Tampereen yliopisto

Simo Kivistö

HAJAUTETUT RELAATIOTIETOKANNAT JA NIIDEN SKAALAUTUVUUS

Informaatioteknologian ja viestinnän tiedekunta
Pro gradu -tutkielma
Ohjaaja: Marko Junkkari
Maaliskuu 2019

TIIVISTELMÄ

Simo Kivistö: Hajautetut relaatiotietokannat ja niiden skaalautuvuus
Pro gradu -tutkielma, 73 sivua
Tampereen yliopisto
Tietojenkäsittelyoppi
Maaliskuu 2019

NoSQL-tietokannat ovat nousseet 2000-luvulla perinteisten relaatiotietokantojen rinnalle esimerkiksi verkkosovellusten palvelintietokannoiksi. Näillä uusilla ratkaisuilla on pyritty vastaamaan muun muassa skaalautuvuuden tarpeisiin. Monissa järjestelmissä kuitenkin vaaditaan tietokannalta ominaisuuksia, jotka puuttuvat NoSQL-järjestelmistä. Tässä tutkielmassa selvitetään olemassa olevia tapoja hajauttaa relaatiotietokanta useasta palvelimesta koostuvaksi järjestelmäksi ja keinoja, joilla palvelimien määrää vaihtamalla voidaan sopeutua vaihtuneeseen työkuormaan tai datan määrään.

Yleisten kaupallisten järjestelmien ominaisuudet havaitaan vielä rajoittuneiksi, kun niitä verrataan erilaisissa tutkimuksissa esitettyihin ratkaisuihin. Suurin osa kaupallisista ratkaisuista ei tarjoa automaattista skaalautuvuutta ja käytetyimpien verkkosovellusten palveluntarjoajat ovatkin luoneet tarvittaessa omia, räätälöityjä relaatiotietokantaratkaisujaan.

Hajautus muodostuu tietokannan taulujen kopioinnista tai niiden rakenteen tai sisällön hajauttamisesta hajautetun järjestelmän palvelinten välillä. Useita heuristiikkaan perustuvia ratkaisuja on esitetty lähes optimaalisen hajautuksen hakemiseksi. Myös tietokannan käyttötarkoitus vaikuttaa parhaaseen metodiin. Lisäksi skaalautuvuutta varten on esitetty erilaisia tapoja siirtää dataa tuotantokäytössä olevan järjestelmän palvelinten välillä ilman käyttökatkoa, jotta järjestelmän arkkitehtuuri saataisiin vastaamaan muuttuneen työkuorman tarpeita.

Avainsanat ja -sanonnat: relaatiotietokannat, hajautetut järjestelmät, hajautetut tietokannat

Sisällys

Sisällys i
1. Johdanto 1
2. Tietokantajärjestelmien käsitteistöä 5
2.1. Kyselyjen optimointi 7
2.2. Samanaikaisuuden hallinta 8
2.2.1. Lukot 8
2.2.2. Aikaleimaperustainen rinnakkaisuuden hallinta 9
2.2.3. Optimistinen rinnakkaisuuden hallinta 9
2.3. Tiedon oikeellisuus ja säilyvyys 9
2.4. OLTP- ja OLAP-järjestelmät 10
3. Esimerkkitietokanta 12
4. Rinnakkaisten tietokantajärjestelmien toimintaperiaatteet 15
4.1. Tietokantanoodit ja niiden hallinta 17
4.2. Replikointi 18
4.3. Partitiointi 21
4.4. Kyselyiden käsittely ja ohjaus noodeille 26
4.5. Kyselyjen rinnakkainen ajo 28
4.6. Hajautuksessa käytettyjen tekniikoiden hyödyt 29
5. Olemassa olevia hajautettuja järjestelmiä 30
5.1. Tutkimuksissa käytettyjä tai niihin kehitettyjä järjestelmiä 30
5.2. Kaupalliset tietokantajärjestelmät 32
5.2.1. PostgreSQL 33
5.2.2. MySQL 34
5.2.3. Microsoft SQL Server 37
5.2.4. Muut järjestelmät 38
6. Optimaalinen taulujen hajautus 42
6.1. Hajautus OLTP-järjestelmässä 43
6.2. Uudelleenhajautus OLTP-järjestelmässä 49
6.3. Hajautus OLAP-järjestelmässä 54
6.4. Uudelleenhajautus OLAP-järjestelmässä 58
7. Optimaalinen kyselyjen käsittely 60
8. Johtopäätökset 63
Viiteluettelo 66
Liitteet

1. Johdanto

Internetin ja pilvipalvelujen yleistymisen myötä tiedonhallinnan hajautettu luonne on kasvanut. Verkkosovellukset, joilla saattaa olla miljoonia yhtäaikaisia käyttäjiä, ovat tuoneet uusia haasteita tietokantajärjestelmien suorituskyvylle. Perinteisesti suuren kuormituksen ongelmat on ratkaistu hankkimalla tehokkaampia tietokoneita. Näiden hinta on kuitenkin usein huomattavasti normaaleita palvelinkoneita korkeampia, ja usein suuryritysten konesaleissa käytetäänkin tavallisia palvelimia. Tietokantajärjestelmän teho- tai tilavaatimusten kasvaessa pitää suorituskyvyn kasvattamiseksi tällöin palvelinryhmän päälle rakentaa hajautettu tietokantaratkaisu, joka muodostuu toisistaan erillisistä palvelimista.

Kun tietokanta hajautetaan, saatetaan päätyä käyttämään niin sanottua *NoSQL-järjestelmää*. NoSQL on löyhä termi ratkaisuille, joissa luovutaan ainakin osittain perinteisen tietokannan tarjoamista eheän tiedon varmistavista säännöistä ja vaatimuksesta määritellä järjestelmän tietosisällön rakenne tarkasti etukäteen. Mikäli kyse on esimerkiksi pankkijärjestelmästä, ei näistä tiedon oikeellisuuden vaatimuksista kuitenkaan voida luopua. Säännöillä varmistetaan muun muassa, että kaikki järjestelmään yhteyttä ottavat tahot näkevät järjestelmän sisältävät tiedot samassa tilassa.

Perinteinen *vertikaalinen skaalautuvuus* saavutetaan lisäämällä resursseja (muistia tai prosessoritehoa) olemassa olevaan järjestelmään. Suuremmissa järjestelmissä vertikaalisen skaalautuvuuden ongelmana ovat nopeasti vastaan tulevat tehokkaimpienkin palvelinkoneiden suorituskyvyn rajat, vaikka pilviympäristöt ovatkin helpottaneet palvelintehojen lisäyksen haasteita. Kun skaalautuvuus saavutetaan jakamalla data ja työkuorma useammalle resursseja jakamattomalle palvelimelle käytetään termiä *horisontaalinen skaalautuvuus* [Cattell 2011].

Kriittisissä järjestelmissä pitää palvelun olla jatkuvasti käytettävissä. *Korkean saatavuuden* (high availability) ratkaisuissa on eri tavoin pyritty vastaamaan tähän tarpeeseen. Rinnakkaisen tietokannan tapauksessa tämä saatavuus voidaan toteuttaa siten, että eri palvelimet toimivat toistensa varakoneina. Tällöin yksittäisellä palvelimella tapahtunut häiriö ei vaikuta loppukäyttäjän kokemukseen, koska häiriön sattuessa pyynnöt ohjataan mahdollisimman nopeasti toiselle palvelimelle. Näissä järjestelmissä pyritään välttämään tilannetta, jossa järjestelmän toimivuus olisi yksittäisestä palvelimesta kiinni (niin sanottu single point of failure).

Jotta hajautetun järjestelmän hyödyt saadaan käyttöön pitää työkuorma tasata eri palvelinten kesken (load balancing). Tietokantajärjestelmässä voidaan hajauttaa tietokantaan tallennettu tieto ja siihen tehtävät kyselyt. Mikäli sama tieto löytyy

useammalta palvelimelta, voidaan kysely ohjata palvelimelle, jolla sen hetkinen työkuorma on kevein.

Tieto jaetaan (partitioidaan) palvelinten kesken joko kopioimalla sama data useammalle palvelimelle tai hajauttamalla yhden tai useamman taulun data palvelinten välillä. Partitioinnin avulla voidaan suuria tietokantatauluja käsiteltäessä käyttää hyväksi useamman palvelimen tilakapasiteettia. Lisäksi yhteen palvelimeen kohdistuva kuorma vähenee, kun kyselyt jaetaan kaikkien niiden palvelinten kesken, joilta haettava tieto löytyy. Keskeinen tutkimuskysymys onkin tiedon optimaalinen hajautus järjestelmän suorituskyvyn kannalta.

Tässä tutkielmassa keskitytään tarkastelemaan tietokantojen välistä tiedon kahdennusta ja hajautusta. Jätän pääsääntöisesti aihealueen ulkopuolelle esimerkiksi levytasolla toimivat jaetun levyn ratkaisut (shared disk) ja keskityn ainoastaan *resursseja jakamattomiin* (shared nothing) ratkaisuihin, joissa jokaisella palvelimella on omat resurssinsa (muisti, levy, prosessori) ja niiden välinen kommunikointi tapahtuu TCP/IP-verkossa. Keskityn tietokannan tasolla tapahtuvaan hajautukseen, joten käyttöjärjestelmän tai laitteiston tasolla järjestetyt hajautusratkaisut jäävät tarkastelun ulkopuolelle.

Optimaalinen datan hajautus riippuu olennaisesti tietokantajärjestelmän käyttötarkoituksesta. Reaaliaikaisissa järjestelmissä, joissa dataa tuotetaan, tieto pyritään partitioimaan siten, että mahdollisimman suuri osa kyselyistä pystytään ajamaan yhdellä palvelimella. Mikäli kysely joudutaan hajauttamaan, joudutaan turvautumaan monimutkaisempiin lukitusmalleihin, jotka huolehtivat riittävästä lukitustasosta yhtäaikaisesti eri partitioissa. Mikäli järjestelmä palvelee raportointitarkoituksiin ajettavia monimutkaisia kyselyitä, sisältyy näihin usein monimutkaista laskentaa, jolloin saattaa olla mielekästä pyrkiä ajamaan kysely pienemmissä osasissa usealla palvelimella.

Tietokannan partitioinnin suunnittelu voidaan tehdä manuaalisesti tai se voidaan laskea automaattisesti. Lisäksi jo käytössä olevan järjestelmän hajautusstrategiaa voidaan muuttuneen työkuorman perusteella muuttaa, ja joissakin järjestelmissä tämä onnistuu ilman käyttökatkoa [Elmore et al. 2015]. Järjestelmä voi myös skaalautua automaattisesti, jolloin uusia palvelimia lisätään tarvittaessa ja eniten kuormitetut rivit siirretään niille automaattisesti ilman katkosta tuotannossa.

Ensimmäiset tietokantajärjestelmät syntyivät 1960-luvulla, mutta nämä eivät vielä käyttäneet relaatiomallia, vaan hierarkkista tai verkkomallia [Harrison 2015]. Edgar Codd kehitti relaatiotietokannan mallin vuonna 1970 [Codd 1970] ja tämän perusteella Oracle kehitti oman varhaisen, laitteistosta riippumattoman relaatiotietokantajärjestelmänsä [Lake and Crowther 2013]. Jo 1960-luvulla tietokoneiden välisten verkkojen kehitys mahdollisti myös hajautetut tietokannat [Deppe

and Fry 1976] ja ensimmäiset artikkelit aiheesta löytyvät 70-luvun alusta (katso esimerkiksi Booth [1972]).

Yksi ensimmäisistä hajautetuista järjestelmistä oli 1970-luvun lopulla kehitetty System for Distributed Databases (SDD-1) [Rothnie et al. 1980]. Tuohon aikaan pullonkaulana oli tiedonsiirron nopeus verkossa. Myös alan käsitteistö oli vasta muodostumassa ja esimerkiksi ACID-malli tietokannan tiedon oikeellisuudesta muodostettiin vasta 80-luvussa [Gray 1981]. Deppe ja Fry totesivatkin vuonna 1976, ettei suuntaus resurssien jakamiseen ollut vielä täysin toteutunut siihen aikaan käytössä olleella teknologialla.

Myös varhaisista hajautetuista tietokannoista tuotantokäytössä löytyy esimerkkejä 1970-luvulta: Australian puolustusministeriön järjestelmässä dataa kopioitiin keskustietokoneen tietokannasta segmenteissä eri puolilla Australiaa sijaitsevien tukikohtien minitietokoneille. Synkronointi ei tapahtunut reaaliajassa, vaan ajastetusti useamman kerran päivässä. [Lake and Crowther 2013]

Suurien tietomassojen käsittelyyn haetaan usein hajautettua ratkaisua. Esimerkiksi Pohjois-Chilessä rakenteilla oleva LSST-teleskooppi kerää valmistumisensa jälkeen suunnitellun kymmenen vuoden käyttöaikansa kuluessa yli 50 petatavua dataa yötaivaasta [LSST 2018]. LSST:n datan varastointiin on kehitetty hajautettu tietokantajärjestelmä Qserv, joka on rakennettu MySQL:n ja hajautetun tiedostojärjestelmän Xrootd:n pohjalle [Wang et al. 2011].

Tässä tutkielmassa selvitän mitä ratkaisuja horisontaalista skaalautuvuutta varten on perinteisiin relaatiomallin mukaisiin tietokantajärjestelmiin kehitetty. Tarkastelen sekä reaaliaikaisia tarpeita palvelevia järjestelmiä että tiedon varastointiin ja raportointiin kehitettyjä järjestelmiä (OLAP-järjestelmät). Käyn läpi myös suosituimpien kaupallisten relaatiotietokantajärjestelmien hajautusominaisuudet. Skaalautuvista SQL-yhteensopivista ratkaisuista on käytetty termiä *NewSQL*. Tällä viitataan järjestelmiin, jotka pyrkivät samaan skaalautuvuuteen kuin NoSQL-järjestelmät säilyttäen kuitenkin ACID-säännöt [Aslett 2011a; 2011b].

Pyrin selvittämään soveltuvatko tällä hetkellä tarjolla olevat hajautetut relaatiotietokantajärjestelmät horisontaaliseen skaalautuvuuteen. Selvitän myös erilaisia tutkittuja tai esitettyjä tietokannan hajauttamisen malleja, joita ei ole tällä hetkellä toteutettu kaupallisiin järjestelmiin, mutta joista voisi olla hyötyä tulevaisuuden hajautusratkaisuissa.

Rajaan pois tästä katsauksesta NoSQL-järjestelmät ja keskityn relaatiomallin mukaisiin tietokantajärjestelmiin. Lisäksi rajaan pois pääosin sellaisten järjestelmien erityiskysymykset, joissa kokonaisratkaisuun kuuluvat tietokannat ovat keskenään erilaisia. Oman aihealueensa muodostavat tietokantajärjestelmät, joissa resurssit jaetaan

useamman tietokannan kesken ja kokonaisia tietokantoja siirretään tarvittaessa palvelinten välillä (katso esimerkiksi Elmore et al. [2011]). Keskityn tässä kuitenkin järjestelmiin, joissa yhden tietokannan sisältö on hajautettu tai kahdennettu useamman palvelimen kesken.

