin tehtäviin, joihin relaatiotietokantajärjestelmä ei ole paras vaihtoehto. Tästä kerrotaan tarkemmin jatkossa.

Neil Leavitt [Lea10] toteaa, että NoSQL-tietokantajärjestelmät ovat kannattajiensa mielestä niin joustavia, koska ne eivät noudata relaatiotietokantajärjestelmien kaikkia ominaisuuksia. Tämän ansioista sovellusohjelmoija pystyy hyödyntämään niitä relaatiotietokantajärjestelmiä paremmin. Jaroslav Pokornyn [Pok11] mielestä NoSQL-tietokantajärjestelmät eivät korvaa relaatiotietokantajärjestelmiä, koska NoSQL-järjestelmät ovat vielä kovin kaukana pitkälle kehittyneestä relaatiotietokantajärjestelmien tekniikasta. Michael Rys [Rys11] on taas sitä mieltä, että tietokantaa hyödyntävien skaalautuvien sovellusten kehittäminen ei niinkään riipu käytettävästä tietokantajärjestelmästä, vaan

tarpeeksi ketterästi skaalautuvasta sovellusarkkitehtuurista. Lopuksi Rick Cattell [Cat 10] tulee siihen johtopäätökseen, että mikään tietokantajärjestelmä ei ole paras kaikkiin tehtäviin. Soveltuva tietokantajärjestelmä riippuu täysin yrityksen ja sovelluksen vaatimuksista sekä näiden vaatimusten tärkeysjärjestyksestä. Tämän lisäksi hänen mielestään relaatiotietokantajärjestelmät pystyvät myös ainakin teoriassa skaalautumaan, jos tietokantaan kohdistuvat operaatiot ovat pieniä ja suppeita.

8.3 Useamman erilaisen tietokantajärjestelmän ja työkalun hyödyntäminen

Mikään tietokantajärjestelmä tai työkalu ei ole paras kaikkiin tehtäviin. Kunkin järjestelmän tulisi hoitaa suosiolla ne tehtävät joissa on toista parempi, eikä yritetä kehittää vain yhtä yksi koko sopii kaikille -järjestelmää, jonka odotetaan hoitavan kaikki nämä tehtävät hyvin. NoSQL-tietokantajärjestelmät ovat erikoistuneita ja kovin heterogeenisiä keskenään, kukin järjestelmä on kehitetty tiettyjen vaatimusten perusteella ja tietynlaista ongelmaa varten. Kussakin järjestelmässä on järkevää ja tehokasta käsitellä ja varastoida pääosin tietyn kaltaisen sovellusalueen tietoa. Tämän takia näiden eri järjestelmien ominaisuudet, kuten tietokannan tietomalli, ohjelmointirajapinta ja ohjelmointi- tai kyselykieli sekä fyysinen toteutus voivat poiketa toisistaan. Tietynlaista sovellusalueen tietoa on yleensä järkevää mallintaa tietynlaisella tietomallilla, joka siten voidaan toteuttaa mahdollisimman tehokkaasti. Tavoitteena on hyödyntää kunkin tietokantajärjestelmän vahvuuksia oikeassa kohtaa. Yrityksen tulee selvittää millaisia vaatimuksia sillä on käsiteltäville tiedoille. Jos yrityksellä on paljon erilaisia vaatimuksia, yhtä kaikkiin vaatimuksiin sopivaa tietokantajärjestelmää ei kuitenkaan välttämättä ole saatavilla. Vaihtoehtona saattaa silloin olla useamman erilaisen, juuri tietyn ongelman ratkaisevan tietokantajärjestelmän ja työkalun hyödyntäminen.

Edes kaikki laajan mittakaavan Web 2.0 -sovelluksetkaan eivät käytä samaa tietokantajärjestelmää ja työkaluja kaikkiin tehtäviin. Twitter [Twi12] käyttää esimerkiksi useita erilaisia tietokantajärjestelmiä ja työkaluja toiminnoissaan [Wei10]. Twitter käyttää NoSQL-tietokantajärjestelmiä sellaisiin tehtäviin, joissa relaatiotietokantajärjestelmä ei ole paras vaihtoehto. Twitter käsittelee paljon tietoa päivässä. Nämä tiedot ovat erityyppisiä. Palvelun kautta varastoidaan paljon rakenteellista tietoa, kuten esimerkiksi Tweet-viestit, välitetyt Tweet-viestit, käyttäjätiedot, *käyttäjien seurantatiedot* (followers), rekisteröidyt puhelintiedot, suosikit, tallennetut haut, sijaintitiedot, autentikointitiedot

sekä kolmansien osapuolien asiakastiedot. Twitteriin rekisteröitynyt käyttäjä voi määritellä keiden muiden käyttäjien Tweet-viestejä hän voi vastaanottaa. Tämä tarkoittaa, että käyttäjä seuraa toista käyttäjää. Palvelun käytöstä kertyy lisäksi eri tyyppistä semirakenteellista tietoa, kuten kirjautumis- ja hakutoimintojen lokeja. Twitterin käyttöön liittyy myös käyttäjien välisiä suhteita, kuten edellä mainittu seurantasuhde, joista muodostuu sosiaalisen verkon kaltaisia rakenteita.

Näiden tietojen perusteella lasketaan erilaisia Twitter-palvelun [Twi12] käyttöä ja toimintaa analysoivia tilastoja, kuten palvelun hakujen määrä päivässä tai palvelussa päivän aikana prosessoitujen pyyntöjen lukumäärä ja keskimääräinen suoritusaika [Wei10]. Osa laskentatehtävistä on hyvin monimutkaisia, kuten eri tyyppisten käyttäjien vertailu. Twitter analysoi käyttäjien tietoja verratakseen palvelun suurkuluttajia, mobiilikäyttäjiä sekä kolmansien osapuolien asiakkaiden käyttäjiä. Twitter haluaa tietää hyödyntävätkö nämä käyttäjät palvelua eri tavalla kuin tavalliset käyttäjät. Tavoitteena on esimerkiksi selvittää miten tavallisesta käyttäjästä tulee palvelun suurkuluttaja. Tekeekö esimerkiksi jonkun tietyn käyttäjän seuraaminen tai jonkun tietyn palvelun ominaisuuden käyttäminen ihmisestä palvelun suurkuluttajan? Lisäksi analysoidaan esimerkiksi minkä tyyppisiä Tweet-viestejä välitetään eniten, minkälaiset sosiaalisen verkon rakenteet ovat mahdollisimman tehokkaita sekä kuinka oikeiden ihmisten lähettämät Tweet-viestit erotellaan jonkun automaatin lähettämistä viesteistä.

MySQL-relaatiotietokannan hallintajärjestelmä [Ora12e] ei ole ollut riittävän tehokas kaikkiin edellä kuvattuihin tehtäviin [Wei10]. Näihin tehtäviin Twitter hyödyntää laajan mittakaavan ympäristöä. Tiedot varastoidaan tietokonerypäiden koneille hajautettuun tietokantajärjestelmään. Tämä infrastruktuuri perustuu Apache Hadoop -toteutukseen [The12b], jota hyödynnetään tietojen varastointiin ja rinnakkaiseen käsittelyyn. Hadoop MapReduce -rajapinnan avulla lasketaan edellä mainittuja monimutkaisia analyyttisiä tehtäviä. Twitter varastoi käyttäjien tietoja *HBase-varastoon* [The12c]. Apache HBase on Google Bigtable -tietomalliin perustuva avoimen lähdekoodin sarakeperhevarasto. Myös HBase toimii Apache Hadoop -ympäristössä. Twitter hyödyntää HBasea MapReduce-ohjelmien syötetietojen lähteenä ja laskennan lopputuloksen kohteena. Java on osoittautunut liian monimutkaiseksi kieleksi, joten Twitter käyttää korkeamman tason Pig Latin -kieltä [ORS08, The12g] MapReduce-tehtävien ohjelmoimiseen [Wei10]. Twitter käyttää myös *FlockDB-verkkotietokantaa* (FlockDB graph database) [Twi11]