Luvussa 2 käydään läpi relaatiotietokantojen peruskäsitteet tietojen tallennukseen ja kyselyjen käsittelyyn liittyen. Luku 3 esittelee myöhemmissä osioissa käytettävän esimerkkitietokannan rakenteen ja sisällön. Luvussa 4 käsitellään yleisesti hajautettujen tietokantojen toimintaa sekä käsitteitä ja termejä. Luvussa 5 perehdytään olemassa olevien kaupallisten ja tutkimuskäytössä olevien relaatiotietokantojen hajautusratkaisuihin. Luvussa 6 tutkitaan erilaisia hajautusratkaisuja, joilla skaalautuvuuden ja korkean saatavuuden vaatimukset voidaan saavuttaa. Hajautuksen ja skaalautuvuuden asettamia vaatimuksia kyselynkäsittelylle käydään läpi luvussa 7. Luku 8 on varattu johtopäätöksille.

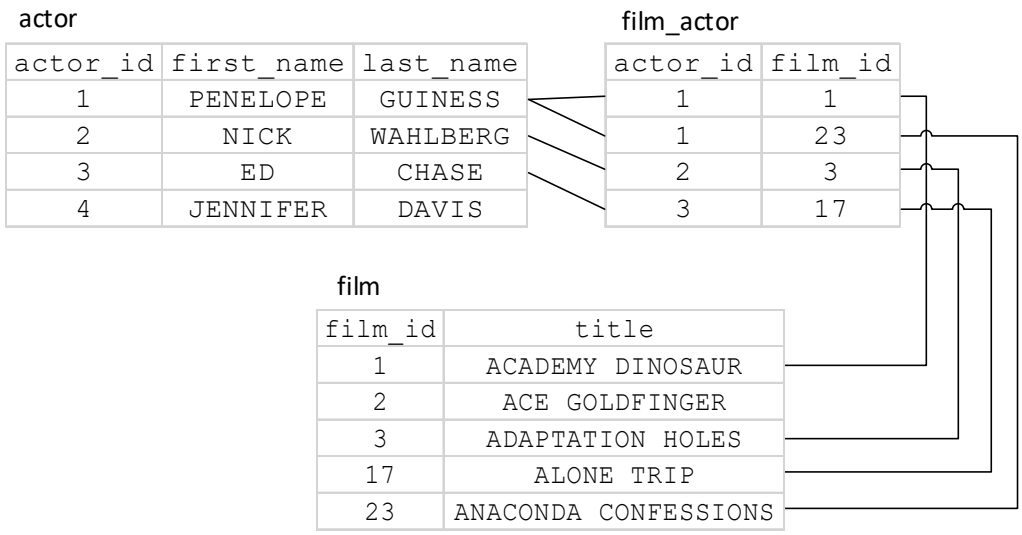
2. Tietokantajärjestelmien käsitteistöä

Tässä luvussa selvitetään erilaisia tietokantajärjestelmien käsitteitä. Luvussa keskitytään avaamaan relaatiotietokantoihin liittyviä termejä ja perustoimintoja, kuten kyselynkäsittelyä ja lukkojen hallintaa. Myöhemmissä luvuissa tutkitaan tarkemmin näiden ominaisuuksien toimintaa ja optimointia silloin, kun kyseessä on hajautettu tietokanta.

Tietokanta on kokoelma toisiinsa liittyviä tietoja. *Tietokantajärjestelmä* (database management system – dbms) on kokoelma ohjelmia, jotka mahdollistavat tietokannan luomisen ja ylläpidon. Tietokannan luonti aloitetaan määrittelemällä tiedon rakenne ja rajoitukset. Järjestelmä sisältää tietokantaan tallennetun tiedon lisäksi myös tiedot kannan rakenteesta. Lisäksi järjestelmä erottelee datan sitä käsittelevistä ohjelmista; mikäli tietokannan rakenteeseen tehdään muutoksia, ei järjestelmään yhteyttä ottavia ohjelmia yleensä tarvitse muuttaa. [Elmasri and Navathe 2011]

Relaatiotietokannassa tiedot on jaoteltu *tauluihin* eli *relaatioihin*, jotka edustavat käsitteitä ja käsitteiden välisiä suhteita. Myöhemmin esiteltävässä esimerkkitietokannassa taulu `film` edustaa käsitettä elokuva. Käsitteeseen liittyvät tiedot ovat taulun *attribuutteja* eli *sarakkeita*, esimerkiksi elokuvan nimi, valmistusvuosi ja kesto. Yksi taulun *rivi* eli *monikko* (tuple) vastaa yhtä elokuvaa. Taululla on yleensä *pääavain*, joka on yksilöllinen jokaiselle taulun riville ja jolla taulun yksittäiset rivit voidaan tunnistaa. Pääavain koostuu yhdestä tai useammasta taulun sarakkeesta. Usein pääavainta varten luodaan erillinen ID-sarake, jonka ainoa tarkoitus on yksilöidä rivit. Tällä sarakkeella ei ole käyttöä tietokannan ulkopuolella. Esimerkiksi `film`-taulussa on kokonaislukutyyppinen sarake `film_id`.

Taulujen väliset *yhteydet* eli *suhteet* määritellään *vierasavaimilla*. Vierasavain-attribuutti viittaa toisen taulun sarakkeeseen, joka sisältää viitattavan taulun pääavaimen. `film`-taulusta löytyy sarake `language_id`, joka on vierasavain tauluun `language`. Tämä avain kertoo `film`-taulun elokuvan kielen. Taulun `language` suhde tauluun `film` on *yhden suhde moneen* -tyyppinen, sillä yhdellä elokuvalla voi tässä rakenteessa olla vain yksi kieli, mutta monessa elokuvassa voi olla sama kieli käytössä. *Monen suhde moneen* -tyyppinen yhteys on esimerkiksi taulujen `film` ja `actor` välillä: elokuvalla voi olla monta näyttelijää ja sama näyttelijä voi liittyä useampaan elokuvaan. Tällöin taulujen väliset viittaukset tallennetaan erilliseen tauluun `film_actor`, jonka rivi viittaa sekä yhteen elokuvaan että yhteen näyttelijään. Kuvaan 1 on valittu neljä riviä `actor`-taulusta, viisi riviä `film`-taulusta, sekä näitä yhdistävät `film_actor`-taulun rivit (vain nimi-, otsikko-, pääavain- ja vierasavainsarakkeet on jätetty näkyviin), minkä lisäksi on viivoilla merkitty taulujen väliset yhteydet. Kaikkien taulujen rakenteet ja niiden väliset suhteet muodostavat yhdessä *tietokannan rakenteen*.



Kuva 1. Taulujen `actor`, `film_actor` ja `film` väliset viittaukset.

Tiedon lisäystä, muokkausta tai hakua varten asiakasohjelmistot lähettävät tietokantaan *kyselyjä* (query). Sanaa kysely käytetään yleisesti myös pyynnöistä, jotka päivittävät tietokannan dataa. Tietokannan operaatiot ajetaan usein osana *tapahtumia* eli *transaktioita* (transactions) ja tapahtuma voi sisältää yhden tai useamman *operaation* eli *lauseen*. Operaatio voi lisätä, päivittää tai poistaa tietokannan rivejä. Vaikka asiakasohjelmisto ei lähettäisi kyselyä tapahtuman osana, muodostaa järjestelmä yleensä automaattisesti kyselyn ympärille tapahtuman [MySQL Ref. Manual 2018] [PostgreSQL 2018].

Kun tapahtuma päättyy, sen pitää jättää tietokanta virheettömään tilaan, jossa kaikki tietokannan rajoitukset täyttyvät. Lisäksi tapahtuma pitää suorittaa aina kokonaan, tai peruuttaa kaikki sen tekemät muutokset. Tapahtuman viimeinen vaihe on joko *vahvistus* (commit) tai *peruutus* (rollback). Vahvistuksen onnistuttua tapahtuma katsotaan suoritetuksi, eivätkä sen tekemät muutokset saa enää kadota tietokannasta. Peruutuksen onnistuttua mitkään tapahtuman tekemät muutokset eivät ole näkyvissä tietokannassa.

Lukuoperaatiot tai *-kyselyt* palauttavat tietokannasta *tulostaulun*, joka muodoltaan vastaa tietokannan tauluja eli tieto koostuu riveistä ja sarakkeista. SQL on vakiintunut kyselykieli relaatiotietokannalle. Yksinkertainen lukuoperaatio voidaan esittää SQL:llä seuraavasti:

```
SELECT sarake1, sarake2, ...
FROM taulun_nimi
WHERE ehto;
```

SELECT-osiossa määritellään kyselyn palauttamat sarakkeet, FROM-osio kertoo lähteenä käytettävän taulun ja WHERE-osa eli ehto-osa sisältää ehdon, joka rajoittaa palautettavien rivien määrää. Ehto on looginen lauseke, joka koostuu AND- ja OR-operaatioilla yhdistetyistä ehdoista. Palautuneen tiedon sisältönä voi olla esimerkiksi yhden taulun koko sisältö, tai kyselyn ehto-osan rajaamat rivit. Useamman taulun tiedot saadaan yhdistettyä samaan tulostauluun käyttämällä *liitos*- eli *JOIN-operaatiota*. Liitosehdossa määritetään tapa, jolla eri taulujen rivit yhdistetään tulokseen. Mikäli liitosehdossa tarkastetaan ehdoissa käytettävien sarakkeiden arvojen yhtäsuuruus, on kyseessä *yhtäläisyysliitos* (equijoin).

Suorien kyselyiden lisäksi asiakasohjelmistot voivat ajaa *tallennettuja proseduureja* (stored procedure), jotka ovat tietokantaan tallennettuja aliohjelmia. Niille voidaan määritellä parametreja, jotka annetaan syötteenä proseduuria kutsuttaessa, ja ne voivat sisältää ohjelmalogiikkaa sekä kyselyjä. Tallennetut proseduurit vähentävät liikennettä asiakasohjelmiston ja tietokannan välillä.

Tietosisällön tallentamisen yksikköä tietokantajärjestelmässä kutsutaan yleensä *sivuksi* tai *lohkoksi.* Yleensä tietoa luetaan levyltä tietokannan muistiin sivu kerrallaan. Yhden sivun koko on tyypillisesti 2kB - 8kB ja se koostuu otsikkotiedoista sekä taulun riveistä.

2.1. Kyselyjen optimointi

Kyselyjen optimoinnissa tietokantaan saapuvasta kyselystä muodostetaan *kyselysuunnitelma* (*QEP* - Query execution plan), jossa kyselyn sisältö jaetaan järjestelmän sisäisesti käyttämiin tiedonkäsittelyoperaatioihin. Suunnitelma muodostetaan siten, että suunnitelman *kustannus* on mahdollisimman pieni. Kustannus määritellään kustannusfunktiolla (cost function), jossa eri tietokantaoperaatioille määritellään hinnat siten, että raskaimmalla operaatiolla on suurin kustannus. Optimointia suorittava tietokannan osa on *kyselyn optimoija* (query optimizer). [Elmasri and Navathe 2011, pp. 679, 700-701]

QEP muodostuu yleensä kolmesta komponentista: *hakuavaruudesta*, *kustannusmallista* ja *hakustrategiasta*. Hakuavaruus on joukko vaihtoehtoisia hakusuunnitelmia kyselylle. Hakusuunnitelmat antavat saman hakutuloksen, mutta eroavat operaatioiden ajojärjestyksessä ja toteutuksessa. Kustannusmalli ennustaa kyselysuunnitelman kustannuksen. Hakustrategia tutkii hakuavaruutta ja valitsee kustannusmallia käyttäen parhaan suunnitelman. Hajautetussa järjestelmässä käytettävässä kustannusmallissa täytyy olla huomioituna hajautetun ratkaisun erityispiirteet. [Özsu and Valduriez 2011]

Monimutkaisissa kyselyissä yhtä tehokkaiden operaatiojärjestysten lukumäärä voi olla niin suuri, että vaihtoehtoisten operaatiojärjestysten sisältävän hakuavaruuden läpikäynti kokonaan kestäisi selvästi kauemmin kuin itse operaation ajo. Esimerkiksi N:n eri relaation väliset JOIN-operaatiot voi järjestää O(N!):llä tavalla liitospuuksi. Tämän vuoksi kyselyn optimoijat yleensä rajoittavat hakuavaruuden kokoa heuristiikan avulla. [Özsu and Valduriez 2011, pp. 247-248]

Hakuavaruutta voidaan rajoittaa myös liitospuun muodolla. *Lineaarisessa liitospuussa* (linear join tree) jokaisessa liitosoperaatiossa vähintään toinen operandeista on lähdetaulu. *Tuuheassa liitospuussa* (bushy join tree) saattaa olla myös liitosoperaatioita, missä molemmat operandit ovat kyselyn käytössä olevia väliaikaisia relaatioita. Jos tuuheat puut jätetään pois hakuavaruudesta, hakuajan suuruusluokka supistuu kokoon $O(2^N)$. Toisaalta hajautetussa järjestelmässä tuuheat liitospuut ovat joissakin tilanteissa hyödyllisiä, koska niiden yhteydessä voidaan käyttää rinnakkaisuutta. [Özsu and Valduriez 2011, p. 248]

2.2. Samanaikaisuuden hallinta

Samanaikaisuuden hallinnalla varmistetaan, että samanaikaiset tapahtumat on eristetty toisistaan. Useimmilla tekniikoilla pyritään *sarjallistuvuuteen* (serializability) eli siihen, että tapahtuminen samanaikainen ajo jättää tietokannan samaan tilaan, kuin jos tapahtumat olisi ajettu peräkkäin. Koska sarjallisuus on olennainen osa nykyaikaisia tietokantajärjestelmiä, pitää yleensä myös hajautetuissa tietokantajärjestelmissä huolehtia sarjallisuuden toteutumisesta.

2.2.1. Lukot

Sarjallistuvuus voidaan toteuttaa *lukkojen* avulla. Tietokantajärjestelmässä lukko varaa taulun, sarakkeen tai osan taulun riveistä siten, että muut kyselyt eivät voi samanaikaisesti kirjoittaa varattuun dataan.

Lukoilla transaktiot varaavat käyttöönsä tietokannan tauluja, rivejä tai sivuja. Normaalisti lukkoja on vähintään kahdenlaisia: *lukulukkoja* (shared lock) ja *kirjoituslukkoja* (exclusive lock). Kun transaktio lukee tietuetta, sen tarvitsee lukita tietue lukulukolla ennen lukua. Kirjoituslukkoa tarvitaan, jos kyseistä riviä päivitetään. Lukulukko ei estä muiden transaktioiden lukuoperaatioita, mutta kirjoituslukko estää kaikki muut kirjoitus- ja lukuoperaatiot. *2-vaiheisessa lukituksessa* lukot hankitaan ensimmäisessä vaiheessa ja vapautetaan toisessa vaiheessa. Tällöin mitään tietoja ei voi enää lukita ensimmäisen lukon vapautuksen tapahduttua. 2-vaiheinen lukitus takaa sarjallistuvuuden.

Lukkiuma (deadlock) syntyy, jos kaksi tapahtumaa jää odottamaan toistensa lukkoja. Esimerkiksi tapahtuma A kirjoittaa ensin riviin *m* ja samanaikaisesti tapahtuma B kirjoittaa saman taulun riviin *n*. Jos nyt A pyrkii lukemaan riviä *n*, se jää odottamaan B:n lukon vapautumista kyseiseltä riviltä. Jos B pyrkii lukemaan riviä *n*, josta puolestaan A:lla on lukko, syntyy lukkiuma. Yleensä nykyiset tietokantajärjestelmät havaitsevat tilanteen automaattisesti ja ratkaistaan esimerkiksi peruuttamalla toinen tapahtumista.

2.2.2. Aikaleimaperustainen rinnakkaisuuden hallinta

Tätä hallintaa käytettäessä tapahtumat järjestetään aikaleimojensa perusteella (tapahtuman aikaleima on sen aloitusaika). Tällöin rinnakkaisten tietokantatapahtumien vastaava sarjallinen suoritusjärjestys on aina aikaleimojen mukainen vanhimmasta uusimpaan. Hallinta-algoritmi liittää jokaiselle seurattavalle tietokannan osiolle luku- ja kirjoitusaikaleimat. Mikäli tapahtuma yrittää käynnistää luku- tai kirjoitusoperaation ja sen aikaleima on pienempi kuin kohteen aikaleimat, kyseinen tapahtuma peruutetaan ja käynnistetään uudestaan uudella aikaleimalla.

2.2.3. Optimistinen rinnakkaisuuden hallinta

Toisin kuin edellisissä rinnakkaisuuden hallinnan tavoissa, optimistisessa rinnakkaisuuden hallinnassa luku- ja kirjoitusoikeus dataan tarkastetaan vasta sen jälkeen, kun uusi data on valmis kirjoitettavaksi tietokantaan. Tapahtumat tekevät kirjoitusoperaatiot ensin paikallisiin kopioihin päivitettävästä datasta. Tarkastusvaiheessa tarkistetaan, ettei mikään ajettava päivitys riko sarjallisuutta. Jos tarkastusvaiheessa ei todeta ristiriitoja, ajetaan päivitykset varsinaiseen tietokantaan. Mikäli ristiriita löydetään, hylätään tapahtuman laskemat päivitykset ja tapahtuma käynnistetään uudestaan.

2.3. Tiedon oikeellisuus ja säilyvyys

ACID-periaatteet (Atomicity, Consistency, Isolation, Durability) kuvaavat neljää relaatiotietokannan periaatetta, jotka liittyvät järjestelmän tietojen eheyteen ja säilyvyyteen. *Atomisuus* (atomicity) tarkoittaa, että järjestelmässä ajettu tapahtuma ajetaan joko kokonaisuudessa tai kaikki sen tekemät muutokset perutaan. *Eheys* (consistency) takaa, että tapahtuman jälkeen tietokanta on aina eheässä tilassa eli rakenteelle ja tietoalueille asetetut rajoitukset täytetään. Esimerkiksi aiemmin mainitun `film`-taulun `language_id`-vierasavainta vastaavien arvojen pitää löytyä `language`-taulun pääavaimesta. *Eristyneisyys* (isolation) piilottaa samanaikaiset keskeneräiset tapahtumat toisiltaan, jolloin samanaikaisten ajojen lopuksi järjestelmä jää samaan tilaan, kuin jos tapahtumat olisi ajettu peräkkäin. *Pysyvyys* (durability) tarkoittaa, että vahvistettujen tapahtumien tekemien muutosten täytyy säilyä, vaikka järjestelmä kaatuisi tai palvelin sammutettaisiin. Käytännössä tämä tarkoittaa yleensä tapahtuman

muutosten kirjoittamista levylle. Hajautetussa järjestelmässä saatetaan pysyvyyden ehtona pitää myös tapahtuman kirjoittamista $k > 1$ eri palvelimelle, missä k:n suuruus on säädettävissä [Jones et al. 2010]. Myös k*eskusmuistipohjaisissa tietokantajärjestelmissä* (in-memory databases), joissa tietokannan koko sisältö ladataan keskusmuistiin, pyritään jatkuva levylle kirjoittaminen korvaamaan muilla keinoilla.

Normalisointi on tietokannan rakenteelle tehtävä useasta askeleesta koostuva toimenpide, jolla taulut jaetaan siten, ettei samaa tietoa tallenneta useampaan kertaan eri paikkoihin. Tällä myös yksinkertaistetaan tietojen päivitystä. *Denormalisoinnissa* normalisoitujen taulujen sisältö yhdistetään, jolloin rakenne on sama kuin ennen normalisointia.

Järjestelmäloki eli *tapahtumaloki* huolehtii järjestelmän palautumisesta järjestelmävirheen sattuessa. Kaikki tietokantajärjestelmän tapahtumat kirjoitetaan tietokantalokiin ja lokin sisältö kirjoitetaan ennalta määrätyin väliajoin levylle. Järjestelmävirheen sattuessa menetetään ne tiedot, joita ei ole tallennettu tapahtumalokista levylle. Mikäli tietokanta joudutaan palauttamaan tilaan, jossa ei ole kaikkia lokiin kirjoitettuja tapahtumia, voidaan tapahtumat *uudelleenkirjoittaa* (redo operation) lokista tietokantaan. Tapahtumalokia voidaan myös käyttää peruutettavan tapahtuman aiemmin tehtyjen, peruutettavien muutosten hakemiseen ja kumoamiseen (undo operation). Mikäli lokia käytetään tähän tarkoitukseen, voidaan siitä käyttää nimitystä *peruutusloki* (undo log).

2.4. OLTP- ja OLAP-järjestelmät

OLTP (Online transaction processing) tai *tosiaikainen tapahtumankäsittely* on tietokannan käyttötapa, jossa järjestelmään tulee suuri määrä yksinkertaisia kyselyjä, jotka yleensä pitää suorittaa nopeasti ja reaaliajassa. OLTP-järjestelmiä ovat muun muassa pankkien ja pörssien tietokantajärjestelmät, joissa liikkuu jatkuvasti tietoa kaupankäynnistä ja rahaliikenteestä. Esimerkiksi New Yorkin pörssi toimii Oracle OLTP-ratkaisun päällä ja järjestelmässä tapahtuu päivittäin noin miljardi tapahtumaa [Singh 2018]. OLTP-järjestelmästä syntynyttä tietoa siirretään yleensä ajastetusti tietovarastoon (data warehouse), missä siitä voidaan OLAP-työkalujen avulla käyttää raportointiin ja tiedonlouhintaan.

OLTP-järjestelmien tyypillisiä piirteitä ovat:

1. Useimmat tapahtumat käyttävät vain pientä määrää rivejä.
2. Useimpien tapahtumien suoritusaika on lyhyt.
3. Tapahtumista suuri osa koostuu tallennettujen proseduurien ajosta.

Tosiaikaisen tiedonjalostuksen (OLAP – Online analytical processing) järjestelmät taas keskittyvät kerätyn tiedon raportointiin ja *liiketoimintatiedon hallintaan* (business intelligence) yrityksen päätöksenteon tueksi. Raportoidun tiedon keräämistä varten OLAP-järjestelmässä ajetaan paljon raskaita ja pitkäkestoisia lukukyselyjä. Yleensä päivitykset tietoon tehdään tietyin väliajoin OLTP-järjestelmän puolelta, esimerkiksi kerran päivässä. OLAP-tietovarasto koostuu faktatauluista ja dimensiotauluista. Fakta-tauluissa on tallennettuna varsinainen liiketoiminnasta kerääntyvä data ja dimensiotauluista löytyy lisätietoja faktatauluun kerättyjen rivien eri kentistä. Fakta-taulut viittaavat vierasavaimilla dimensiotauluihin. Fakta-tauluja on usein yhdessä tietokannassa vain yksi, kun taas dimensiotauluja on useita (esimerkiksi TPC-H suorituskykytestaus-tietokannassa faktatauluja on 2 ja dimensiotauluja 6). Dimensiotaulujen sisältö voi olla denormalisoitua, jolloin faktataulu viittaa suoraan kaikkiin dimensiotauluihin (tähtimalli). Mikäli dimensiotaulujen tieto on normalisoitua, on kyseessä lumihiutalemalli. [Kimball and Ross 2013]

3. Esimerkkitietokanta

Esimerkkitietokannan avulla voidaan tarkastella erilaisten hajautusmallien toimintaa konkreettisten taulujen tasolla. Tässä tutkielmassa esimerkkitietokantana käytetään MySQL-tietokantajärjestelmän mukana toimitettavaa Sakila-tietokantarakennetta ja sen esimerkkisisältöä. Sakila mallintaa videovuokraamon toiminnassa varastoitavia tietoja. Valitsin Sakilan sen helposti hahmotettavan ja suhteellisen yksinkertaisen rakenteensa vuoksi.

Tietokanta koostuu kuvitteellisista elokuvista, varastotiedoista, myymälöistä ja näiden osoitetiedoista. Taulukossa 1 ovat listattuina tietokannan taulut ja kunkin taulun tietokantarivien määrät.

Taulukko 1. Sakila-tietokannan esimerkkisisällön rivimäärät.

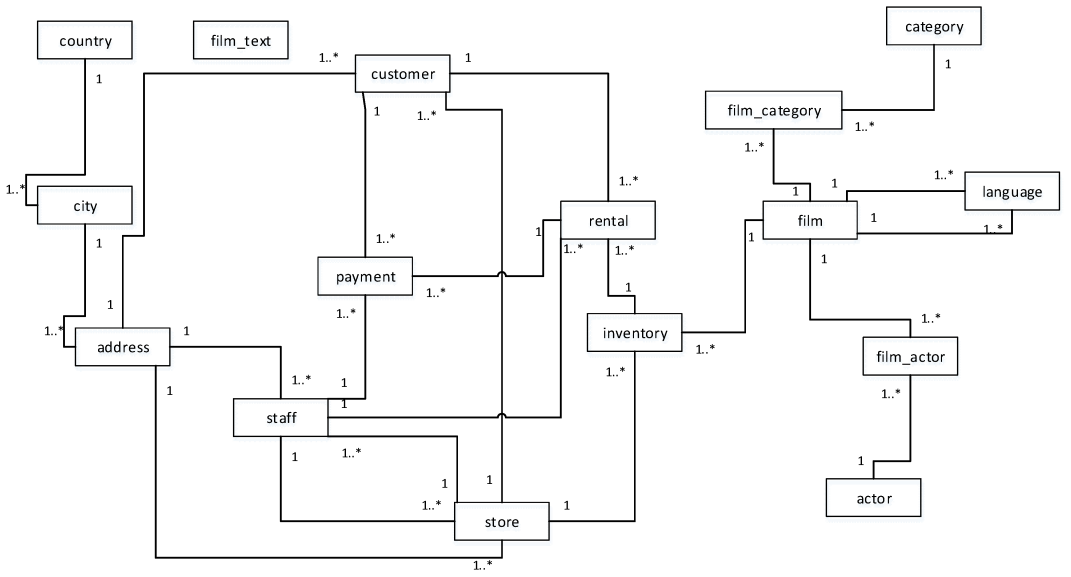
Taulun nimi	Rivien määrä	Taulun kuvaus
actor	200	tiedot näyttelijöistä
address	605	asiakkaiden, henkilökunnan ja toimipisteiden osoitetiedot, viiteavain tauluun city
category	16	elokuvagenret
city	600	kaupungit, tauluun viiteavain taulussa address
country	109	maat, taulusta city viitataan tähän tauluun
customer	599	asiakkaat, viiteavain address-tauluun ja store-tauluun (asiakkaan ”oma” toimipiste)
film	1000	elokuvat
film_actor	5462	elokuvan näyttelijät, viiteavaimet tauluihin film ja actor
film_category	1000	elokuvan genre, viiteavaimet tauluihin film ja category
film_text	5588	film-taulun aputaulu, jossa lisätietoja elokuvista
inventory	4581	yksittäiset vuokrattavat elokuvan kopiot vuokraamoissa, viiteavaimet tauluihin film ja store (toimipiste, jossa kyseinen kappale on vuokrattavissa)
language	6	kielet, viiteavaimet taulusta film
payment	16049	maksutapahtumat
rental	16044	vuokraustapahtumat, viiteavaimet tauluihin customer ja staff
staff	2	työntekijät, viiteavain tauluun store (työntekijän ”oma” toimipiste)
store	2	toimipisteet

Kuvassa 2 on esitetty Sakila-tietokannan taulut ja niiden väliset suhteet. Useimmin päivittyvät taulut ovat keskellä näkyvät payment (maksutapahtumat), rental (vuokraustapahtumat) ja inventory (vuokrattavat kopiot). Nämä puolestaan yhdistyvät vierasavaimilla elokuvien, asiakkaiden ja toimipisteiden tietoihin.

Taulussa rental ovat vuokraustapahtumien tiedot ja se sisältää viittaukset vuokranneeseen asiakkaaseen, vuokrauksen käsitelleeseen työntekijään ja vuokratun

elokuvan kopioon. `inventory`-taulussa ovat tiedot elokuvien yksittäisistä kopioista ja se yhdistyy tauluihin `film` (elokuvan tiedot) ja `store` (myymälä, jossa kyseinen kopio sijaitsee). `payment` sisältää tiedot maksutapahtumista ja se yhdistyy vuokraustapahtumaan, maksaneeseen asiakkaaseen ja maksun käsitelleeseen myyjään. Asiakkaiden, työntekijöiden ja myymälöiden osoite- ja paikkatiedot löytyvät tauluista `address`, `city` ja `country`.

Taulu `film` sisältää elokuvien tiedot ja siihen liittyviä tauluja ovat elokuvagenren sisältävä `film_category`, elokuvan kielien tiedot sisältävä `language` ja elokuvassa näytteleviin näyttelijöihin viittaava `film_actor`. Taulu `film` viittaa kahteen kertaan tauluun `language`: kenttä `language_id` kertoo vuokrattavan kopion kielen ja `original_language_id` elokuvan alkuperäisen kielen. Taulu film_text on taulu, joka sisältää samat rivit kuin `film`, mutta vähemmän sarakkeita.



Kuva 2. Sakila-tietokannan rakenteen relaatiokaavio.