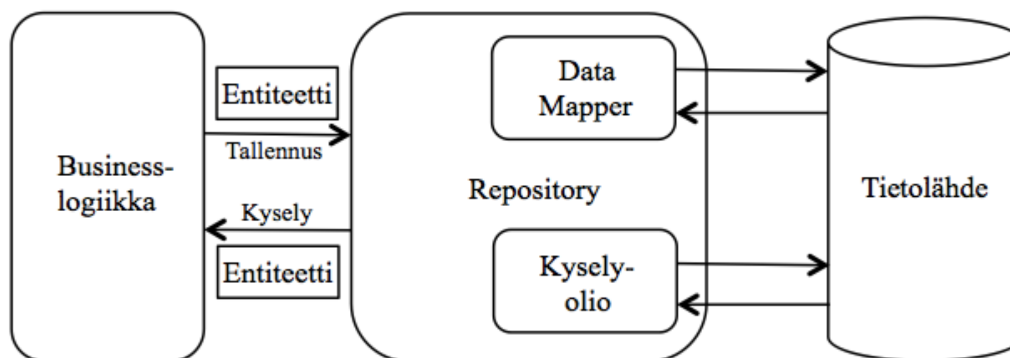
sosiaalisten verkkojen analysointiin [Wei10]. FlockDB on tosiaikainen hajautettu avoimen lähdekoodin verkkotietokanta. Twitter tallentaa FlockDB:n tiedon kuka seuraa palvelussa ketäkin eli tietokantaan tallennetaan käyttäjien keskinäisiä suhteita. Tästä syntyy verkkomainen rakenne. Näiden suhteiden avulla myös rajoitetaan Tweet-viestien näkyvyyttä eri käyttäjille. Twitterillä on lisäksi ollut tuotantokäytössä *Apache Cassandra* [LaM10, The12e], esimerkiksi käyttäjien sijaintitietojen tallentamiseen sekä tiedon louhintaan [Wei10]. Cassandra on alun perin Facebookin [Fac12] kehittämä avoimen lähdekoodin sarakeperhevarasto. Twitter kuitenkin edelleen myös testaa Cassandraa [Wei10]. Cassandraa yritetään hyödyntää esimerkiksi laajan mittakaavan tosiaikaiseen laskentaan. Lokien käsittelyyn Twitter hyödyntää *Sribe-kehystä* (Scribe framework) [Git12] [Wei10]. Scribe on Facebookin kehittämä lokitietojen koostamiseen tarkoitettu skaalautuva ja laajennettava työkalu [Git12]. Scribe toimii laajan mittakaavan tietokonerypäissä. Lokitiedot koostetaan tosiaikaisesti useammasta solmusta yhdelle keskitetylle Scribe-palvelimelle.

Miten useampaa erilaista tietokantajärjestelmää ja työkalua voidaan hyödyntää käytännössä? Eräs tapa yhtenäistää ja helpottaa useamman erilaisen tietokantajärjestelmän käyttöä on käyttää jotain *väliohjelmaa* (middleware), joka hoitaa tietojen välittämisen, linkittämisen ja muuntamisen sovelluksen ja eri tietokantojen välillä. Tällaisen väliohjelman voi korvata myös ohjelmistokehys ja sen sisältämät komponentit. *Spring Data* (Spring Data) [Spr12b] on esimerkiksi Spring-ohjelmistokehyksen [Spr12a] avoimen lähdekoodin projekti, jonka yhteen kokoamien aliprojektien komponenttien avulla sovellus voi hyödyntää relaatiotietokantajärjestelmiä, NoSQL-tietokantajärjestelmiä, MapReduce-laskentaa sekä pilvipalveluja. Spring Data on korkeamman tason projekti tai *moduuli* (module), joka kokoaa yhteen kuhunkin tietokantajärjestelmään liittyviä aliprojekteja tai *alimoduuleja* (sub-module) [Spr12b]. Jokaiselle alimoduulille kehitettyjen komponenttien avulla sovellukselle voidaan määritellä pääsy johonkin tietokantaan. Näitä projekteja on perustettu ohjelmistokehyksille ja rajapinnoille kuten JPA, jonka avulla voidaan hyödyntää relaatiotietokantajärjestelmiä. Alimoduuleja on perustettu myös erilaisille NoSQL-tietokantajärjestelmille, kuten Apache Hadoop [The12b], Apache HBase [The12c], Redis, MongoDB ja Neo4j. MapReduce-laskentaa taas voidaan hyödyntää Apache Hadoopin [The12b] kautta. Edellä mainittuja alimoduuleja kehittävät yleensä kuhunkin tietokantajärjestelmään perehtyneet sovellusohjelmoijat ja yritykset. Spring Data käyttää *repository-ratkaisumallia* (repository pattern). Ker-

rosarkkitehtuuria noudattavassa sovelluksessa on reposity-ratkaisumallissa business-logiikan ja tietokannan välillä oma kerros [Mic12e]. Repository-ratkaisumallissa business-logiikka ei käsittele suoraan tietolähdekerrosta, vaan välissä on repository-kerros joka hoitaa tietojen välittämisen, linkittämisen ja muuntamisen sovelluksen ja tietolähteen välillä. Tietolähde voi olla esimerkiksi tietokanta tai joku muu palvelu. Tätä kerrosta voidaan kutsua kerrosarkkitehtuurissa myös *tietokerrokseksi* (data layer) [Mic12d]. Kerrosarkkitehtuurissa jokin toiminnallisuuskokonaisuus yhdistetään ja keskitetään samaan kerrokseen niin, että sovellusta on helpompi ymmärtää, määritellä ja ylläpitää. Tietokerros sisältää keskitetysti kaiken logiikan, jonka avulla käsitellään alempaa tietolähdekerrosta. Tietokerros välittää sovelluksen kyselyjä tietolähteeseen, linkittää, muuntaa ja integroi sovellusalueen tietoja tietolähteen tietomalliin sekä välittää käsiteltävien tietojen muutoksia tietolähteeseen.

Repository toimii eri sovellusalueiden operaatioiden ja tietojen lähteen yhdistäjänä. Tietojen yhdistämiseen käytetään usein *data mapper -suunnittelumallia* (data mapper), jossa tehdään tietokytkös olion ja tietokannan välillä [Mar12, Mic12e]. Tietokytköksenä toimiva komponentti siirtää tietoa olion ja tietokannan välillä sekä pitää ne erillisinä toisistaan. Tietokytkös tehdään kääntämällä tiedon esitystapa kunkin osapuolen ymmärtämään muotoon eli esimerkiksi määritellään sovellusalueen tietojen vastaavuus tietokannan tietojen kanssa [Mar12]. Tämä menettelytapa tekee sovelluksen riippumattomaksi käytettävän tietolähteen tekniikasta. Sovelluksen ei tarvitse tietää millä tekniikalla tietokantaa hyödynnetään. Sovellus ainoastaan kutsuu repositorysta kunkin tietolähteen omaa laajennettua rajapintaa [Mic12e]. Repository hoitaa kaiken kommunikoinnin tietolähteen kanssa soveltuvalla tekniikalla. Se eristää nämä toiminnot sovelluslogiikasta ja tietojen lähteestä. Tämän ansiosta kaikkiin mahdollisiin tietokytköksiin voidaan soveltaa yhtenäisesti keskitetyssä repositoryssa hallinnoituja sääntöjä ja logiikkaa.

Kuvassa 15 esitetään repository-ratkaisumallin toiminta. Tietolähteeseen operoiva logiikka erotetaan business-logiikasta. Sovelluksen business-logiikkakerros lähettää entiteetin repositoryyn päivitystä tai poistoa varten. Business-logiikan ja tietolähteen välillä toimiva repository tekee tietokytköksen tietolähteeseen sekä päivittää tietolähteen data mapper-suunnittelumallin avulla. Vastaavasti repository tekee kyselyn tietolähteeseen ja palauttaa ajan tasalla olevan entiteetin business-logiikkakerrokseen. Business-logiikka ei missään vaiheessa ole suoraan kytköksessä tietolähteeseen.



Kuva 15: Repository-ratkaisumallin toiminta [Mic12e].

Spring Data repositoryssa ei ole yhtä kaikkien tietokantajärjestelmien kanssa yhteensopivaa ohjelmointirajapintaa [PGR12]. Jokaisella alimoduulilla on sen sijaan oma Spring-ohjelmistokehyksen ohjelmointimallin mukaan toteutettu rajapinta tietojen yhteensovittamiseen. Rajapinnat on toteutettu Spring-ohjelmistokehyksen *template-suunnittelumallin* (template pattern) avulla. Spring template -suunnittelumalli on luokka, joka sisältää valmiita metodeja usein käytettävien operaatioiden suorittamiseen. Tällainen template-luokka voi sisältää esimerkiksi alimoodulikohtaisen metodin tietojen muunnoksia varten. Spring Datassa on lisäksi kaikille yhteinen abstrakti luokka, jonka alimooduulikohtaisella toteutuksella tavanomaisille tietokannan CRUD-operaatioille voidaan ohjelmoida tietokytkös business-logiikasta tietolähteeseen mahdollisimman helposti olio-ohjelmointimallin mukaisesti. Tämä yhteinen luokka osaa hyödyntää kunkin käyttöön määritellyn alimoduulin tietokytköksen tekniikkaa. Sovellusohjelmoija voi esimerkiksi tehdä toteutuksen, jossa määritellään vain yksinkertaiset metodit kullekin operaatiolle. Tällainen voi olla esimerkiksi metodi, jossa määritellään että asiakkaan tiedot haetaan asiakasnumeron perusteella. Spring Datan infrastruktuuri suorittaa varsinaisen tietokytköksen kunkin alimoduulin valmiiksi määritellyn rajapinnan metodien perusteella. Edellä mainitut alimoduulikohtaiset toteutukset ja metodit taas noudattavat mahdollisimman paljon samankaltaista rakennetta. Nämä yhtenäisen mallin mukaan toteutetut rajapinnat helpottavat ja yksinkertaistavat työskentelyä erilaisten tietokantajärjestelmien kanssa. Yhtenäisen mallin mukaan suunniteltuja ja samankaltaisia uudelleen käytettäviä rajapintoja voidaan hyödyntää laajentamalla yhteiseen toimintalogiikkaan perustuen. Nämä yhtenäiset toiminnot siis vähentävät sovellusohjelmoijan työtä. Sovellusohjelmoijan ei tarvitse opetella paljon toisistaan poikkeavien erilaisten rajapintojen käyttöä. NoSQL-tietokantajärjestelmissä ei yleensä lisäksi ole mitään valmista toimintoa, jolla sovellusalueen tietoja voidaan mallintaa ja muuntaa kyseisen järjestel-

män tietokannan tietomalliin. Sovellusohjelmoija joutuu ohjelmoimaan itse tämän tyyppisen tietojen vastaavuusmäärityksen ja muuntamisen. Spring Datan alimoduulit kuitenkin sisältävät suuren osan tästä toiminnallisuudesta valmiina.