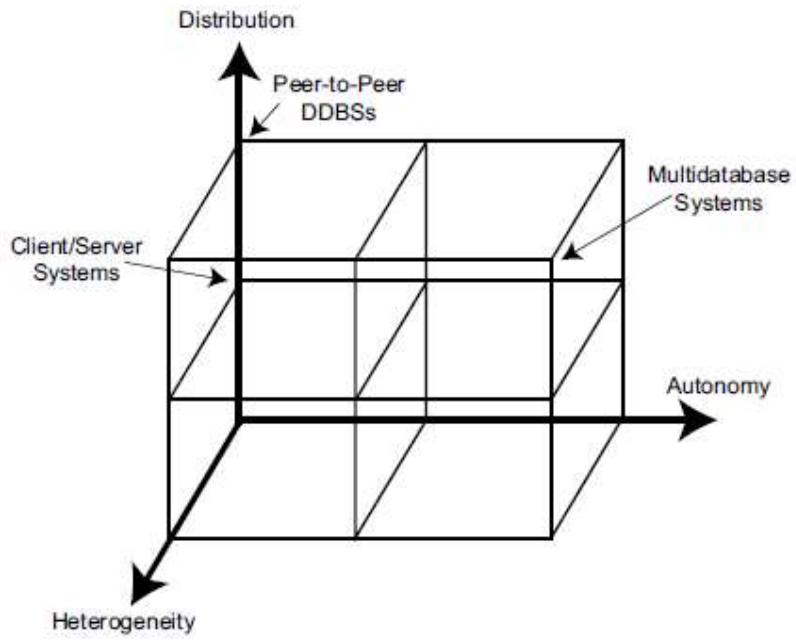
4. Rinnakkaisten tietokantajärjestelmien toimintaperiaatteet

Hajautettu tietokantajärjestelmä on kokoelma loogisesti toisiinsa liittyviä tietokantoja, jotka on hajautettu tietokoneverkkoon [Özsu and Valduriez 2011]. Vaikka yleisemmin tietokantajärjestelmä tarjoaa yhtenäisen ja hallitun pääsyn kokoelmaan tietoja, ei hallinnan tai tiedon tallennuksen tarvitse välttämättä olla keskitettyä, mikä mahdollistaa myös hajautetut järjestelmät. Tietokannat voivat kuitenkin olla fyysisesti samassa tilassa, kuten palvelinsalissa. Hajautettu tietokanta koostuu useammasta palvelimesta, joiden välille tietokannan rakenne on jaettu. Näitä järjestelmän muodostavia palvelimia kutsutaan *noodeiksi* eli *tietokantanoodeiksi* eli *tietokantapartitioiksi*.

Özsu ja Valduriez [2011] luokittelevat hajautetut tietokanta-arkkitehtuurit kolmen ominaisuuden mukaan (kuva 3): *itsenäisyyden* (autonomy), *epäyhtenäisyyden* (heterogeneity) ja *tiedonhallinnan tehtävien jakautumisen* (distribution). Itsenäisyydellä voidaan tarkoittaa toisaalta yksittäisten tietokantapalvelimien tietoa muista palvelimista tai laajemmin järjestelmästä ja toisaalta yksittäisten palvelimien ohjattavuutta niiden ulkopuolelta. Epäyhtenäisyydellä kuvataan yksittäisten palvelimien samankaltaisuutta: käytetäänkö jokaisella palvelimella samaa tietokantajärjestelmää, vai onko palvelinten välillä vaihtelua. Tiedonhallinnan tehtävät voivat puolestaan jakautua asiakas- ja palvelinkoneisiin, jolloin asiakaskone huolehtii tietokantakyselyn lähetyksestä ja palvelinkone sen vastaanotosta sekä käsittelystä. Mikäli tätä jaottelua ei tehdä, toimii jokainen järjestelmään osallistuva kone sekä asiakkaana, että palvelimena ja kyse on tällöin vertaisverkosta. Pienin tehtävien hajautuksen aste on järjestelmällä, jota ei ole lainkaan hajautettu.

Järjestelmät sijoittuvat eri kohtiin kuvan 3 kuvaajalla riippuen niiden arkkitehtuurista. Kuvassa näkyvistä esimerkkiratkaisuista asiakas/palvelin-järjestelmät (Client-Server Systems) ja hajautetut vertaistietokannat (Peer-to-Peer DDBSs) sijoittuvat kuvan nuolien osoittamiin kohtiin vain, jos ne koostuvat keskenään samanlaisista tietokannoista ja niiden toiminta on sidottu vahvasti kokonaisjärjestelmään. Tällöin niiden itsenäisyyden ja epäyhtenäisyyden aste on alhainen.

Kuvan kolmannessa Multidatabase Systems -esimerkissä järjestelmä koostuu useasta itsenäisestä tietokannasta, joita ei ole mitenkään sidottu järjestelmään, eikä niillä luultavasti ole tietoa toistensa olemassaolosta. Ne ovat myös arkkitehtuuriltaan keskenään erilaisia.



Kuva 3. Hajautettujen järjestelmien luokittelu itsenäisyyden, epäyhtenäisyyden ja tiedonhallinnan tehtävien jakautumisen mukaan [Özsu and Valduriez 2011, p. 25].

Jatkossa rajoitun tarkastelemaan järjestelmiä, joissa järjestelmään osallistuvat palvelimet ovat keskenään yhtenäisiä ja lähtökohtaisesti niissä kaikissa on sama tietokantajärjestelmä. Myös asiakas- ja palvelinkoneiden tehtävänjako on tarkkaan rajattu, eivätkä tarkastelemissani järjestelmissä tietokantanoodit toimi asiakkaina, vaikka ne voivatkin lähettää kyselyn osia muille noodeille tarpeen vaatiessa. Itsenäisyyden taso voi sen sijaan vaihdella; toisissa järjestelmissä noodeilla käytetään tavallista hajauttamatonta versiota tietokantapalvelimesta ja toisissa järjestelmissä noodit huolehtivat kyselyyn tarvittavien tietojen hankinnan muilta noodeilta (esimerkiksi H-Store).

Hajautetussa tietokannassa rajapintana asiakasohjelmistojen suuntaan toimii joko tietokannan sisällöstä erillinen *väliohjelmisto* (middleware) tai ohjausohjelma (esimerkiksi MaxScale). Vaihtoehtoisesti kyselyt voidaan syöttää suoraan noodeille, jotka hoitavat tarvittaessa kyselyn ohjauksen eteenpäin (esimerkiksi H-Store). Edellisessä tapauksessa ohjausohjelmisto jakaa kyselyt osiin ja lähettää ne eteenpäin palvelimille. Kyselyjen käsittelyssä voi olla käytössä myös erillinen ohjauskone. Järjestelmän hajautus voidaan piilottaa myös käyttöjärjestelmätason ratkaisuilla, jolloin järjestelmä näyttää käyttäjälle yksittäiseltä palvelimelta, vaikka esimerkiksi sen levyresurssit on voitu hajauttaa usean järjestelmän kesken [Page et al. 1985] [Wang et al. 2011].

Yleensä järjestelmät toteuttavat *hajautuksen tuntumattomuuden* (distribution transparency), jolloin asiakasohjelmiston ei tarvitse tietää hajautuksen arkkitehtuurista. Tietokanta näyttäytyy loogisina tauluina, joista kustakin saattaa olla olemassa useampi fyysinen kopio, tai joiden tieto saattaa olla hajautettu useamman noodin kesken [Özsu and Valduriez 2011]. Esimerkiksi hajautettuun Sakila-kantaan voidaan antaa normaalisti kysely `SELECT * FROM country;`, vaikka taulu `country` löytyisi useammalta noodilta tai vaikka sen tiedot olisi hajautettu useamman noodin kesken.

4.1. Tietokantanoodit ja niiden hallinta

Rinnakkaisessa tietokantajärjestelmässä tietokannan sisältö jakautuu tietokantanoodien kesken, joissa jokaisessa sijaitsee osa tietokannan taulujen tiedoista. Tietokantanoodi on enemmän tai vähemmän itsenäisesti toimiva tietokantajärjestelmä, joka hoitaa joko osaa tai kaikkia perinteisen tietokantajärjestelmän tehtävistä. Yleensä yhdelle tietokantanoodille on varattu oma erillinen palvelimensa, jonka sisältämiä resursseja kuten muistia ja prosessoria ei jaeta. Periaatteessa samalla palvelimella voi myös toimia useampi noodi esimerkiksi omiksi prosesseikseen hajautettuina.

Hajautetussa tietokantajärjestelmässä on järjestelmän tauluihin tallennettu tieto kopioitu tai hajautettu noodien kesken. Sisällön ollessa kopioituna useammalle tietokantanoodille puhutaan *replikoinnista*. Mikäli yhden taulun tiedot on hajautettu useamman noodin välille, puhutaan *partitioinnista* (partitioning), *osituksesta* tai *pirstaloinnista* (sharding). Partitioinnista käytetään satunnaisesti myös termejä fragmentation ja declustering. Replikointia ja partitiointia voidaan myös käyttää rinnan, jolloin partitioitu osa taulusta replikoidaan myös toiselle noodille.

Dataa sisältävien datanoodien lisäksi järjestelmässä voi olla yksi tai useampi *kuormantasaaja* eli palvelin, joka vastaanottaa kyselyjä ja ohjaa niitä tarvittaessa eteenpäin oikeisiin tietokantanoodeihin. Kyselyt pyritään ohjaamaan siten, että järjestelmän kokonaiskuorma on jakautunut tasaisesti noodien välille. Sitä osaa järjestelmästä, joka vastaanottaa kyselyn, erittelee sen osasiin tarvittaessa ja välittää osaset oikeille noodeille kutsutaan *kyselynkäsittelijäksi*.

Järjestelmän kokonaisuuden hallinnassa saatetaan käyttää erillistä *hallintanoodia*. Usein hallintanoodi hoitaa virheidenkäsittelyn ja muutokset järjestelmän datanoodien kokoonpanossa. Tyypillisiä virheitä ovat yhden tietokantanoodin kaatuminen tai verkkohäiriöt, joiden vuoksi noodit eivät saa enää yhteyttä toisiinsa

Esimerkiksi MySQL Cluster käyttää yhtä tai useampaa SQL-palvelinta (SQL nodes), jotka ottavat vastaan kyselyt ja hakevat tiedot erillisiltä tietokantanoodeilta (Data nodes). Lisäksi MySQL Clusterissa on erillinen hallintanoodi (management node), joka ohjaa keskitetysti sekä SQL noodien että tietokantanoodien toimintaa. MySQL Clusterissa

hallintanoodi pystyy myös hoitamaan versiopäivitykset datanoodeihin ilman käyttökatkoa ajamalla datanoodit yksi kerrallaan pois käytöstä päivitysten ajaksi. Tätä varten tiedon pitää olla replikoituna toiselle palvelimelle, jota voidaan käyttää päivitettävän palvelimen sijaan.

Hajautettujen tietokantojen tutkimuksessa *tietokantarypäs* tai *-klusteri* (database cluster) on yleensä määritelty ryhmänä itsenäisiä tietokantapalvelimia, joissa toimii valmiina saatava räätälöimätön tietokantajärjestelmä. Koska yksittäisten noodien tietokantajärjestelmää ei ole muokattu hajautuksen tarpeisiin, pitää esimerkiksi kyselyjenkäsittely toteuttaa väliohjelmiston avulla [Akal et al. 2002] [Özsu and Valduriez 2011]. Tietokantaryppään merkitys tosin vaihtelee eri yhteyksissä: esimerkiksi PostgreSQL:ssä tietokantarypäs koostuu yhden palvelinprosessin alaisista tietokannoista [PostgreSQL 2018]. Jatkossa käytän termiä väljästi tarkoittamaan hajautettua järjestelmää, jossa noodien määrää ei ole rajattu ja kaikki noodit ovat osana samaa hajautettua järjestelmää. Usein noodien toimintaa ohjataan väliohjelmistolla ja kuormantasaaja ohjaa kyselyt noodeille. Kaupallisista klusteriratkaisuista suurin osa toteuttaa tämän määritelmän (muun muassa Galera cluster, MySQL NDB Cluster ja Tungsten Clustering).

4.2. Replikointi

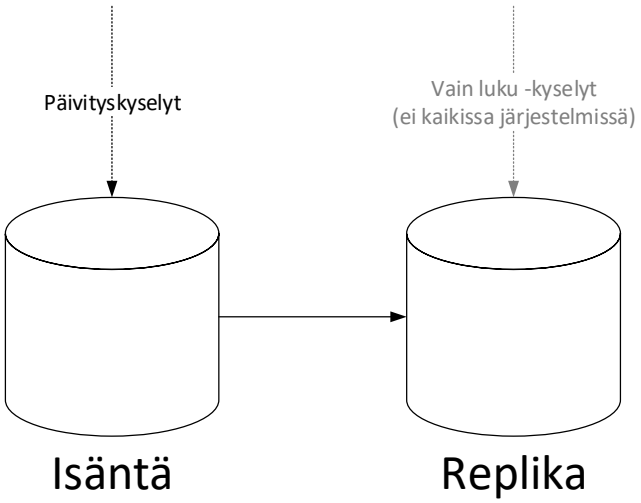
Replikoinnissa osa tai kaikki tietokantanoodin tiedoista kopioidaan toiseen noodiin, minkä jälkeen tiedot synkronoidaan tasaisin väliajoin. Tällöin tieto on saavutettavissa molemmilta noodeilta, vaikka usein vain alkuperäinen noodi ottaa vastaan tietokantakyselyjä. Mikäli tietokannan sisältö on *täysin replikoitu*, replikoidaan kaikki taulut toiselle noodille. Mikäli on tehty vain *osittainen replikointi*, on toiselle noodille kopioitu vain osa tauluista (tai niiden osia). Replikoinnilla saavutetaan useita hyötyjä:

1. Järjestelmän saavutettavuus: Tiedon ollessa useammalla noodilla ei yhden noodin kaatuminen kaada koko järjestelmää (jos vain osa tauluista on replikoitu, ei saavutettavuus parane).
2. Suorituskyky: Jos tieto on replikoitu sille noodille, missä kysely suoritetaan, ei kyseisen taulun rivejä tarvitse hakea toiselta noodilta, jolloin säästetään noodien väliseen kommunikointiin tarvittava aika.
3. Skaalautuvuus: Järjestelmän noodien määrän kasvaessa tiedon replikoinnilla saadaan vasteajat pidettyä alhaisempina.
4. Sovelluksien vaatimukset: Osana sovelluksen järjestelmävaatimuksia saattaa olla tiedon tallentaminen useampana kopiona.

[Özsu and Valduriez 2011]

Replikaatioratkaisu on *keskitetty*, mikäli kyseessä on järjestelmä, jossa uudet tiedot kirjoitetaan aina tiettyyn tietokantaan. Tällöin kyseistä noodia kutsutaan *isäntätietokannaksi* (master database) ja toissijaisia tietokantoja, joihin tieto kopioidaan, *replikoiksi* eli *replika-* tai *orjatietokannoiksi* (slave). Replikoidussa kannassa voidaan ajaa ainoastaan lukukyselyjä, mikäli kyselyt replikaan ylipäänsä sallitaan. Yksinkertainen keskitetty replikointi esitetään kuvassa 4. Tässä arkkitehtuurissa vältetään eri noodeilla samaa tietoa muokkaavien kyselyjen aiheuttamat ristiriitatilanteet. *Hajautetussa* replikointiratkaisussa (myös distributed group replication tai multi-master replication [Gray et al. 1996]) päivitysoperaation voi puolestaan tehdä millä tahansa järjestelmän noodilla.

Usein järjestelmissä, joissa on toteutettu yksinkertainen replikaatio, ei ole erillistä ohjainnoodia, vaan mukana olevat tietokannat huolehtivat keskenään replikoinnin toimivuudesta. Useimmat tietokantajärjestelmät tukevat replikaatiota. Esimerkiksi sekä MySQL:n että PostgreSQL:n uusimmat tuotantoversiot tarjoavat keskitetyn replikoinnin. Lisäksi molempiin löytyy kolmansien osapuolten toteuttamia vaihtoehtoisia toteutuksia.



Kuva 4. Keskitetty replikointi

Esimerkki 1: Ensisijaiseen tietokantaan A on luotu Sakila tietokannan rakenne ja tuotu `country`*-tauluun neljä riviä. Tämä replikoidaan nyt toiseen tietokantaan B. Tällöin tietokannasta A tulee isäntätietokanta ja tietokannasta B orjakanta. Replikaa pystytettäessä kopioidaan* `country`*-taulun rakenne ja sen neljä riviä (taulukko 2). Kun replika on toiminnassa, siitä siis löytyvät samat taulukon 2 rivit kuin*

tietokannasta A. Tällöin voidaan ajaa SELECT-kyselyjä `country`*-tauluun kummassa tahansa tietokannassa. Sakila-tietokannassa voitaisiin esimerkiksi kopioida kaikkien taulujen tiedot toiseen kantaan, tai vain pienempien taulujen* `address`*,* `city` *ja* `country` *tiedot.*

Taulukko 2. Taulun `country` 4 riviä

country_id	country	last_update
1	Afghanistan	2006-02-15 04:44:00
2	Algeria	2006-02-15 04:44:00
3	American Samoa	2006-02-15 04:44:00
4	Angola	2006-02-15 04:44:00

Replikoidun tietokantajärjestelmän pitää huolehtia tietokannan eheyteen ja *ristiriidattomuuteen* (consistency) liittyvistä kysymyksistä. Mikäli järjestelmä toteuttaa *vahvan konsistenssin* (strong consistency), kaikki noodit ovat samassa tilassa päivitysoperaation jälkeen. *Heikko konsistenssi* (weak consistency) sallii myös vapaamman päivitysstrategian.

Synkronisessa replikoinnissa (synchronous tai eager replication) kaikki tietokantanoodit näyttäytyvät aina asiakasohjelmistoille samassa tilassa. Jos edellisessä esimerkissä syötetään `country`-tauluun tietokannassa A uusi rivi (`5,'Anguilla','2006-02-15 04:44:00'`), saa rivin päivittävä asiakasohjelmisto tiedon onnistuneesta tapahtumasta vasta, kun replikastakin löytyy kyseinen rivi. Kaikki replikat päivitetään siis isäntänoodin tasalle päivittävän tapahtuman sisällä. *Asynkronisessa replikoinnissa* (asynchronous tai lazy replication) tapahtuma vahvistetaan, kun se on kirjoitettu isäntäkantaan, ja muihin tietokantoihin kopiointi suoritetaan tapahtuman jo loputtua. Tällöin siis lukukysely voi hetkellisesti palauttaa eri arvoja riippuen siitä, ohjataanko se isäntäkantaan vai replikaan. [Gray et al. 1996]

Asynkronista replikointia on pidetty suorituskyvyn kannalta parempana ratkaisuna ja suurin osa nykyisistä kaupallisista replikointiratkaisuista käyttää asynkronista replikointia. Mikäli synkronisen replikoinnin järjestelmässä voidaan ajaa päivityksiä useammalla noodilla, ongelmana ovat lukkiumat ja niiden automaattinen ratkaisu. Kun järjestelmä skaalautuu ylöspäin, lukkiumien määrä kasvaa, mikä saattaa lopulta johtaa järjestelmän tilaan, josta se ei enää pysty palautumaan. Mikäli päivityksiä voidaan ajaa vain yhdellä noodilla, kyseisestä noodista voi muodostua järjestelmän pullonkaula. [Gray et al. 1996]

Kemme ja Alonso [2000] ehdottavat synkronisen replikoinnin suorituskykyongelmien ratkaisemiseksi protokollaa, jossa kaikki kirjoitusoperaatiot kerätään ja lähetetään tapahtuman lopussa yhtenä pakettina kaikille noodeille. Myös kaikki tarvittavat lukot kirjoitettaville tiedoille haetaan jokaisella noodilla heti kirjoitusoperaation alussa. Viimeistään vahvistusvaiheessa päivityksen tiedot lähetetään muille noodeille varmistettavaksi ja mikäli ristiriitoja ei ilmene, noodit myös vahvistavat päivityksen [Pedone 1999].

Jossakin tilanteissa päivityskyselyt ovat *kommutatiivisia* (commutative updates), jolloin päädytään aina samaan tietokannan tilaan riippumatta kyselyjen ajojärjestyksestä. Tällaisia kyselyjä olisivat esimerkiksi vakioiden lisääminen ja vähentäminen kokonaisluvusta [Gray et al. 1996].

Jatkossa oletetaan, että kaikki replikointiin liittyvät datanoodit ovat jatkuvasti saatavilla. Mikäli osa noodeista olisi ajoittain poissa verkosta, voitaisiin käyttää esimerkiksi kaksiportaista replikointia, jolloin yhteydettömillä noodeilla ajetaan *alustavia tapahtumia* (tentative transactions). Nämä tapahtumat ajetaan uudestaan varsinaisina tapahtumina, kun noodi saa yhteyden verkkoon ja ne voivat vielä tässä vaiheessa myös epäonnistua. [Gray et al. 1996]

Kuuman valmiuden palvelin (hot standby) on replikapalvelin, jonka tila pidetään reaaliaikaisesti isäntätietokannan tasalla. Isäntäpalvelimen toiminnan keskeytyessä ottaa valmiuspalvelin välittömästi isäntäpalvelimen roolin. Monissa järjestelmissä kuuman valmiuden palvelin myös palvelee lukukyselyitä. *Lämpimän valmiuden palvelinta* (warm standby) ei päivitetä reaaliajassa, vaan tietyin aikavälein. Myös lämpimän valmiuden palvelin palvelee joissakin järjestelmissä lukukyselyitä. [PostgreSQL 2018] [SQL Server Warm Standby 2012]

4.3. Partitiointi

Partitioinnissa tietokannan taulu jaetaan eri tietokantanoodien kesken. Mikäli taulun rivit jaetaan eri noodien kesken, puhutaan *horisontaalisesta partitioinnista*. Taulun sarakkeita jaettaessa on kyseessä *vertikaalinen partitiointi*. Koska horisontaalinen partitiointi tietokantatasolla on huomattavasti yleisempi ominaisuus, jatkossa tarkoitan partitioinnilla nimenomaan horisontaalista partitiointia. Taulun jaettuja osia kutsutaan *partitioiksi* tai *fragmenteiksi*. Mikäli partitioinnissa syntyneiden partitioiden jakoa noodeille käsitellään erillisenä ongelmana, puhutaan *tiedon sijoittamisesta* (allocation).

Vertikaalisessa partitioinnissa jokaiselle partitiolle tallennetaan valittujen sarakkeiden lisäksi myös pääavaimen sarakkeet, jotta rivit voidaan yhdistää yksiselitteisesti. Tallentamalla esimerkiksi hyvin suurien sarakkeiden tai harvaan käytettyjen sarakkeiden tiedot toiseen partitioon, saadaan yleisemmin käytetyn partition

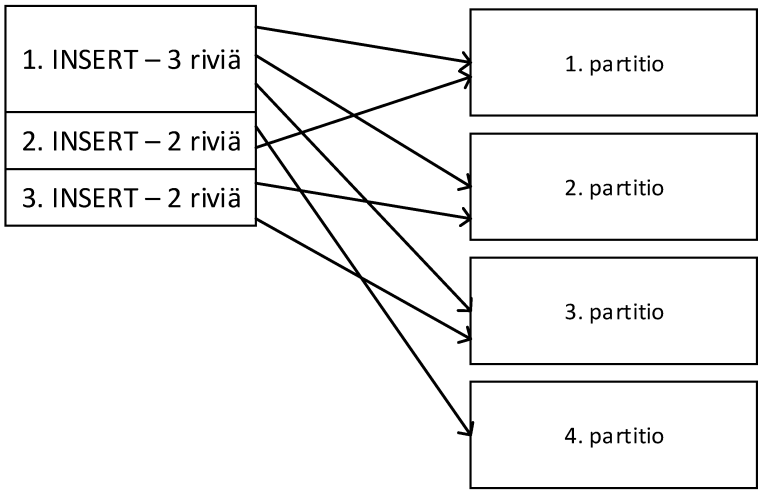
kokoa pienemmäksi. Yleensä kaupallisissa tietokannoissa ei ole tukea vertikaaliselle partitioinnille, mutta sitä voidaan mallintaa luomalla tavalliset taulut jokaiselle partitiolle suunnitellun jaon mukaan [Agrawal et al. 2004]. Tietokantanäkymän avulla jaetun taulun sisällöt voidaan saada jälleen näkymään yhtenäisesti. Vertikaalisen partitioinnin tavoin tietokantaa normalisoitaessakin taulu jaetaan, mutta siinä jakamisen tavoitteet eivät liity erityisesti suorituskykyyn, vaan kannan rakenteen eheyteen. Partitiointia tehtäessä taulujen tietorakenne on jo suunniteltu ja mahdollinen normalisointi tehty.

Partitiointiskeema sisältää tiedot eri noodien sisältämästä datasta ja sen avulla voidaan kyselyt ohjata oikeille noodeille. *Täydessä partitioinnissa* (full partitioning) taulun data jaetaan kaikkien järjestelmän noodien kesken [Özsu and Valduriez 2011]. *Muuttuvassa partitioinnissa* (variable partitioning) partition koko määritellään taulun koon ja käyttöasteen mukaan, mikä voi johtaa ajan kuluessa taulun *uudelleenpartitiointiin* eli partitiointiskeeman muutokseen ja taulun tietojen siirtoon uuden skeeman mukaiseksi [Copeland et al. 1988].

Yksinkertaiseen datan partitiointiin on kolme mahdollisuutta: *kiertovuorottelupartitiointi* (round-robin partitioning), *arvovälipartitiointi* (range partitioning) ja *hajautuspartitionti* (hash partitioning). Kiertovuorottelussa tauluun lisättävät rivit jaetaan järjestyksessä partitioiden kesken. Jos siis partitioita on *n* kappaletta, niin lisäysjärjestyksessä rivi numero *i* annetaan partitiolle, jonka numero on *i jakojäännös n*. Mikäli `film`-taulun hajautukseen valittaisiin kiertovuorottelupartitiointi, ja ensimmäiset lisäyslauseet lisäisivät rivejä seuraavasti:

1. lisäyslause: 3 riviä
2. lisäyslause: 2 riviä
3. lisäyslause: 2 riviä

lisättäisiin tällöin ensimmäisen lauseen 3 riviä partitioille 1-3, toisen lisäyksen 2 riviä partitioille 4 ja jälleen partitiolle 1 jne. (kuva 5).

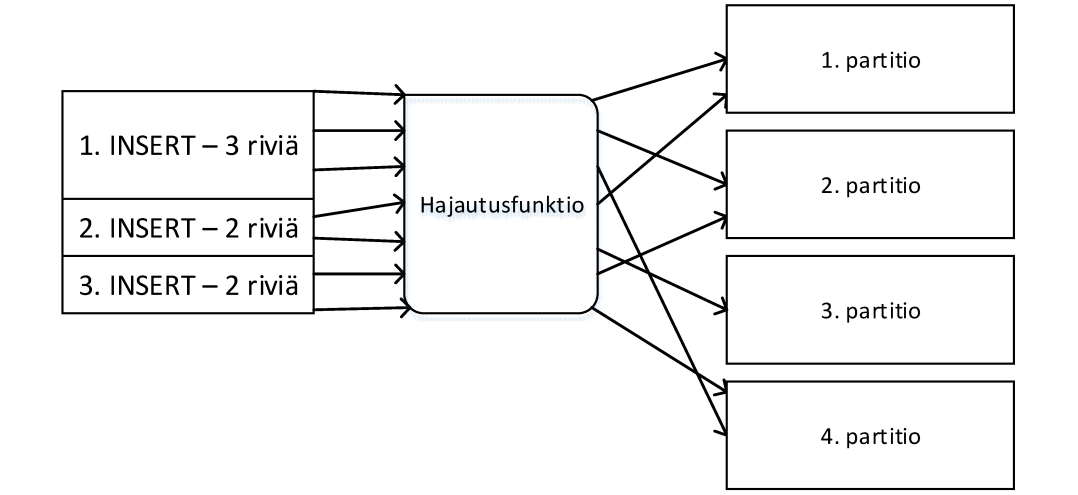


Kuva 5. Kiertovuorottelupartitiointi – rivit lisätään vuorotellen noodeille lisäysjärjestyksessä.

Arvovälipartitioinnissa valitaan arvovälit jostain sarakkeesta, kuten taulun ID-sarakkeen arvoista. Esimerkiksi `film`-taulun 1000 rivin sisältö voidaan arvovälipartitioinnilla partitioida neljän noodin kesken, jolloin jokaisella noodilla on 250 riviä (kuva 6). Jos partitioinnin pohjana käytetään taulun saraketta, kutsutaan tätä *partitiointisarakkeeksi* tai *-attribuutiksi*.

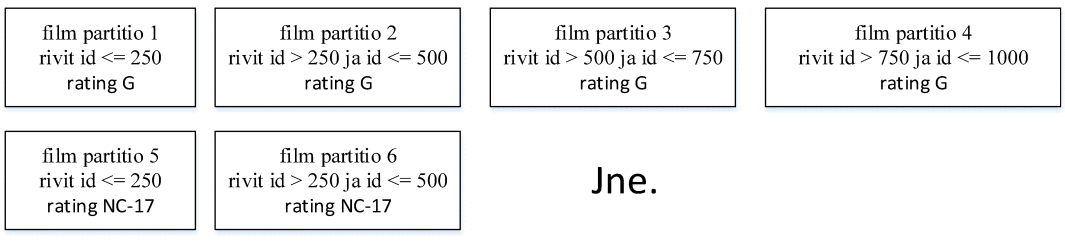
Kuva 6. Arvovälipartitiointi

Hajautuspartitioinnissa hajautusfunktion tulos määrittelee rivin partition. Yleensä funktiolle pitää määritellä parametriksi joku taulun kentistä. Tämän sarakkeen perusteella funktio palauttaa järjestelmän noodin, johon rivi lisätään. Esimerkiksi MySQL:ssä funktiolle annettavan parametrin pitää olla muodoltaan kokonaisluku ja luvun pitää olla vaihteleva, mutta ei satunnainen [MySQL Ref. Manual 2018]. Tähän tarkoitukseen sopii usein taulun ID-sarake. Kuvassa 7 on havainnollistettu hajautusfunktion toimintaa.



Kuva 7. Hajautuspartitiointi – valittu hajautusfunktio määrittelee rivien sijoittelun.

Periaatteessa sekä vuorovälipartitioinnissa että hajautuspartitioinnissa voidaan käyttää myös useampaa saraketta parametrina. Esimerkiksi `film`-taulun toiseksi parametriksi voitaisiin valita elokuvaikärajoja kuvaava `rating`-sarake, josta löytyy viittä eri arvoa: G, PG, PG-13, R ja NC-17. Mikäli tätä saraketta käytetään partitiointiin siten, että vain yhtä arvoa löytyy yhdeltä partitiolta, tarvitaan siis viisi partitiota. Partitioimalla sekä edellä kuvatun `film_id`-arvovälin että `rating`-sarakkeen arvojen mukaan saataisiin yhteensä 4 * 5 = 20 partitiota (kuva 8). Usein yhtä attribuuttia käyttämällä saavutetaan kuitenkin parempia tuloksia elleivät valitut sarakkeet esiinny säännönmukaisesti kyselyissä yhdessä. Esimerkiksi kysely `SELECT * FROM film WHERE film_id = 25;` jouduttaisiin lähettämään kaikille niille noodeille, joissa partitioinnin perusteella voi `film_id` olla 25, koska rivin `rating`-sarakkeen arvoa ei tiedetä (kuvassa näkyvistä partitioista partitiot 1 ja 5).