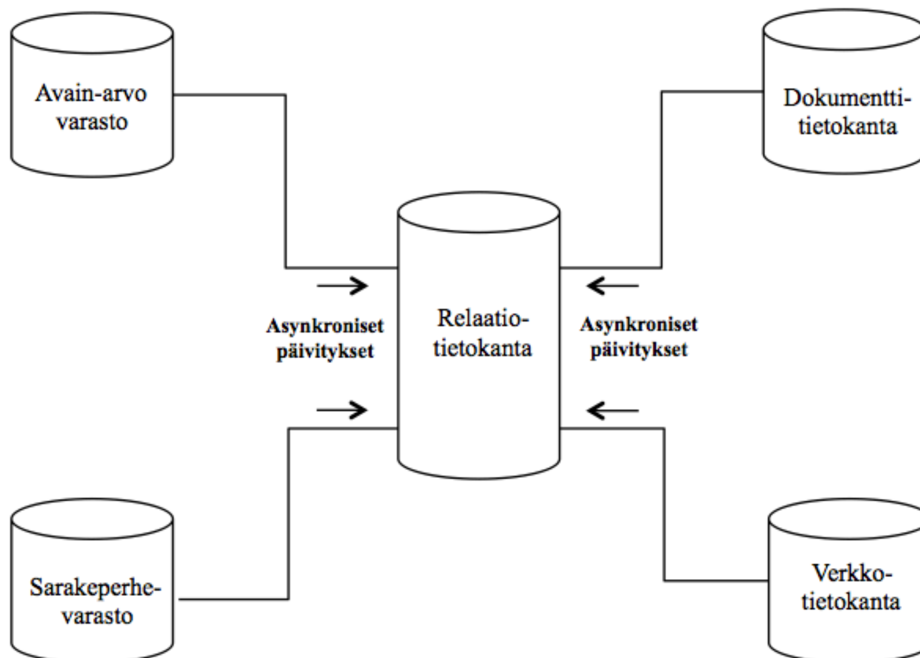
Spring-ohjelmistokehyksen Spring Data -moduuli siis helpottaa ja yhtenäistää joidenkin tietokantajärjestelmien käyttöä sovelluksessa. Erilaisia tietokantajärjestelmiä voidaan näin hyödyntää eri sovelluksissa. Tämän ansiosta voidaan kehittää myös yksi sovellus, jonka komponentit hyödyntävät useampaa eri tietokantajärjestelmää. Tätä kutsutaan joskus myös *monikieliseksi pysyvyydeksi* (polyglot persistence) [FoS12, Le08]. Monikielisyydellä tarkoitetaan, että kuhunkin yhden tai useamman sovelluksen jokaiseen tietojen varastointitarpeeseen voidaan valita parhaiten sopiva tietokantajärjestelmä. Pysyvyys taas viittaa sovelluksen tarpeeseen tallentaa tietoja pysyvästi tietokantaan. Spring Datan kautta voidaan kuitenkin hyödyntää vain niitä tietokantajärjestelmiä, joille on kehitetty oma Spring Data -alimoduuli. Tämä arkkitehtuuriratkaisu on silti joustava ja laajennettava. Uusia alimoduuleja voidaan kehittää ja lisätä tarvittaessa myöhemmin.

Kyseessä on silti vain yksi toteutus, jota voivat hyödyntää vain Spring-ohjelmistokehystä sekä Javaa käyttävät sovellukset. Sovelluksen tarvitsemalle tietokantajärjestelmälle pitää myös löytyä olemassa oleva Spring Data -alimoduuli. Saman kaltaista ratkaisua voi kuitenkin mahdollisesti soveltaa myös jossain toisessa ohjelmistokehyksessä. Useampien erilaisten tietokantajärjestelmien yhteiskäyttöä ajatellen on esitetty myös muita ratkaisuja. Debasish Ghosh esittää ratkaisuksi *multitietokantaparadigman* (multidatabase paradigm) [Gho10]. Paradigman avulla sovellus voi käsitellä eri sovellusalueiden keskenään heterogeenista tietoa siten, kuinka se on kunkin sovellusalueen tietojen kohdalla tehokkainta. Tässä arkkitehtuuritason ratkaisu on käyttää yhtä keskitettyä *päättävää varastojärjestelmää* (system of record) sekä useampaa erillistä ja mahdollisesti erilaista tietokantajärjestelmää, jotka kommunikoivat päättävän järjestelmän kanssa. Näitä kutsutaan joskus myös *edustavarastoiksi* (frontal datastore). Päättävä varastojärjestelmä toimii päättävässä asemassa useamman järjestelmän arkkitehtuurissa [Inm03]. Kyseinen päättävä varastojärjestelmä vastaa tietokannan laadusta ja turvallisuudesta. Jos muilla tietokantajärjestelmillä on ristiriita jonkin tiedon kohdalla, tietojen oikeellisuus ja semanttinen merkitys voidaan tarvittaessa tarkistaa päättävästä varastosta. Debasish Ghoshin arkkitehtuuriratkaisussa [Gho10] kyseessä on siis hajautettu tietokantajärjestelmä, joka koostuu monesta erilaisesta mahdollisesti eri pisteessä toimivasta tietokanta-

järjestelmästä sekä niihin operoivasta sovelluskerroksesta. Tällaista järjestelmää voidaan kutsua myös *monitietokantajärjestelmäksi* (multidatabase system) [SKS11 s. 857].

Debasish Ghoshin arkkitehtuuriratkaisussa [Gho10] päättävä tietokantajärjestelmä on keskitetty relaatiotietokantajärjestelmä. Tässä ratkaisussa ei kuitenkaan käytetä yhteistä tietomallia, vaan jokainen monitietokantajärjestelmän tietokanta voi noudattaa omaa tietokannan tietomalliaan. Relaatiotietokantajärjestelmän kanssa kommunikoivat muut tietokantajärjestelmät voivat olla esimerkiksi eri tietokannan tietomallia noudattavia NoSQL-tietokantajärjestelmiä, kuten avain-arvo-varasto, dokumenttivarasto, sarakeperhevarasto tai verkkotietokanta. Kyseiset NoSQL-tietokantajärjestelmät kommunikoivat keskitetyn relaatiotietokantajärjestelmän kanssa asynkronisen viestinnän avulla. Jokainen yksittäinen NoSQL-tietokantajärjestelmä varastoi ja palvelee vain oman sovellusalueensa tietoja. Osa tiedoista kuitenkin toisinnetaan päättävään relaatiotietokantajärjestelmään. Sovelluslogiikassa määritellään ne tiedot, jotka pitää toisintaa relaatiotietokantaan. Sovellus päivittää tiedot ensin NoSQL-tietokantaan, jonka jälkeen tarvittavat päivitykset laitetaan jonoon sekä toisinnetaan relaatiotietokantaan jossain myöhemmässä vaiheessa. Tämä tarkoittaa, että päättävän tietokantajärjestelmän tietokannan tiedot eivät ole aina oikeellisessa tilassa. Debasish Ghoshin ratkaisussa [Gho10] siis löyhennetään tietokannan transaktioiden ACID-ominaisuuksista.

Kuvassa 16 esitetään multitietokantaparadigmaa noudattavan multitietokantajärjestelmän arkkitehtuuri. Keskellä on päättävänä varastojärjestelmänä relaatiotietokantajärjestelmä. Tämän lisäksi kuvassa on neljä eri tietokannan tietomallia noudattavaa edustavarastona toimivaa tietokantajärjestelmää. Kunkin edustavaraston tietokanta palvelee vain oman sovellusalueensa tietojen varastona. Jokin väliohjelma levittää asynkronisesti kunkin edustavaraston tietokannan muutokset relaatiotietokantajärjestelmän tietokantaan.