Kuva 8. Ositus kahden sarakkeen perusteella, yhteensä partitioita muodostuisi 20 kappaletta.

Ensisijainen horisontaalinen partitiointi (primary horizontal fragmentation) suoritetaan käyttämällä taululle määriteltyjä partitioinnin predikaatteja, esimerkiksi ID-numeron arvovälejä. Jos taulun partitiointi tehdään siihen liittyvän toisen taulun partitioinnin perusteella, on kyseessä *johdettu horisontaalinen partitiointi*.

Esimerkki 2: Jos taulu `film` *partitioidaan arvoväli-partitioinnilla kuvan 6 mukaisesti, voidaan taulu* `inventory` *partitioida vastaavasti* `film_id`*-vierasavaimen perusteella:*

`Inventory`*, partitio 1:* `film_id` *1-250 – yhteensä 1130 riviä*

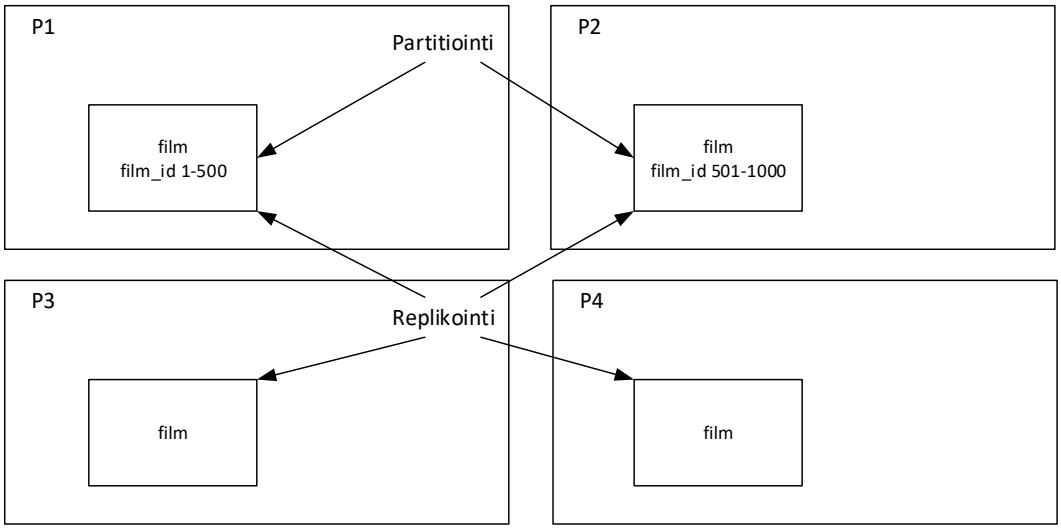
`Inventory`*, partitio 2:* `film_id` *251-500 – yhteensä 1173 riviä*

`Inventory`*, partitio 3:* `film_id` *501-750 – yhteensä 1122 riviä*

`Inventory`*, partitio 4:* `film_id` *751-1000 – yhteensä 1156 riviä*

Tässä esimerkissä myös johdetulla partitioinnilla partitioitu taulu jakaantuu harvinaisen tasaisesti neljälle noodille. Aina näin ei kuitenkaan ole, vaan tietokannan datasta ja rakenteesta riippuen johdettu partitiointi voi myös johtaa rivien epätasaiseen jakautumiseen noodien välillä.

Partitioinnin voi myös yhdistää replikoinnin kanssa joko replikoimalla noodit, replikoimalla taulupartitiot, tai yhdistämällä taulujen replikointia ja partitiointia keskenään. Kuvassa 9 tietokantapartitioiden P1-P4 välillä toimii hajautettu taulutason replikointi. Taulu `film` on partitioitu arvovälipartitioinnilla noodeille P1 ja P2, mutta sama taulu on replikoitu osittamattomana myös noodeille P3 ja P4. Mille tahansa noodille tuleva päivitys tauluun replikoidaan kaikille niille noodeille, joissa kyseinen rivi sijaitsee. Esimerkiksi päivitys riville `film_id` = 510 ajetaan noodeilla P2, P3 ja P4.



Kuva 9. Yhdistetty partitiointi ja replikointi – taulu `film` löytyy osittain noodeilta P1 ja P2 sekä kokonaan noodeilta P3 ja P4.

Partitiointia käytetään usein myös normaaleissa hajauttamattomissa tietokantajärjestelmissä pienentämään taulujen ja siihen liittyvien indeksien kokoa. Tällä saadaan parannettua järjestelmän suorituskykyä (katso esimerkiksi Agrawal et al. [2004]). Useimmissa kaupallisissa tietokantajärjestelmissä onkin tuki tämän tyyppiselle partitioinnille.

4.4. Kyselyiden käsittely ja ohjaus noodeille

Tietokantajärjestelmien kyselyiden käsittelyssä pyritään hakemaan tietokannasta kyselyn tarvitsema data mahdollisimman pienillä lukuoperaatioilla. Kyselynkäsittelijä tekee muun muassa päätöksen indeksin käytöstä ja tavasta yhdistää eri tauluista saatuja tulosjoukkoja. Hajautetussa tietokannassa kyselynkäsittelijän pitää tietää, miltä noodilta tarvittava tieto löytyy, jolloin kysely voidaan ohjata oikealle partitiolle. Mikäli kyseessä on replikoitu tieto, valitaan yksi replikoidun taulun tallentavista noodeista tiedon hakuun. Yleensä tällöin valitaan kuormantasauksen vuoksi vähiten kuormitettu noodi. Monet järjestelmät käyttävät erillisiä kuormantasaajia, joihin asiakasohjelmistot ottavat yhteyttä ja jotka reitittävät kyselyt tai kyselyjen osat eteenpäin eri partitioille.

Mikäli kysely hajautetaan eri noodeille, jaetaan kysely pienemmiksi *alikyselyiksi*, jotka ajetaan eri noodeilla, ja joiden tulosjoukot yhdistämällä muodostetaan lopullinen tulosjoukko. Mikäli erillisiä kyselynkäsittelijöitä tai kuormantasaajia ei käytetä, sisältää jokainen noodi tiedot partitiointiskeemasta. Näin toimii esimerkiksi H-Store. Tällöin kysely saapuu ensin *pohjanoodille* (base partition), joka huolehtii kyselyn ajamisesta ja alikyselyiden lähettämisestä muille noodeille. Jos koko kysely ajetaan yhdellä partitiolla,

eikä tietoja tarvitse hakea muilta partitioilta, on kyseessä *yhden noodin tapahtuma* (single partition transaction). [Pavlo et al. 2011]

OLAP-järjestelmissä pyritään usein hyödyntämään kyselyn sisäistä rinnakkaisuutta. Tällöin voidaan käyttää *kaksivaiheista kyselyn optimointia* (two-phased query optimization): ensimmäisessä vaiheessa väliohjelmisto valitsee kyselyn suorittavat noodit ja jakaa kyselyn näiden kesken. Toisessa vaiheessa noodit valitsevat optimaalisen kyselysuunnitelman omalle osalleen kyselystä. [Akal et al. 2002]

Rinnakkaisuuden hallintaan monet hajautetut järjestelmät käyttävät lukkojen sijaan vaihtoehtoisia tapoja, esimerkiksi *spekulatiivista rinnakkaisuuden hallintaa*, jolloin jonossa olevia tapahtumia suoritetaan etukäteen ja tarpeen tullen peruutetaan [Jones et al. 2010]. Joidenkin määritelmien mukaan NewSQL-ratkaisuiksi lasketaankin vain järjestelmät, joissa ei käytetä lukkoja rinnakkaisuuden hallinnassa [Pavlo and Aslett 2016].

Lukkoja käytettäessä voidaan 2-vaiheista lukitusta hallita hajautetun järjestelmän yhteydessä keskitetysti yhdellä noodilla tai vaihtoehtoisesti jokaisella noodilla voi toimia oma lukkojen hallinta. Edellisessä tapauksessa, jokainen noodi hoitaa tapahtumien lukitukseen liittyvän kommunikoinnin tämän lukkojenhallintanoodin kanssa [Alsberg and Day 1976]. Tässä ratkaisussa lukkoja hallitsevasta noodista muodostuu helposti pullonkaula. Järjestelmän toiminta on myös alttiimpi virheille kyseisessä noodissa.

Hajautetussa 2-vaiheisessa lukituksessa lukkojen hallinta tapahtuu jokaisella noodilla. Tällöin datan käsittelyn käskyt noodille tulevat kyseisen noodin lukkojenhallinnan kautta. Lukkoihin liittyvät viestit lähtevät suoraan lähettävältä noodilta muille noodeille. Lukkojen varaushetki voi myös vaihdella. Mikäli kysely kuitenkin varaa lukot kaikilta noodeilta ennen kyselyn varsinaista ajamista, vältytään lukkiumilta [Kemme and Alonso 2000].

Kuormantasaus (load balancing) tarkoittaa kyselyiden ohjaamista noodeille siten, että kuorma jakautuu tasaisesti palvelinten kesken. Yksinkertainen tasauksen malli on johtaa kysely aina vähiten kuormitetulle noodille. Lukukyselyjen kuormantasaus replikoitujen palvelimien välillä voidaan toteuttaa myös verkkoprotokollan tasolla siten, että kuormantasaaja tarkkailee ainoastaan palvelimien kuormaa, ja kyselyjen reititys tapahtuu reitittämällä kyselyt verkossa eteenpäin oikeille noodeille [Kaitsa et al. 2007].

Hajautetuissa järjestelmissä pyritään siihen, että kyselyt jakautuisivat mahdollisimman tasaisesti palvelinten kesken. Epätasaista jakautumaa kutsutaan *vääristymäksi*. Vääristymää voi ilmetä sekä tiedon jakautumisessa noodeille että tietoa käsittelevässä kuormassa. Vääristymä rivien jakautumisessa on korjattavissa ainoastaan siirtämällä dataa noodilta toiselle [Sadat and Lecca 2009]. Työkuorman vääristymää voidaan pyrkiä pienentämään kuormantasaajan toiminnalla.

Monesti OLTP-sovellusten transaktiot ottavat huomattavasti enemmän yhteyksiä pieneen osaan taulun riveistä. Esimerkiksi New Yorkin pörssin tapahtumista 40 - 60% kohdistuu 40 yrityksen osakkeisiin, kun kohteita on yhteensä 4000 [Nazaruk and Rauchman 2013]. Tällöin nämä yksittäiset rivit kannattaa jakaa eri tietokantapartitioihin kuorman tasoittamiseksi. Kuorma voi myös muuttua vuorokauden ajan tai vuodenajan mukaan (*ajallinen vääristymä*). Esimerkiksi eri puolilla maailmaa käytetyn järjestelmän yhteen maahan liittyviin riveihin saattaa syntyä eniten kuormaa silloin, kun kyseisessä maassa on päivä.

4.5. Kyselyjen rinnakkainen ajo

Rinnakkaisuus kyselyiden ajossa tapahtuu joko kyselyn sisällä tai kyselyiden välillä. *Kyselyiden välisessä rinnakkaisuudessa* (inter-query parallelism) useampi kysely ajetaan samanaikaisesti järjestelmässä. *Kyselyiden sisäisessä rinnakkaisuudessa* (intra-query parallelism) kyselyn eri operaatiot ajetaan rinnakkain. Tällöin voidaan käyttää sekä operaattorien välistä että niiden sisäistä rinnakkaisuutta. Mikäli järjestelmä on replikoitu ja useampi noodi palvelee kyselyitä, voidaan käyttää hyväksi sekä kyselyjen sisäistä että kyselyjen välistä rinnakkaisuutta. Partitioidut taulut lisäävät kyselyjen sisäistä rinnakkaisuutta, mutta hyödyntävät kyselyjen välistä rinnakkaisuutta vain, mikäli kyselyt käsittelevät taulujen eri partitioilla sijaitsevia rivejä. OLTP-järjestelmät, joissa kyselyt ovat reaaliaikaisia ja lyhyitä, hyödyntävät tyypillisesti kyselyjen välistä rinnakkaisuutta. OLAP-järjestelmissä kyselyt ovat puolestaan monimutkaisia ja niiden ajoaika on pitkä, joten niissä pyritään usein kyselyjen sisäiseen rinnakkaisuuteen. Koska noodien rinnakkaisuuden lisääminen lisää myös kommunikaatiota noodien välillä, se kannattaa vain raskaampien kyselyiden kohdalla.

Käytettäessä *operaattorin sisäistä rinnakkaisuutta* (intra-operator parallelism) ajettava operaattori jaetaan osiin, jotka ajetaan rinnakkain eri noodeissa tai suorittimissa. Esimerkiksi yksinkertainen SELECT-lause voidaan ajaa usealla partitiolla, minkä jälkeen tulokset kootaan yhteen. Mikäli WHERE-osiossa on käytetty ehtoja, jotka ovat myös partitioinnin pohjana, voidaan tämän perusteella osa partitioista jättää pois kyselystä.

Operaattorien välinen rinnakkaisuus voidaan jakaa *johdannaiseen* (pipeline) ja *riippumattomaan rinnakkaisuuteen* (independent parallelism). Johdannaisessa rinnakkaisuudessa kahdesta ajettavasta operaattorista toinen on riippuvainen ensimmäisen tuloksista. Riippumattomassa rinnakkaisuudessa ajettavat operaattorit eivät liity toisiinsa, jolloin niiden ei myöskään tarvitsee odottaa toistensa tuloksia. [Özsu and Valduriez 2011]

4.6. Hajautuksessa käytettyjen tekniikoiden hyödyt

Taulukko 3 havainnollistaa eri hajautusratkaisujen hyödyt suhteessa skaalautuvuuteen, saatavuuteen ja kuormantasaukseen. Replikointi on jaettu täyteen ja osittaiseen replikointiin, koska näiden hyödyt jakautuvat eri tavoin: täysi replikointi vastaa saatavuuden ongelmiin ja lisäksi lukukyselyjen kuormaa voi hajauttaa replikoille. Skaalautuvuuteen palvelimen replikointi ei kuitenkaan auta, koska replikapalvelimien lukumäärän kasvaessa kasvaa myös päivityskyselyiden aiheuttama kuorma. Osittaisessa replikoinnissa voidaan rajoittaa replikointi vain *lukuintensiivisiin tauluihin* eli tauluihin, joihin tehdään enimmäkseen lukukyselyjä, eivätkä päivityskyselyt tuota silloin samanlaista ongelmaa. Tällöin tietokannan hajautuksessa käytetään replikoinnin ja partitioinnin yhdistelmää. Koska vain osa sisällöstä kopioidaan, ei osittainen replikointi yleensä paranna järjestelmän saatavuutta.

Partitiointi jakaa tauluun kohdistuvaa kuormaa eri palvelinten kesken. Lisäksi suuren taulun hajauttaminen pienentää tilantarvetta yksittäisillä noodeilla, jolloin noodeja lisäämällä voidaan kasvattaa järjestelmän tallennuskapasiteettia. Koska noodeja lisäämällä saadaan myös käytettävissä olevan keskusmuistin kokonaismäärää lisättyä, voidaan tavallisia palvelimia lisäämällä suurempi osa tietokannan sisällöstä ladata muistiin.