Kuva 16: Multitietokantaparadigmaa noudattavan multitietokantajärjestelmän arkkitehtuuri [Gho10].

Tässä arkkitehtuuriratkaisumallissa kunkin NoSQL-tietokantajärjestelmän ominaisuuksia, kuten skaalautuvuutta, saatavuutta ja suorituskykyä voidaan hyödyntää täysipainoisesti. Järjestelmässä voidaan myös hyödyntää tehokkaasti kullekin sovellusalueelle sopivaa tietokannan tietomallia ja ominaisuuksia. Kaikki tarvittavat tiedot saadaan aina lopulta jossain vaiheessa oikeellisina myös päättävään relaatiotietokantaan. Tämä arkkitehtuuriratkaisu muistuttaa jossain määrin *tietovarastoa* (data warehouse), jossa yhteen keskitettyyn varastoon kerätään tietoa useammasta operatiivisesta tietokannasta. Debasish Ghoshin artikkelissa [Gho10] ei kuitenkaan mainita, että edustatietokantajärjestelmän pitää toimia operatiivisesti, tai että päättävän tietokantajärjestelmän tarvitse välttämättä toimia vastaavalla tavalla tietovarastona. Artikkelissa kuitenkin kuvataan vain hyvin yleinen arkkitehtuuritason ratkaisu. Ghoshin mukaan ratkaisu on yksinkertaisempi kuin jos apuna käytetäisiin jotain ohjelmistokehystä. Ratkaisu helpottaa hänen mukaansa sovellusohjelmointia, koska tietokytkös on yksinkertaisempi ja suora. Artikkelin mukaan jokin viestinvälitykseen suuntautunut väliohjelma toisintaa tiedot eri tietokantajärjestelmien välillä. Artikkelista ei kuitenkaan selviä tarkemmin mikä tämä väliohjelma voi olla. Artikkelissa ei lisäksi kerrota miten ja missä vaiheessa varsinainen tietokytkös sekä *semanttinen integrointi* (semantic integration) heterogeenisten NoSQL-tietokantajärjestelmien sekä keskitetyn relaatiotietokantajärjestelmän välillä määritellään. Useamman tietokantajärjestelmän laajuisessa multitietokantajärjestelmässä samannimisillä

ominaisuuksilla voi olla eri merkitys. Lukumäärää kuvaavat arvot voivat esimerkiksi tarkoittaa eri asioita. Paino voidaan ilmaista kiloina tai paunoina. Hinta voidaan ilmaista euroina tai dollareina. Nämä arvot pitää muuntaa joksikin sovituksi yhteiseksi arvoksi. Tätä kunkin usean eri sovellusalueen tietojen muuntamista yhteisesti sovittuun samaa tarkoittavaan muotoon kutsutaan joskus myös semanttiseksi integroinniksi [SKS11 s. 858]. Debasish Ghoshin artikkelissa [Gho10] todetaan kuitenkin lopuksi, että tämä multitietokantaparadigmaa noudattava arkkitehtuuriratkaisu voi johtaa mahdollisesti hyvin monimutkaiseen kokonaisjärjestelmään.

9 Yhteenveto

NoSQL-tietokantajärjestelmien nykyistä kehitysvaihetta voidaan verrata aikaan ennen SQL:ää. NoSQL-tietokantajärjestelmät ovat keskenään kovin heterogeeninen joukko. Nämä järjestelmät voivat poiketa paljon toisistaan sekä tietokannan loogisen mallin että fyysisen rakenteen osalta. Kussakin järjestelmässä on järkevää ja tehokasta käsitellä pääosin tietyn kaltaisen sovellusalueen tietoa. NoSQL-tietokantajärjestelmille yritetään silti kehittää yhteisiä standardeja, kuten yhteistä tietokannan kyselykieltä. Tällaisia kaikille yhteisiä ominaisuuksia ei kuitenkaan ole vielä otettu laajalti käyttöön.

Koska NoSQL-tietokantajärjestelmät ovat kovin heterogeenisiä järjestelmiä, niiden luokittelu on vaikeaa. Erilaisten NoSQL-tietokantajärjestelmien luokitteluun ei kuitenkaan ole olemassa mitään vakiintunutta mallia. Luokitteluun on käytetty erilaisia toisistaan vaihtelevia malleja. NoSQL-tietokantajärjestelmien tietokannan tietomallin yhteinen perusominaisuus on kuitenkin yleensä avain-arvo-pari. Käsitteellisesti yksinkertaiseen avain-arvo-pariin perustuvasta tietomallista on johdettu monimutkaisempia tietomalleja ja erikoistuneita toteutuksia. Monimutkaisempiin tietomalleihin saattaa kuitenkin liittyä enemmän rajoituksia, joten ne eivät ole yhtä joustavia kuin yksinkertaisemmat mallit. Nämä järjestelmät siis tasapainoilevat yksinkertaisuuden ja monimutkaisuuden, sekä vastaavasti joustavuuden ja rajoittavuuden välillä. Useimmat NoSQL-tietokantajärjestelmät perustuvat lisäksi myös avoimeen lähdekoodiin. Avoimen lähdekoodiin perustuvien järjestelmien käyttö säästää kustannuksia, antaa enemmän mahdollisuuksia eikä ole yhtä sitovaa kuin jonkun kaupallisen järjestelmän.

Tietyt NoSQL-tietokantajärjestelmät soveltuvat hyvin nimenomaan laajan mittakaavan Web 2.0 -sovellusten käyttöön, kuten erityisesti sarakeperhevarastot. Näiden järjestel-

mien tietokannan tietomalli voi olla moniulotteinen, järjestetty, harva, pysyvä ja hajautettu. NoSQL-tietokantajärjestelmien tietokannan tiedot voidaan yleensä osittaa vaakasuoraan. Sarakeperhevarastojen tiedot voidaan kuitenkin osittaa myös pystysuoraan sarakkeittain. Sarakeryhmät joihin kohdistuu operaatioita usein samaan aikaan voidaan sijoittaa lähekkäin toisiaan. Näin toisiaan lähekkäin olevia tietoja voidaan hyödyntää tehokkaasti. NoSQL-tietokantajärjestelmät noudattavat usein löyhempää BASE-oikeellisuusmallia, jolloin tietokannan tietojen ei tarvitse olla aina oikeellisia vaan riittää, että tietokannan oikeellisuus saavutetaan jossain myöhemmässä vaiheessa. Tietokannan päivityksien levittämiseen toisinteisiin hyödynnetäänkin usein asynkronista viestintää. Tietokannan transaktioiden ACID-ominaisuudet voidaan kuitenkin usein taata pienemmälle tietokokonaisuudelle, kuten rivikohtaisesti tai jollekin entiteettiryhmälle. NoSQL-tietokantajärjestelmät skaalautuvat lisäämällä raakaa laskentatehoa eli solmujen määrää tarvittaessa. Kuitenkin myös yksittäisten solmujen suorituskykyyn on hyvä kiinnittää huomiota, koska pienemmällä solmujen määrällä säästetään monia kustannuksia. Näissä järjestelmissä on yleensä myös MapReduce-rajapinta laajan mittakaavan analyyttista rinnakkaislaskentaa varten. MapReduce ei kuitenkaan ole riippuvainen mistään tietokantajärjestelmästä, joten sitä voidaan hyödyntää myös relaatiotietokantajärjestelmissä.

NoSQL-paradigmaa voidaan hyödyntää palveluna. NoSQL-tietokantapalveluna näyttää soveltuvan hyvin ainakin pienimuotoiseen, väliaikaiseen ja ei-kriittiseen käyttöön. NoSQL-tietokantapalveluna voi olla hyvä ratkaisu myös aloittavalle yritykselle. Jos yrityksen web-sovelluksen liikenne kasvaa, NoSQL-tietokantapalvelu skaalautuu tarpeen mukaan. Tarvittaessa yritys voi siirtyä palvelun maksulliseen käyttöön, jolloin lisäresursseista maksetaan tarpeen mukaan. Yrityksen olemassa olevan sovelluksen siirtäminen pilvipalveluun ei kuitenkaan ole välttämättä kovin yksinkertaista. Yritys voi esimerkiksi joutua tekemään sovellukseen muutoksia. Pilvipalvelujen käyttöön liittyy myös monenlaisia rajoituksia. Esimerkiksi pilvipalvelujen resursseja rajoitetaan. Nämä rajoitukset saattavat vaikuttaa siihen onko yrityksen mahdollista edes harkita jotain pilvipalvelua. Näiden seikkojen lisäksi tulee huomioida esimerkiksi palvelun tietoturva, lainsäädäntö, hinnoittelumalli sekä palvelutasosopimus. Yritys voi jatkossa myös olla sidottu pitkäksi aikaa kaupalliseen järjestelmään.

NoSQL-tietokantajärjestelmissä ei ole useita perinteisten relaatiotietokantajärjestelmien valmiita pitkälle kehitettyjä ominaisuuksia. Tietokannan skeemaa ei välttämättä ole ja

tietokannan indeksointitoiminnot ovat rajoittuneita. SQL-kyselykielen liitos- ja kooste-operaatioita ei voida toteuttaa helposti. NoSQL-tietokantajärjestelmien ohjelmakoodi ei välttämättä ole vielä riittävän kehittyneellä tasolla ja tietokantatason tietoturvaominaisuudet saattavat olla puutteellisia. Näissä järjestelmissä ei myöskään ole kaikkia relaatiotietokantajärjestelmien lisäominaisuuksia ja työkaluja. NoSQL-tietokantajärjestelmien ylläpito saattaa lisäksi olla hankalaa. Suurin osa edellä mainituista ominaisuuksista pitää toteuttaa sovelluslogiikassa, joten ne jäävät sovellusohjelmoijan vastuulle. Sovellusohjelmoijan on myös usein otettava kantaa tietokannan fyysiseen toteutukseen, kuten hajautetun tietokannan tietojen fyysiseen sijoitteluun. Tietokannan käsittelyyn taas tarjotaan yleensä joku alemman tason ohjelmointikieli, josta puuttuu SQL-kyselykielen deklaratiivinen ilmaisukyky.

NoSQL-tietokantajärjestelmien käyttöä perusteellaan usein sillä, ettei kaikkia raskaiden relaatiotietokantajärjestelmien ominaisuuksia edes tarvita. Näiden ominaisuuksien puute tekee järjestelmästä joustavan ja yleisluontoisen. Tutkielmasta kuitenkin ilmenee, että edellä kuvatuille ominaisuuksille on näissäkin järjestelmissä jossain määrin edelleen tarvetta. Joissakin NoSQL-tietokantajärjestelmissä voidaan esimerkiksi hyödyntää tietokannan skeemaa. Joihinkin järjestelmiin on kehitetty mahdollisuus käyttää tietokannan toissijaisia indeksejä. Näille järjestelmille on myös kehitetty korkeamman tason ohjelmointi- ja kyselykieliä. Tietokantatason tietoturvaominaisuuksille saattaa lisäksi olla tarvetta. MapReduce-kehystä hyödyntävien järjestelmien ja relaatiotietokantajärjestelmien ohjelmointirajapintojen ominaisuudet lähentyvät myös toisiaan. MapReduce-toteutuksiin on kehitetty uusia rajapintoja korkeamman tason ohjelmointikielille. Joissakin multimallitietokannoissa voidaan hyödyntää sekä MapReducea että SQL-kyselykieltä.

Merkittävät yritykset käyttävät edelleen relaatiotietokantajärjestelmiä ydintoimintojensa taustalla. NoSQL-tietokantajärjestelmät saattavat olla vielä täysin tuntemattomia yrityksille. Yritykset eivät lisäksi halua siirtyä käyttämään NoSQL-tietokantajärjestelmiä, koska tietokannan laatua ei voida täysin taata tietokannan transaktioiden ACID-ominaisuuksilla. ACID-ominaisuuksista ei yleensä voida tinkiä kriittisten sovellusten kohdalla, koska niiden tietojen tulee olla mahdollisimman oikeellisia sekä pysyviä. Eri NoSQL-tietokantajärjestelmien tietokannan tietojen oikeellisuuden saavuttamisen taso saattaa kuitenkin vaihdella. Sovellusohjelmoijalla voi olla myös mahdollisuus vaikuttaa näiden

tietojen oikeellisuustasoon. Jos sovelluksen tietojen ACID-ominaisuuksista on mahdollista tinkiä, yrityksen pitää selvittää sovellusalueen edellyttämä tietojen oikeellisuuden taso.

Mikään tietokantajärjestelmä tai työkalu ei ole paras kaikkiin tehtäviin. Kussakin järjestelmässä on järkevää ja tehokasta käsitellä ja varastoida pääosin tietyn kaltaista sovellusalueen tietoa. Sama koskee myös tietokannan yhteydessä käytettäviä työkaluja, kuten MapReduce-kehystä. MapReduce ja relaatiotietokantajärjestelmät eivät ole kumpikaan hyviä siinä mitä toinen osaa hyvin. MapReduce sopii esimerkiksi ETL-tehtäviin sekä laajan mittakaavan monimutkaiseen analyyttiseen rinnakkaislaskentaan. MapReduce myös skaalautuu kestävällä tavalla laajan mittakaavan ympäristössä. Relaatiotietokantajärjestelmien pitkälle kehitetyt tietokannan kyselyominaisuudet ovat taas esimerkiksi intuitiivisia ja hyvin tehokkaita. Soveltuva tietokantajärjestelmä ja työkalu riippuu täysin yrityksen ja sovelluksen vaatimuksista. Yrityksen tulee arvioida näitä sovellusalueen vaatimuksia. Tietokantajärjestelmän valintaan vaikuttavat esimerkiksi sovelluksen tietokantaoperaatioiden määrä ja tyyppi, sovellusalueen tietojen rakenne ja rakenteen vaatima tietokannan joustavuus sekä järjestelmältä edellytettävä skaalautuvuus, saatavuus ja tehokkuus. Tietokantajärjestelmän luotettavuudella, kestävyydellä ja tietoturvalla saattaa olla suuri merkitys. Näiden lisäksi yrityksen tulee huomioida tietokantajärjestelmän käytettävyys ja ylläpidettävyys.

Jos kaikkiin vaatimuksiin sopivaa yhtä tietokantajärjestelmää ei ole, voidaan hyödyntää monikielistä pysyvyyttä. Näin voidaan tehdä esimerkiksi käyttämällä ratkaisua, jossa tietolähteeseen operoiva logiikka erotetaan business-logiikasta. Eräs vaihtoehto on käyttää yhtä keskitettyä päättävää varastojärjestelmää sekä useampaa erilaista edustavarastona toimivaa tietokantajärjestelmää, jotka kommunikoivat päättävät järjestelmän kanssa. Tuloksena saattaa kuitenkin olla hyvin monimutkainen kokonaisjärjestelmä.

Lähteet

10g12 10gen, Indexing Overview - MongoDB Manual. http://docs.mongodb.org/manual/core/indexes. [5.11.2012]

Aba07 Abadi, Daniel J., Column-Stores For Wide and Sparse Data. Third Biennial Conference on Innovative Data Systems Research (CIDR '07), Asilomar, California, Yhdysvallat, sivut 292-297.

Aba10 Abadi, Daniel, The problems with ACID, and how to fix them without going NoSQL. 31.8.2010. http://dbmsmusings.blogspot.com/2010/08/problems-with-acid-and-how-to-fix-them.html. [20.11.2011]

ABK09 Abouzeid, Azza, Bajda-Pawlikowski, Kamil, Abadi, Daniel, Silberschatz, Avi, Rasin, Alexander, HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. Proceedings of the VLDB Endowment, 2, 1 (August 2009), sivut 922-933.

Ama12a Amazon, Amazon.com: Online Shopping for Electronics, Apparel, Computers, Books, DVDs & more. http://www.amazon.com. [26.3.2012]

Ama12b Amazon, Amazon Web Services LLC, AWS Products & Solutions, Amazon EC2 Details, Amazon EC2 FAQs. http://aws.amazon.com/ec2/faqs. [1.11.2012].

Ama12c Amazon, Amazon Web Services LLC, AWS Products & Solutions, Amazon EC2 Details, Amazon EC2 Service Level Agreement. http://aws.amazon.com/ec2-sla. [1.11.2012].

Ama12d Amazon, Amazon Web Services LLC, AWS Products & Solutions, Amazon EC2 Details, Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/#pricing. [1.11.2012].

Ana08 Anand, Ajay, Scaling Hadoop to 4000 nodes at Yahoo! - Yahoo! Hadoop Blog. 30.9.2008. http://developer.yahoo.com/blogs/hadoop/posts/2008/09/scaling_hadoop_to_4000_nodes_a. [15.11.2012]

AnT10 Anderson, Eric, Tucek, Joseph, Efficiency matters! ACM SIGOPS Operating Systems Review, 44, 1 (2010), sivut 40-45.

BaF95 Baru, Chaitanya, Fecteau, Gilles, An overview of DB2 parallel edition. Proceedings of the 1995 ACM SIGMOD international conference on Management of data (SIGMOD ’95), San Jose, California, Yhdysvallat, 1995, sivut 460-462.

BBC11 Baker, Jason, Bond, Chris, Corbett, James C., Furman, JJ, Khorlin, Andrey, Larson, James, Le´on, Jean-Michel, Li, Yawei, Lloyd, Alexander, Yushprakh, Megastore: Providing Scalable, Highly Available Storage for Interactive Services. 5th Biennial Conference on Innovative Data Systems Research (CIDR ‘11), Asilomar, California, Yhdysvallat, 2011, sivut 223-234.