Taulukko 3. Hajautuksessa käytettyjen tekniikoiden hyödyt suhteessa skaalautuvuuteen, korkeaan saatavuuteen ja kuormantasaukseen.

	Skaalautuvuus	Saatavuus	Kuormantasaus
Täysi replikointi		x	x
Osittainen replikointi	x		x
Partitiointi	x		x
Kyselyjen rinnakkaisuus	x		x

5. Olemassa olevia hajautettuja järjestelmiä

Tällä hetkellä saatavilla olevat hajautetut tietokannat voidaan karkeasti jakaa tutkimuskäyttöön kehitettyihin järjestelmiin ja toisaalta tuotantokäyttöön tarkoitettuihin järjestelmiin. Tutkimuskäytössä olevien järjestelmien yhteydessä on usein tehty järjestelmän optimoinnissa kaupallisia ratkaisuja pidemmälle vietyjä kokeiluja, jotka eivät välttämättä toimisi normaalikäytössä. Yleisimpien kaupallisten tietokantajärjestelmien replikaatio- ja partitiointitoiminnallisuudet toteuttavat puolestaan vain rajoitetun määrän ominaisuuksia. Monestikaan nämä ominaisuudet eivät riitä horisontaalisesti skaalautuvan ratkaisun toteuttamiseen. Tässä luvussa käydään läpi tuotanto- tai tutkimuskäyttöön tehtyjä järjestelmiä, jotka toteuttavat vähintäänkin osan edellisen luvun hajautuksen periaatteista.

5.1. Tutkimuksissa käytettyjä tai niihin kehitettyjä järjestelmiä

Tietokantojen tutkimuksen yhteydessä kehitetyistä järjestelmistä osa on rakennettu yleisempään käyttöön ja osa ainoastaan yksittäisten tutkimusten tarpeisiin. Monen kehitys on loppunut tutkimukseen liittyvien artikkelien julkaisun jälkeen. Taulukkoon 4 on koottu tässä tutkielmassa mainituissa artikkeleissa käytettyjä järjestelmiä ja niiden viimeisimmät julkaisuvuodet.

H-Store on useiden yliopistojen kehittämä hajautettu tietokanta tutkimuskäyttöön [Kallman et al. 2008]. Kyselyt lähetetään suoraan H-Storen noodeille, mutta ainoastaan tallennettujen proseduurien ajoa tuetaan, millä vähennetään liikennettä järjestelmän ja asiakasohjelmiston välillä. H-Storea on käytetty ja sitä on laajennettu useissa rinnakkaisuuteen ja hajautukseen liittyvissä tutkimuksissa (esimerkiksi Pavlo et al. [2012], Elmore et al. [2011] ja Taft et al. [2014]). H-Storen aktiivinen kehitys on toistaiseksi lopetettu [H-Store 2018]. VoltDB on H-Storen arkkitehtuuriin perustuva kaupallinen tietokantajärjestelmä [H-Store 2018].

Taulukko 4. Tutkimuksissa käytettyjä tietokantajärjestelmiä

Tietokantajärjestelmä	Käyttötarkoitus	Viimeisin versio
H-Store	OLTP	2016
Sequoia	OLTP	2009
Peloton	OLTP/OLAP	ei virallista versiota, aktiivisessa kehityksessä
ParGRES	OLAP	2015
PargreSQL	OLAP	ei versiota saatavilla
Dtxn	OLTP/OLAP	ei versiota saatavilla
PowerDB	OLAP	ei versiota saatavilla
cgmOLAP	OLAP	ei versiota saatavilla
SmaQSS	OLAP	ei versiota saatavilla
OLAP*	OLAP	ei versiota saatavilla

Jones [2012] esittelee *Dtxn*:n, hajautetun järjestelmän, joka ei itsessään sisällä datanoodeja, vaan hallinnoi erillisiä noodeina toimivia tietokantapalvelimia. Järjestelmään liitettävien noodien pitää toteuttaa Dtxn:n vaatima rajapinta, joka määrittelee tietokantajärjestelmän perusoperaatiot. Käytännössä Dtxn käsittelee datanoodeja mustina laatikkoina, joiden sisältö tai toteutustapa ei ole sen tiedossa. Tapahtumat koostuvat *operaatiofragmenteista* – operaatioista, jotka välitetään noodin sisältämälle tietokantajärjestelmälle. Dtxn ympäröi tietokannan partitiopalvelimella, joka huolehtii tiedon kestävyydestä, verkkoliikenteestä ja tietokannan tapahtumista. Tapahtumienhallintaa varten partitiopalvelimella on vuorottaja (scheduler). Jones on testannut Dtxn-järjestelmää esimerkiksi korvaamalla VoltDB-tietokannan sisäänrakennetun tapahtumaprotokollan Dtxn:n vastaavalla.

ETH Zürich The Database Research Group on ylläpitänyt 2000-luvun alussa PowerDB-projektia, jonka tarkoituksena on ollut sekä OLAP- että OLTP-kyselyjä palveleva, automaattisesti skaalautuva ja hajautussuunnitelmaa muuttava klusteri [Röhm et al. 2000] [Akal et al. 2002]. Järjestelmässä yksittäisen noodin sisällä toimii Dtxn:n tavoin normaali tietokantajärjestelmä, jossa ei ole hajautusominaisuuksia, vaan hajautus toteutetaan täysin väliohjelmiston avulla [PowerDB 2018]. PowerDB:n noodien tietokantajärjestelmänä on käytetty muun muassa PostgreSQL:ää [Lima et al. 2004].

ParGRES on väliohjelmisto kyselyiden ajamiseksi rinnakkain OLAP-ympäristössä [Paes et al. 2008]. Sitä ei pidä sekoittaa Venäjällä kehitettyyn PargreSQL tietokantaan [Pan and Zymbler 2013]. ParGRESiin on toteutettu myöhemmässä luvussa esiteltävä virtuaalinen partitiointi. Ratkaisua on testattu sekä TPC-H suorituskykytestillä että

käyttämällä Brasilian väestönlaskennasta kerättyä dataa testiaineistona, ja ajamalla sitä vasten OLAP-kyselyjä [Paes et al. 2008]. Näissä testeissä on noodeja lisäämällä saatu huomattavasti parannettua kyselyjen ajonopeutta.

5.2. Kaupalliset tietokantajärjestelmät

Useimmista suosituimmista tietokantajärjestelmistä löytyy jonkintasoinen rinnakkaisuuden tuki. Eri järjestelmien tarjoamat ominaisuudet on koottu lyhyesti taulukkoon 5. Taulukkoon on merkitty suluilla ominaisuudet, joita tuetaan vain osittain: MySQL NDB Clusterin voi asettaa replikoitumaan palvelintasolla, mutta ratkaisu on monimutkaisempi kuin täysi replikointi normaalisti [MySQL Ref. Manual 2018]. IBM DB2:n osittainen replikointi puolestaan toimii vain linkittämällä replikoitava taulu näkymien kautta toisille noodeille [IBM DB2 2018] ellei käytetä erillistä IBM InfoSphere Data Replication -ratkaisua [IBM InfoSphere 2017]. Jaetun levyn järjestelmät Oracle RAC ja IBM DB2 pureScale on jätetty pois taulukosta. Avoimen lähdekoodin järjestelmistä (PostgreSQL ja MySQL) löytyy myös epävirallisia hajautusta varten tehtyjä versioita (esimerkiksi Postgres-XL ja Galera) [Postgres-XL 2018] [Galera 2018].

Taulukko 5. Suosituimpia tuotantokäytössä olevia tietokantajärjestelmiä ja niiden hajautusominaisuudet.

Tietokanta-järjestelmä	Täysi replikointi	Osittainen replikointi	Partitiointi
Microsoft SQL Server	x		
MySQL	x		
MySQL NDB Cluster	(x)	x	x
Galera Cluster	x		
PostgreSQL	x		
IBM DB2	x	(x)	x
Oracle (Data Guard)	x		
Oracle (Streams)		x	
MaxDB	x		
MariaDB	x		
VoltDB	x	x	x

5.2.1. PostgreSQL

PostgreSQL on vapaan lähdekoodin tietokantajärjestelmä, joka on 1970-luvulla syntyneen Ingres-tietokannan seuraaja. Sen kehitys alkoi vuonna 1986. 2000-luvun ensimmäisellä vuosikymmenellä hajautusominaisuudet kuten replikaatio olivat rajattuina pois ydintuotteen kehityksestä, mutta tämä on muuttunut viime vuosina. Samanaikaisuuden hallintaan PostgreSQL käyttää lukkojen sijasta oletuksena *moniversioivaa samanaikaisuuden hallintaa* (MVCC - multiversion concurrency control) [PostgreSQL 2018]. Ratkaisussa vanhentuneet taulujen rivien versiot jäävät yhä tallennetuiksi tietokantaan, mutta ne eivät enää näy tapahtumille. Kun version vanhenemishetkellä pyörineet tapahtumat on kaikki vahvistettu tai peruttu, voidaan vanhentunut versio sopivalla hetkellä poistaa [Kapila 2015]. Koska ydintuotteesta puuttuivat pitkään olennaiset hajautusominaisuudet, on PostgreSQL:n ympärille syntynyt useita tietokantatason replikaatioon tai tietokantaklusteriin tähtääviä projekteja, joista moni on sittemmin lopetettu, kuten PgCluster ja PostgreSQL-X2 [PostgreSQL Wiki 2018].

PostgreSQL on tarjonnut 9.0 versiosta lähtien tietokannan perusominaisuuksina virtaavan replikoinnin (streaming replication) ja lämpimän valmiuden replikoinnin. Virtaavan replikoinnin mallissa PostgreSQL:n tapahtumalokin (PostgreSQL:ssä tästä käytetään nimitystä *WAL-XLOG - Write-Ahead Log*) tietoja lähetetään jatkuvasti valmiuspalvelimille, jotka päivittävät tilansa sen mukaan. Tietoja lähetetään, vaikka tapahtumaloki ei olisi vielä täynnä (tapahtumaa ei ole vahvistettu). Oletuksena replikointi on asynkronista, mutta se voidaan asettaa myös synkroniseksi. Lämpimän valmiuden mallissa tapahtumaloki lähetetään vasta kun se on täytetty kokonaan. Näihin replikoinnin malleihin liittyy myös kuuman valmiuden mahdollisuus, jota käyttämällä replika hyväksyy lukukyselyt.

PostgreSQL:n *kaksisuuntainen replikointi* (*BDR* - Bi-Directional Replication) tarjoaa hajautetun replikaatioratkaisun. BDR vaatii muunnellun version PostgreSQL 9.4-versiosta. Sen ominaisuuksia siirretään vähitellen osaksi PostgreSQL:n normaalia tuotantoversiota [Ringer 2015]. Postgres-XL on PostgreSQL:n versio, joka tukee muun muassa OLAP-perusteisia toimintoja ja kyselyjen hajauttamista kaikille klusterin koneille [Postgres-XL 2018]. Sekä BDR:ää että PostgreSQL-XL:ää kehittää 2ndQuadrant.

Postgres-XL:ssä varsinaiset taulut on tallennettu datanoodeihin. *Globaali tapahtumienhallinta* (GTM - Global Transaction Manager) hallinnoi tapahtumien ID-numeroita ja tilannevedoksia osana MVCC:tä. Lisäksi se huolehtii globaaleista arvoista kuten aikaleimoista ja sekvensseistä. Kyselyiden suunnittelu tapahtuu *koordinaattoreissa*, jotka myös kommunikoivat asiakasohjelmiston, GTM:n ja datanoodien kanssa. GTM:lle

suositellaan omaa palvelinta, mutta koordinaattorit suositellaan sijoitettavaksi samoille palvelimille datanoodien kanssa. Postgres-XL tarjoaa hajautuspartitionnin, kiertovuorottelupartitionnin ja jakojäännökseen perustuvan partitionnin (modulo distribution). Käyttäjä määrittää käytetyn hajautuksen mallin `CREATE TABLE`-lauseen yhteydessä. Samalla voi määrittää noodit, joihin data tallennetaan. Oletuksena data partitioidaan kaikkien noodien välillä. [Postgres-XL 2018]

Pgpool-II on PostgreSQL:ää varten rakennettu kuormantasaaja ja yhteyksien rajoittaja. Lisäksi se ylläpitää tietokantayhteyksiä palvelimelle asiakasohjelmistojen puolesta, jolloin kaikkia yhteyksiä voidaan hallinnoida keskitetysti. Se toimii asiakasohjelmiston ja tietokantapalvelimien välissä. Kuormantasaajana se jakaa lukukyselyjä tasaisesti eri noodien kesken. Se voi myös tarjota korjausmekanismin, mikäli joku tietokannoista ei ole enää tavoitettavissa. Pgpool sisältää myös oman replikointiratkaisunsa, jossa on mahdollista käyttää synkronista kaksisuuntaista replikointia. [pgpool-II 2018]

5.2.2. MySQL

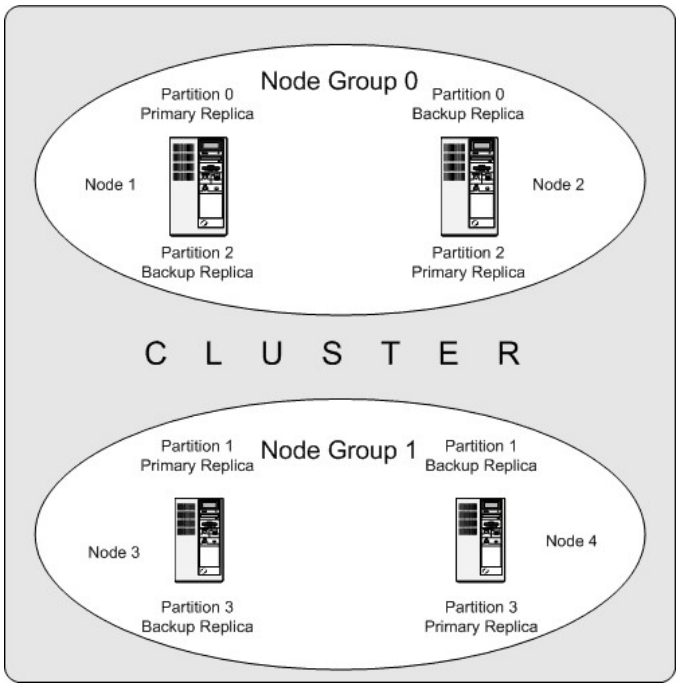
Ruotsalainen MySQL AB julkaisi MySQL:n ensimmäisen version vuonna 1995 ja se on usean arvion mukaan yhä suosituin vapaan lähdekoodin tietokantajärjestelmä [solidIT 2018] [Stack Exchange 2018]. Oracle Corporation hankki MySQL:n omistukseensa vuonna 2010. Tiedon tallennukseen fyysisellä tasolla MySQL tarjoaa useita *tallennusmoottoreita* (storage engines), joista oletuksena käytössä on InnoDB. Hajautettuun tiedon tallennukseen ovat tarjolla moottorit NDB, Merge ja Federated [MySQL Ref. Manual 2018]. Oletusmoottori InnoDB käyttää samanaikaisuuden hallintaan PostgreSQL:n tavoin moniversioivaa samanaikaisuuden hallintaa.