BDH03 Barroso, Andre, Luiz, Dean, Jeffrey, Hölzle, Urs, Web search for a planet: the Google cluster architecture. IEEE Micro, 23, 2 (2003), sivut 22-28.

BeT11 Bermbach, David, Tai, Stefan, Eventual consistency: How soon is eventual? An evaluation of Amazon S3's consistency behavior. Proceedings of the 6th Workshop on Middleware for Service Oriented Computing (MW4SOC '11), Lissabon, Portugali, 2011, sivut 1:1-1:6.

BGS11 Borthakur, Dhruba, Gray, Jonathan, Sarma, Joydeep Sen, Muthukkaruppan, Kannan, Spiegelberg, Nicolas, Kuang, Hairong, Ranganathan, Karthik, Molkov, Dmytro, Menon, Aravind, Rash, Samuel, Schmidt, Rodrigo, Aiyer, Amitanand, Apache hadoop goes realtime at Facebook. Proceedings of the 2011 ACM SIGMOD International Conference on Management of data (SIGMOD ’11), Ateena, Kreikka, 2011, sivut 1071-1080.

BLS11 Bonnet, Laurent, Laurent, Anne, Sala, Michel, Laurent, Benedicte, Sicard, Nicolas, Reduce, You Say: What NoSQL Can Do for Data Aggregation and BI in Large Repositories. 22nd International Workshop on Database and Expert Systems Applications (DEXA ‘11), Toulouse, Ranska, 2011, sivut 483-488.

Bre00 Brewer, Eric, Towards robust distributed systems. Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing (PODC ’00), Portland, Oregon, Yhdysvallat, 2000, sivu 7. [Myös: http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf].

CAB81 Chamberlin, Donald D., Astrahan, Morton M., Blasgen, Michael W., Gray,

James N., King, W. Frank, Lindsay, Bruce G., Lorie, Raymond, Mehl, James W., Price, Thomas G., Putzolu, Franco, Selinger, Patricia Griffiths, Schkolnick, Mario, Slutz, Donald R., Traiger, Irving L., Wade, Bradford W., Yost, Robert A, A history and evaluation of System R. Communications of the ACM, 24, 10 (October 1981), sivut 632-646.

Cat10 Cattell, Rick, Scalable SQL and NoSQL data stores. ACM SIGMOD Record, 39, 4 (December 2010), sivut 12-27.

CCF08 Cafarella, Michael, Chang, Edwad, Fikes, Andrew, Halevy, Alon, Hsieh, Wilson, Lerner, Alberto, Madhavan, Jayant, Muthukrishnan, S., Data management projects at Google. ACM SIGMOD Record, 37, 1 (2008), sivut 34-38.

CDG06 Chang, Fay, Dean, Jeffrey, Ghemawat, Sanjay, Hsieh, Wilson, Wallach, Deborah A., Burrows, Michael, Chandra, Tushar, Fikes, Andrew, Gruber, Robert, Bigtable: a distributed storage system for structured data. Proceedings of the 7th symposium on Operating Systems Design and Implementation (OSDI'06), Seattle, Yhdysvallat, 2006, sivut 205-218.

CDG08 Chang, Fay, Dean, Jeffrey, Ghemawat, Sanjay, Hsieh, Wilson, Wallach, Deborah A., Burrows, Michael, Chandra, Tushar, Fikes, Andrew, Gruber, Robert, Bigtable: a distributed storage system for structured data. ACM Transactions on Computer Systems, 26, 2 (2008), sivut 4:1-4:26.

CGR07 Chandra, Tushar D., Griesemer, Robert, Redstone, Joshua, Paxos made live: an engineering perspective. Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing (PODC ’07), Portland, Oregon, Yhdysvallat, 2007, sivut 398-407.

CHU08 Chu-Carroll, Mark, Databases are hammers; MapReduce is a screwdriver. Good Math, Bad Math. 22.1.2008. http://scienceblogs.com/goodmath/2008/01/22/databases-are-hammers-mapreduc/. [1.10.2012]

CJL08 Chaiken, Ronnie, Jenkins, Bob, Larson, Per-Åke, Ramsey, Bill, Shakib, Darren, Weaver, Simon, Zhou, Jingren, SCOPE: easy and efficient parallel processing of massive data sets. Proceedings of the VLDB Endowment, 1, 2 (august 2008), sivut 1265-1276.

Cod12 CodeFutures Corporation, Database Sharding. http://www.codefutures.com/database-sharding. [21.11.2012]

CSD11 Cuzzocrea, Alfredo, Song, Il-Yeol, Davis, Karen C., Analytics over large-scale multidimensional data: the big data revolution! Proceedings of the ACM 14th international workshop on Data Warehousing and OLAP (DOLAP ’11), Glasgow, Iso-Britannia, 2011, sivut 101-104.

DeG08 Dean, Jeffrey, Ghemawat, Sanjay, MapReduce: simplified data processing on large clusters. Communications of the ACM - 50th anniversary issue: 1958 - 2008, 51, 1 (2008), sivut 107-113.

DeG10 Dean, Jeffrey, Ghemawat, Sanjay, MapReduce: a flexible data processing tool. Communications of the ACM - Amir Pnueli: Ahead of His Time, 53, 1 (2010), sivut 72-77.

DHJ07 DeCandia, Giuseppe, Hastorun, Deniz, Jampani, Madan, Kakulapati, Gunavardhan, Lakshman, Avinash, Pilchin, Alex, Sivasubramanian, Swaminathan, Vosshall, Peter, Vogels, Werner, Dynamo: amazon's highly available key-value store. Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles (SOSP ’07), Stevenson, Washington, Yhdysvallat, 2007, sivut 205-220.

DeS08 DeWitt, David J., Stonebraker, Michael, MapReduce: A major step backwards. The Database Column. 17.1. 2008. http://databasecolumn.vertica.com/2008/01/mapreduce_a_major_step_back.html. [Myös: http://homes.cs.washington.edu/~billhowe/mapreduce_a_major_step_backwards.html]. [1.10.2012]

Dor12 Dormando, memcached - a distributed memory object caching system. http://memcached.org. [15.11.2012]

Eba12 Ebay Inc., Electronics, Cars, Fashion, Collectibles, Coupons and More Online Shopping eBay. http://www.ebay.com. [16.11.2012]

EDL12 Edlich, Stefan, NoSQL, your ultimate guide to the non - relational universe! http://nosql-database.org. [24.3.2012]

Eva09 Evans, Eric A., NOSQL 2009. 12.5.2009. http://blog.sym-link.com/2009/05/

12/nosql_2009.html. [4.3.2012]

Fac12 Facebook Inc., Facebook. http://fi-fi.facebook.com. [26.3.2012]

FoS12 Fowler, Martin, Sadalage, Pramod, Polyglot Persistence. http://martin fowler.com/articles/nosql-intro.pdf. [2.11.2012]

FuW11 Fuller, Alfred, Wilder, Matt, More 9s Please: Under The Covers of the High Replication Datastore. Google I/O developer conference (Google I/O '11), San Francisco, Yhdysvallat, 2011. [Myös: http://www.google.com/events/ io/2011/sessions/more-9s-please-under-the-covers-of-the-high-replication-d atastore.html].

GGL03 Ghemawat, Sanjay, Gobioff, Howard, Leung, Shun-Tak, The Google file system. Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP '03), New York, Yhdysvallat, 2003, sivut 29-43.

Gho10 Ghosh, Debasish, Multiparadigm Data Storage for Enterprise Applications. IEEE Software 27, 5 (September - October 2010), sivut 57-60.

Git12 GitHub Inc., Home facebook/scribe Wiki GitHub. https://github.com/face book/scribe/wiki. [8.11.2012]

Goo12a Google, Google Accounts. https://accounts.google.com. [6.11.2012]

Goo12b Google, Google Developers, Google App Engine. https://developers.google. com/appengine/. [1.11.2012]

Goo12c Google, Google Developers, Google App Engine, App Engine Service Level Agreement. https://developers.google.com/appengine/sla. [1.11.2012]

Goo12d Google, Google Developers, Google App Engine, GQL Reference. https:// developers.google.com/appengine/docs/python/datastore/gqlreference. [9.11.2012]

Goo12e Google, Google Developers, Google App Engine, MapReduce Overview. https://developers.google.com/appengine/docs/python/dataprocessing/overvi ew?hl=en. [1.11.2012]

Goo12f Google, Google Developers, Google App Engine, Migrating to the High Replication Datastore. https://developers.google.com/appengine/docs/admin

console/migration. [1.11.2012]

Goo12g Google, Google Developers, Google App Engine, Standard Mapreduce Input Readers and Output Writers. https://developers.google.com/appengine/docs/python/dataprocessing/readers_writers. [30.11.2012]

Goo12h Google, Google Developers, Google App Engine, What Is Google App Engine? https://developers.google.com/appengine/docs/whatisgoogleappengine?hl=en. [6.11.2012]

Goo12i Google, Google earth. http://www.google.com/intl/fi/earth/index.html. [26.3.2012]

Goo12j Google, Google Gmail. https://mail.google.com. [6.11.2012]

Goo12k Google, Google Project Hosting, protobuf - Protocol Buffers - Protocol Buffers - Google's data interchange format. http://code.google.com/p/protobuf. [6.11.2012]

Goo12l Google, Google Suomi. http://www.google.fi. [26.3.2012]

Goo12m Google, YouTube - Broadcast Yourself. http://www.youtube.com. [26.3.2012]

Goo12n Google, Google+: reaaliaikaisen sisällön jakamisen vallankumous verkossa. https://plus.google.com. [26.3.2012]

HeJ11 Hecht, Robin, Jablonski, Stefan, NoSQL evaluation: A use case oriented survey. International Conference on Cloud and Service Computing (CSC '11), Hong Kong, Kiina, 2011, sivut 336-341.

HEL11 Han, Jing, E., Haihong, Le, Guan, Du, Jian, Survey on NoSQL database. 6th International Conference on Pervasive Computing and Applications (ICPCA '11), Port Elizabeth, Etelä-Afrikka, 2011, sivut 363-366.

HoP10 Hollingsworth, Joel, Powell, David J., Teaching web programming using the Google Cloud. Proceedings of the 48th Annual Southeast Regional Conference (ACM SE '10), Oxford, Mississippi, Yhdysvallat, 2010, sivut 76:1--76:5.

HOF09 Hoff, Todd, A Yes For A NoSQL Taxonomy. http://highscalability.com/

blog/2009/11/5/a-yes-for-a-nosql-taxonomy.html. 5.11.2009. [24.3.2012]

Ibm12a IBM, IBM - DB2 database software. http://www-01.ibm.com/software/data/db2. [16.11.2012]

Ibm12b IBM, IBM - United States. http://www.ibm.com/us/en. [16.11.2012]

IBY07 Isard, Michael, Budiu, Mihai, Yu, Yuan, Birrell, Andrew, Fetterly, Dennis, Dryad: distributed data-parallel programs from sequential building blocks. Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (EuroSys ’07), Lissabon, Portugali, 2007, sivut 59-72.

IMS10 Ishizaki, Kazuaki, Mizuno, Ken, Suganuma, Toshio, Silva, Daniel, Koseki, Akira, Komatsu, Hideaki, Ueda, Yohei, Nakatani, Toshio, Parallel programming framework for large batch transaction processing on scale-out systems. Proceedings of the 3rd Annual Haifa Experimental Systems Conference (SYSTOR ’10), Haifa, Israel, 2010, sivut 15:1-15:14.

Inm03 Inmon, Bill, The System of Record in the Global Data Warehouse. Information Management Magazine, (May 2003). [Myös: http://www.information-management.com/issues/20030501/6645-1.html].

Kat12 Katsov, Ilya, NoSQL Data Modeling Techniques. 1.3.2012. http://highlyscalable.wordpress.com/2012/03/01/nosql-data-modeling-techniques. [17.12.2012]

Kim11 Kim, Won, Cloud architecture: a preliminary look. Proceedings of the 9th International Conference on Advances in Mobile Computing and Multimedia, Ho Chi Minh City, Vietnam, 2011, sivut 2-6.

Lam98 Lamport, Leslie, The part-time parliament. ACM Transactions on Computer Systems, 16, 2 (May 1998), sivut 133-169.

LaM10 Lakshman, Avinash, Malik, Prashant, Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review, 44, 2 (April 2010), sivut 35-40.

Lea10 Leavitt, Neal, Will NoSQL databases live up to their promise? Computer, 43, 2 (February 2010), sivut 12-14.

Leb08 Leberknight, Scott, Polyglot Persistence. http://nearinfinity.com/blogs/scott _leberknight/polyglot_persistence.html. [2.11.2012]

Lee10 Lee, Craig A., A perspective on scientific cloud computing. Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC '10), Chicago, Illinois, Yhdysvallat, 2010, sivut 451-459.

Ler10 Lerner, Reuven M., At the forge: NoSQL? I'd prefer SomeSQL. Linux Journal, 2010, 192 (April 2010), artikkeli nro 5.

LLC12 Lee, Kyong-Ha, Lee, Yoon-Joon, Choi, Hyunsik, Chung, Yon Dohn, Moon, Bongki, Parallel data processing with MapReduce: a survey. ACM SIGMOD Record, 40, 4 (December 2011), sivut 11-20.

Mar12 Fowler, Martin, Data Mapper. http://martinfowler.com/eaaCatalog/dataMap per.html. [31.10.2012]

MeB11 Meijer, Erik, Bierman, Gavin, A co-relational model of data for large shared data banks, Communications of the ACM, 54, 4 (April 2011), sivut 49-58.

MKW11 Malawski, Maciej, Kuzniar, Maciej, Wojcik, Piotr, Bubak, Marian, How to Use Google App Engine for Free Computing. Internet Computing, IEEE, PP, 99 (2011), sivu 1.

Mic09 Microsoft, Case Studies, Microsoft Case Study: Microsoft SQL Server 2005 Enteprise X64 Edition - MySpace. 17.6.2009. http://www.microsoft.com/ca sestudies/Microsoft-SQL-Server-2005-Enterprise-X64-Edition/MySpace/M ySpace-Uses-SQL-Server-Service-Broker-to-Protect-Integrity-of-1-Petabyte -of-Data/4000004532. [8.11.2012]

Mic12a Microsoft, Business Intelligence Database Management Data Warehousing Microsoft SQL Server. http://www.microsoft.com/sqlserver/en/us/default. aspx. [16.11.2012]

Mic12b Microsoft Corporation, Microsoft Access 2012 - Office.com. http://office. microsoft.com/fi-fi/access. [16.11.2012]

Mic12c Microsoft, Microsoft Corporation: Software, Smartphones, Online, Games, Cloud Computing, IT Business Technology, Downloads. http://www.micro

soft.com/fi-fi/default.aspx. [16.11.2012]

Mic12d Microsoft, MSDN Library, Chapter 8: Data Layer Guidelines. http://msdn.microsoft.com/en-us/library/ee658127.aspx. [31.10.2012]

Mic12e Microsoft, MSDN Library, The Repository Pattern. http://msdn.microsoft.com/en-us/library/ff649690.aspx. [31.10.2012]

Mur09 Murray, Paul, Enterprise grade cloud computing. Proceedings of the Third Workshop on Dependable Distributed Data Management (WDDM '09), Nürnberg, Saksa, 2009, sivu 1.

Mur07 Murugesan, San. Understanding Web 2.0. IT Professional, 9, 4 (July-August 2007), sivut 34-41.

Mys12 Myspace LLC, MySpace.com. http://www.myspace.com. [8.11.2012]

NOS11 NoSQL Now! 2011, San Jose, Yhdysvallat, 2011. http://nosql2011.wilshireconferences.com. [24.3.2012]

OGG11 Okman, Lior, Gal-Oz, Nurit, Gonen, Yaron, Gudes, Ehud, Abramov, Jenny, Security Issues in NoSQL Databases. Proceedings of the 2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications (TRUSTCOM '11), Washington, DC, Yhdysvallat, 2011, sivut 541-547.

Ora12a Oracle Corporation, Database 11g Oracle Database 11g Oracle. http://www.oracle.com/us/products/database/overview/index.html. [16.11.2012]

Ora12b Oracle Corporation, Oracle Hardware and Sofware, Engineered to Work Together. http://www.oracle.com/index.html. [16.11.2012]

Ora12c Oracle Corporation, Oracle real application clusters. http://www.oracle.com/us/products/database/options/real-application-clusters/index.html. [26.3.2012]

Ora12d Oracle Corporation, MySQL: MySQL Cluster CGE. http://www.mysql.com/products/cluster. [16.11.2012]

Ora12e Oracle Corporation, MySQL: The world's most popular open source database. http://www.mysql.com. [26.3.2012]

ORS08 Olston, Christopher, Reed, Benjamin, Srivastava, Utkarsh, Kumar, Ravi, Tomkins, Andrew, Pig latin: a not-so-foreign language for data processing. Proceedings of the 2008 ACM SIGMOD international conference on Management of data (SIGMOD ’08), Vancouver, Kanada, 2008, sivut 1099-1110.

OSG10 Ordonez, Carlos, Song, Il-Yeol ,Garcia-Alvarado, Carlos, Relational versus non-relational database systems for data warehousing. Proceedings of the ACM 13th international workshop on Data warehousing and OLAP (DOLAP ’10), Toronto, Ontario, Canada, 2010, sivut 67-68.

Pat99 Paterno, Giuseppe, NoSQL Tutorial: A comprehensive look at the NoSQL database. Linux Journal, 1999, 67, (November 1999), artikkeli nro 23.

PDG05 Pike, Rob, Dorward, Sean, Griesemer, Robert, Quinlan, Sean, Interpreting the Data: Parallel Analysis with Sawzall. Scientific Programming Journal, 13, 4 (2005), sivut 227-298.

PGR12 Pollack, Mark, Gierke, Oliver, Risberg, Thomas, Brisbin, Jon, Hunger, Michael, *Spring Data, Modern Data Access for Enterprise Java*. O’Reilly Media, 2012.

Pok11 Pokorny, Jaroslav, NoSQL databases: a step to database scalability in web environment. Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services (iiWAS ’11), Ho Chi Minh City, Vietnam, 2011, sivut 278-283.

Pop12 Popescu, Alex, Comparing Document Databases to Key-Value Stores. 3.6.2012.http://nosql.mypopescu.com/post/659390374/comparing-document -databases-to-key-value-stores. [16.12.2012]

Pos12 PostgreSQL Global Development Group, PostgreSQL. http://www.postgre sql.org. [26.3.2012]

PPR09 Pavlo, Andrew, Paulson, Erik, Rasin, Alexander, Abadi, Daniel J., DeWitt, David J., Madden, Samuel, Stonebraker, Michael, A Comparison of Approaches to Large-Scale Data Analysis. Proceedings of the 2009 ACM SIGMOD International Conference on Management of data (SIGMOD ’09), Providence, Rhode Island, Yhdysvallat, 2009, sivut 165-178.

Pri08 Pritchett, Dan, BASE: An acid alternative. Queue - Object-Relational Mapping, 6, 3 (May 2008), sivut 48-55.

Rah93 Rahm, Erhard, Parallel query processing in shared disk database systems. ACM SIGMOD Record, 22, 4 (December 1993), sivut 32-37.

Rod12 Rodriguez, Marko A., Graph Pattern Matching with Gremlin 1.1. http://mar korodriguez.com/2011/06/15/graph-pattern-matching-with-gremlin-1-1. [5.1 .2012]

Rys11 Rys, Michael, Scalable SQL. Communications of the ACM, 54, 6 (June 2011), sivut 48-53.

SAD10 Stonebraker, Michael, Abadi, Daniel, DeWitt, David J., Madden, Sam, Paulson, Erik, Pavlo, Andrew, Rasin, Alexander, MapReduce and parallel DBMSs: friends or foes? Communications of the ACM - Amir Pnueli: Ahead of His Time, 53, 1 (January 2010), sivut 64-71.

SDE11 Seoul Data Engineering Camp (SDEC ’11), Soul, Etelä-Korea, 2011. http:// www.sdec.kr. [24.3.2012]

SKS11 Silberschatz, Abraham, Korth, Henry F., Sudarshan, S., *Database System Concepts*, *Sixth Edition*, International Edition 2011, McGraw-Hill, 2011.

SMA07 Stonebraker, Michael, Madden, Samuel, Abadi, Daniel J., Harizopoulos, Stavros, Hachem, Nabil, Helland, Pat, The end of an architectural era: (it's time for a complete rewrite). Proceedings of the 33rd international conference on Very large data bases (VLDB ’07), Wien, Itävalta, 2007, sivut 1150-1160.

Sob07 Sobel, Jason, Keeping up. 21.12.2007. http://blog.facebook.com/blog.php? post=7899307130. [8.11.2012]

Spr12a SpringSource, a division of VMware, springsource community. http://www. springsource.org. [31.10.2012]

Spr12b SpringSource, a division of VMware, Spring Projects SPRING DATA. http: //www.springsource.org/spring-data. [31.10.2012]

StC11 Stonebraker, Michael, Cattell, Rick, 10 rules for scalable performance in 'simple operation' datastores. Communications of the ACM, 54, 6 (June

2011), sivut 72-80.

Sto10 Stonebraker, Michael, In search of database consistency. Communications of the ACM, 53, 10 (October 2010), sivut 8-9.

Sto11 Stonebraker, Michael, Stonebraker on NoSQL and enterprises. Communications of the ACM, 54, 8 (August 2011), sivut 10-11.

Str12 Strozzi, Carlo, NoSQL A Relational Database Management System. http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql/Home%20Page. [4.3.2012]

The12a The Apache Software Foundation, Apache CouchDB. http://couchdb.apache.org. [3.12.2012]

The12b The Apache Software Foundation, Apache Hadoop. http://hadoop.apache.org. [13.8.2012]

The12c The Apache Software Foundation, Apache HBase. http://hbase.apache.org. [15.11.2012]

The12d The Apache Software Foundation, Apache Lucenen - Welcome to Apache Lucene. http://lucene.apache.org. [16.11.2012]

The12e The Apache Software Foundation, Cassandra. http://cassandra.apache.org. [15.11.2012]

The12f The Apache Software Foundation, Struts. http://struts.apache.org. [6.11.2012]

The12g The Apache Software Foundation, Welcome to Apache Pig! http://pig.apache.org. [8.11.2012]

The12h The Apache Software Foundation, Welcome to Hive! http://hive.apache.org. [9.11.2012]

Tin12 TeinkerPop, Gremlin. http://github.com/tinkerpop/gremlin. [5.11.2012]

Tiw11 Tiwari, Shashank, *Professional NoSQL*, Wiley, 2011.

TSJ09 Thusoo, Ashish, Sarma, Joydeep Sen, Jain, Namit, Shao, Zheng, Chakka, Prasad, Anthony, Suresh, Liu, Hao, Wyckoff, Pete, Murthy, Raghotham, Hive: a warehousing solution over a map-reduce framework. Proceedings of the VLDB Endowment, 2, 2 (August 2009), sivut 1626-1629.

TuB11 Tudorica, Bodgan George, Bucur, Cristian, A comparison between several NoSQL databases with comments and notes. Roedunet International Conference (RoEduNet '11), Iasi, Romania, 2011, sivut 1-5.

Twi10 Twitter Inc., Engineering Blog, Cassandra at Twitter Today. 9.7.2010. http://engineering.twitter.com/2010/07/cassandra-at-twitter-today.html. [8.11.2012]

Twi11 Twitter Inc., Engineering Blog, Introducing FlockDB. http://engineering.twitter.com/2010/05/introducing-flockdb.html. [15.11.2012]

Twi12 Twitter Inc, Twitter. http://twitter.com. [8.11.2012]

TYP08 Typical Programmer, Relational Database Experts Jump The MapReduce Shark. 17.1.2008. http://typicalprogrammer.com/?p=16. [1.10.2012]

Vaj09 Vajgel, Peter, Needle in a haystack: efficient storage of billions of photos. http://www.facebook.com/note.php?note_id=76191543919. [18.3.2012]

Ver12 Vertica, Real-Time Analytics Platform. http://www.vertica.com. [16.11.2012]

VMZ10 Vicknair, Chad, Macias, Michael, Zhao, Zhendong, Nan, Xiaofei, Chen, Yixin, Wilkins, Dawn, A comparison of a graph database and a relational database: a data provenance perspective. Proceedings of the 48th Annual Southeast Regional Conference (ACM SE '10), Oxford, Yhdysvallat, 2010, artikkeli nro 42.

Vog12 Vogels, Werner, Amazon DynamoDB - a Fast and Scalable NoSQL Database Service Designed for Internet Scale Applications. http://www.allthingsdistributed.com/2012/01/amazon-dynamodb.html. [13.12.2012]

W3C12 W3C, SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query. [05.11.2012]

Wei10 Weil, Kevin, NoSQL at Twitter. Strange Loop Conference (Strange Loop '10), St. Louis, Missouri, Yhdysvallat, 2010. [Myös: http://strangeloop2010.com/talks/14446]

WIK12a Wikipedia, NoSQL. http://en.wikipedia.org/wiki/NoSQL. [24.3.2012]

WIK12b Wikipedia, Semi-structured model. http://en.wikipedia.org/wiki/Semi-structured_model. [12.11.2012]

Wil12 Williams, Paul, UnQL: A Standardized Query Language for NoSQL Databases. http://www.dataversity.net/unql-a-standardized-query-language-for-nosql-databases. [5.11.2012]

WMH11 Wei-ping, Zhu, Ming-xin, Li, Huan, Chen, Using MongoDB to implement textbook management system instead of MySQL. IEEE 3rd International Conference on Communication Software and Networks (ICCSN '11), Xi'an, Kiina, 2011, sivut 303-305.

Yah12 Yahoo! Inc., Welcome to Flickr - photo sharing. http://www.flickr.com. [26.3.2012]

Yah12b Yahoo! Inc., Yahoo! http://www.yahoo.com. [15.11.2012]

YEN09 Yen, Stephen, NoSQL is a horseless carriage. http://dl.getdropbox.com/u/2075876/nosql-steve-yen.pdf. [24.3.2012]