MySQL:ssä replikoinnissa käytetään yleisesti *binäärilokia* (binary log). Mikäli kirjoitus binäärilokiin on päällä, kirjataan siihen kaikki dataa muuttavat tapahtumat tietokannassa. MySQL:n perusversioon sisältyvä replikointi on perinteisesti tapahtunut replikoimalla tapahtumat isäntänoodin binäärilokilta. Tällöin binäärilokitiedostot synkronoidaan replikoille. Uudempi järjestelmä perustuu *globaaleille transaktiotunnuksille* (global transaction identifiers – GTIDs), joilla tapahtumat voi yksilöidä koko järjestelmän laajuudella. MySQL tarjoaa asynkronista replikointia ja versiosta 5.7 eteenpäin myös *puolisynkronista replikointia*, jossa vähintään yhden orjanoodin pitää todennetusti vastaanottaa tapahtuman tiedot ennen kuin sessio voi jatkaa. Replikointi voi perustua lauseille tai riveille (vain muutetut rivit replikoidaan) tai olla näiden välimuoto (mixed replication). [MySQL Ref. Manual 2018]

MySQL:n Cluster NDB versio tarjoaa klusterin, jossa on mahdollista käyttää hajautettua synkronista replikointia ja partitiointia. NDB-replikoinnissa pitää noodien

käyttää NDB-tallennusmoottoria (myös InnoDB-tallennusmoottorin päälle on kehitetty hajautettu ratkaisu: korkean saatavuuden replikointiratkaisun tarjoava InnoDB-klusteri). NDB on lyhennys sanoista Network Database, mutta sillä viitataan nimenomaan tietokantamoottoriin. MySQL Clusterin arkkitehtuuri koostuu datanoodeista, *SQL noodeista*, jotka toimivat käytännössä kuormantasaajina ja *NDB-ohjainpalvelimesta* (NDB Management Server). Normaalisti noodien väliseen kommunikointiin käytetään TCP/IP-protokollaa; tosin myös vanhempi SCI-protokolla on mahdollinen. NDB:n noodit ovat prosesseja, jotka voivat sijaita samalla tai eri koneilla. [MySQL Ref. Manual 2018]

Ohjainpalvelin huolehtii asetusdatan hallinnasta ja noodien käynnistämisestä sekä pysäyttämisestä, mutta varsinainen tieto sijaitsee datanoodeissa. Taulut on tarkoitettu säilytettäviksi levyn sijaan kokonaan keskusmuistissa. MySQL klusterin yhteydessä käytetään nimitystä *replika* jokaisesta saman tiedon kopiosta (normaalisti replikalla viitataan palvelimeen). Mikäli klusterin replikoiden määrä on asetettu yhteen, tieto löytyy ainoastaan yhdeltä noodilta. Noodi kuuluu aina johonkin noodiryhmään ja noodiryhmien kokonaismäärä on datanoodien määrä jaettuna klusteritasolla määritetyllä replikoiden määrällä. Esimerkiksi jos datanoodeja on 8 ja replikoiden määräksi on asetettu 2, on noodiryhmiä 4. Data on jaettu noodiryhmien välillä ja ryhmän sisällä tieto on jaettu partitioihin, millä tässä yhteydessä tarkoitetaan yleisesti osia noodiryhmän datasta. Datasta on noodiryhmässä *n* määrä kopioita, missä *n* on replikoiden määrä. Kuvassa 10 noodiryhmät 0 ja 1 sisältävät molemmat kaksi noodia ja tiedot on molemmissa ryhmissä jaettu kahteen partitioon. Ryhmässä 0 molemmat noodit sisältävät partitiot 0 ja 2; noodissa 1 on partition 0 ensisijainen kopio ja noodissa 2 partition 2 ensisijainen kopio. Noodiryhmässä 1 molemmat noodit sisältävät vastaavasti partitiot 1 ja 3. [MySQL Ref. Manual 2018]



Kuva 10. Tiedon jakautuminen noodeille MySQL NDB-klusterissa. [MySQL Ref. Manual 2018]

Tapahtumalokia ei kirjoiteta tapahtumaa vahvistettaessa levylle, mutta tieto vahvistetaan kaikkien tiedon säilövien datanoodien keskusmuistiin. Tietoa häviää vasta noodien kaatuessa yhtäaikaisesti. Kuormantasaajina toimivien SQL noodien ja datanoodien välinen kommunikointi käyttää *NDB-protokollaa*, joka määrittelee pyynnöt tiedonhakuun sekä tapahtumienhallintaan [MySQL NDB 2018].

MySQL NDB ei tarjoa kyselyjen sisäistä rinnakkaisuutta. Ngamsuriyaroj ja Pornpattana [2010] ovat esittäneet kyselyjen rinnakkaiseen ajoon vParNDB-väliohjelmistoa, joka kyselyjen reitittämisen lisäksi jakaa kyselyt osiin, lähettää ne oikeille noodeille ja yhdistää alikyselyiden tulokset. Testattaessa ohjelmistoa TPC-H-suorituskykytestillä usean noodin käyttö ja siten kyselyjen sisäisen rinnakkaisuuden lisääminen paransi merkittävästi joidenkin testin kyselyjen suoritusaikaa, tosin ei kaikkien.

Sekä Twitter että Facebook käyttävät MySQL:ää jakaen tiedon suurelle määrälle MySQL-palvelimia. Yhden käyttäjän tiedot löytyvät pääosin yhdeltä palvelimelta, jolloin käyttäjään kohdistuvat lukuoperaatiot ovat nopeita. Tiedon jakautuessa suurelle määrälle palvelimia menetetään kuitenkin osa relaatiotietokannan normaaleista operaatioista ja ACID-periaatteista [Harrison 2015].

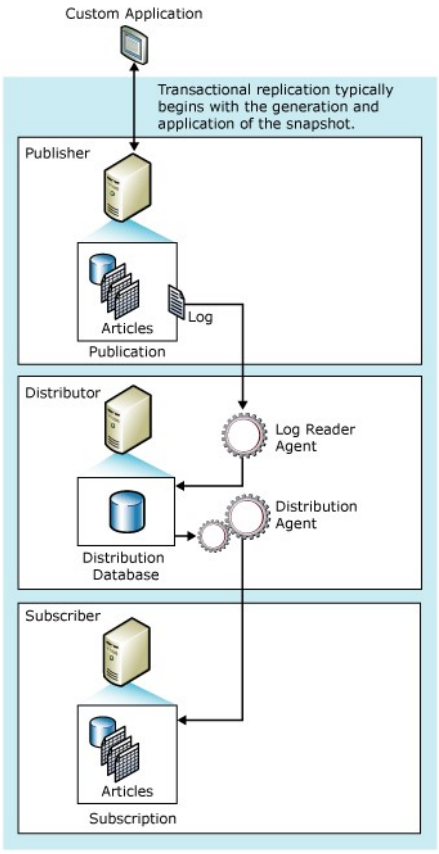
Facebookin käyttäjätieto on jaettu tuhansille palvelimille, joiden välille käyttäjätiedot on partitioitu. Jokaisella palvelimella voi olla useita partitioita ja lisäksi partitiot on replikoitu eri palvelimille ja palvelinsaleihin [Priymak 2013]. Facebookin ratkaisu on pitkälle räätälöity, ja osassa palvelimista käytetään InnoDB:n sijasta Facebookin kehittämää MyRocks tallennusmoottoria [Matsunobu 2016].

5.2.3. Microsoft SQL Server

Microsoft SQL Server on Microsoftin tarjoama relaatiotietokantajärjestelmä, jonka ensimmäinen versio on julkaistu vuonna 1989. Tuotteen versiot on numeroitu julkaisuvuosien mukaan ja viimeisimmät versiot ovat 2016 ja 2017. Järjestelmä tarjoaa täyden replikoinnin ja korkean saatavuuden SQL Server AlwaysOn Availability Groups -ratkaisun. Korkean saatavuuden toteuttamiseksi SQL Serverin pitää olla asennettuna Windows Clusteriin (käyttöjärjestelmätason hajautusratkaisu). Kyseinen ratkaisu tukee lukukyselyitä palvelevia replikoita [SQL Server Av. Groups 2016].

Tavalliseen replikointiin SQL Serverissä voidaan käyttää kolmea tekniikkaa: reaaliaikaista *tapahtumapohjaista replikointia* (transactional replication), asiakas- ja palvelinkoneiden väliseen käyttöön sopivaa *yhdentävää replikointia* (merge replication) ja *tilannevedosreplikointia* (snapshot replication), jota käytetään usein muita replikointiratkaisuja käyttöön otettaessa. Isäntänoodia kutsutaan *julkaisijaksi* (publisher) ja replikanoodia *tilaajaksi* (subscriber). Muutokset tiedoissa lähetetään tilaajalle samassa järjestyksessä kuin missä ne on tehty julkaisijalla. Muutoksia tilaajien datassa ei suositella, sillä mitään ei siirretä takaisin julkaisijalle.

Myös tässä replikoinnissa muutokset toimitetaan kopioimalla niitä tapahtumalokista: *lokinlukija-agentti* (Log Reader Agent) kopioi uudet tapahtumat *jakelijatietokantaan* (distribution database). Jakelija-agentti jakaa puolestaan välittää tapahtumat tilaajille. Kuvassa 11 on nähtävissä tapahtumapohjaisen replikoinnin kokonaisarkkitehtuuri: ohjelmisto lähettää julkaisijatietokantaan päivityskyselyn, joka valuu alas tilaajatietokantaan. [SQL Server Replication 2017]



Kuva 11. SQL Serverin tapahtumapohjainen replikointi. [SQL Server Replication 2017]

5.2.4. Muut järjestelmät

Galera Cluster on Codership Oy:n kehittämä klusteriratkaisu, joka käyttää noodeina MySQL- tai MariaDB-tietokantapalvelimia. Galera Clusterissa noodit toimivat MySQL:n normaalin InnoDB-moottorin kanssa. Järjestelmä mahdollistaa *automaattisen noodin lisäyksen* (node provisioning): jos noodi liitetään klusteriin, se voidaan automaattisesti saattaa synkroniseen tilaan muiden noodien kanssa. Galera käyttää *varmennepohjaista replikointia* (certification-based replication). Tämä tarkoittaa, että tapahtuma ajetaan normaalisti vahvistukseen asti. Tässä vaiheessa kaikista muutoksista koostetaan *kirjoitusjoukko* (write-set), joka lähetetään kaikille noodeille. Jokaisella noodilla tehdään kirjoitusjoukolle *varmennetesti*, jolla testataan voiko muutoksen kirjoittaa vai peruutetaanko tapahtuma. Galera Clusterissa varmennetesti toteutetaan tapahtumille annetuilla ID-numeroilla [Galera 2018].

Ratkaisu koostuu Galera replikaatiokirjastosta ja *kirjoitusjoukon replikoinnin* (WSREP - Write Set Replication) laajennoksesta tietokantaan. Kun järjestelmään liittyy

uusi tai aikaisemmin liittyneenä ollut noodi, voidaan olemassa olevan noodin tila replikoida liittyvään noodiin *tilannevedoksen siirrolla* (SST - state snapshot transfer) tai *kasautuvalla tilan siirrolla* (incremental state transfer). SST:tä käytettäessä olemassa olevasta noodista tehdään täysi kopio uutta noodia varten. Kasautuvaa tilan siirtoa voidaan käyttää, jos liittyvän noodin tilan ID (UUID) on sama kuin ryhmän ja kaikki puuttuvat kirjoitusjoukot (write sets) löytyvät luovuttavan noodin (donor) *kirjoitusjoukkovarastosta* (write-set cache). ID-tunnisteen lopussa olevaa sekvenssinumeroa m voidaan käyttää selvittämään, onko kirjoitusjoukkovaraston sisältö riittävä: mikäli m löytyy kirjoitusjoukkovarastosta, löytyvät myös kaikki tapahtumat joiden sekvenssin ID-tunniste on suurempi kuin m. Tällöin käytetään kasautuvan tilan siirtoa ja siirretään kaikki tapahtumat, joiden sekvenssinumero k on välillä $m \leq k \leq n$, missä n on klusterin noodien nykyisen tilan sekvenssinumero.

Esimerkki 3:

Liittyvän noodin tilan ID on 5a76ef62-30ec-11e1-0800-dba504cf2aab:197222

ID-numeron lopusta löytyy sekvenssinumero 197222

Klusterin noodien tila on 5a76ef62-30ec-11e1-0800-dba504cf2aab:201913

Luovuttava noodi voi tarkastaa löytyykö sen kirjoitusjoukkovarastosta sekvenssinumeroa 197222. Mikäli numero löytyy, käytetään kasautuvaa tilan siirtoa (vahvistetut tapahtumat 197222 – 201913). Mikäli numeroa ei löydy, käytetään tilannevedoksen siirtoa eli kaikki tiedot joudutaan replikoimaan. [Galera 2018]

Myös Continuentin Tungsten-järjestelmät tarjoavat Oraclen, MySQL ja MariaDB:n tietokantojen ympärille replikointia. Ratkaisu tukee keskitettyä ja hajautettua replikointia, sekä ryppäiden välistä replikointia. Replikanoodit toimivat kuumina valmiusnoodeina ja ne palvelevat lukukyselyjä. [Continuent 2018]

SAP SE on yksi suurimmista liikeprosessien hallintaan ohjelmistoja kehittävistä yrityksistä. SAP:n järjestelmien tietokantana toimiva MaxDB tarjoaa kuuman valmiuden replikoinnin, joka koostuu isäntä- ja valmiuspalvelimista. Valmiuspalvelimelle ei voi lähettää kyselyjä, vaan se toimii ainoastaan isäntäpalvelimen varapalvelimena [MaxDB 2018].

MariaDB MaxScale on kuormantasaajaratkaisu MariaDB-tietokannalle. Se käyttää reititysalgoritmeja, jotka määräävät tietokantanoodin, jolle kysely lähetetään. Luku- ja kirjoituskyselyt voidaan erotella joko palvelimella tai ne voidaan jättää asiakasohjelmiston eroteltavaksi. Tällöin asiakasohjelmiston pitää toimittaa tieto siitä kumman tyyppisestä kyselystä on kyse. [MaxScale 2018]

C-JDBC (myöhemmältä nimeltään Sequoia [C-JDBC 2008]) tarjoaa väliohjelmiston asiakasohjelmien ja tietokantapalvelimien väliin. Asiakasohjelmistoille näkyy rajapintana yksi virtuaalinen tietokanta. *Pyyntöjenkäsittelijä* (request manager) käsittelee kaikki yhdelle virtuaalitietokannalle tulevat kyselyt. Se koostuu rinnakkaisuudenhallinnan tehtäviä hoitavasta *vuorottelijasta* (scheduler), välimuisteista ja kuormantasaajasta. C-JDBC käyttää RAIDb tekniikkaa (Redundant Array of Inexpensive Databases) [Cecchet 2004]. Malkowski et al. [2009] ovat kokeellisesti testanneet C-JDBC:llä erilaisia pullonkaulatilanteita. Taton et al. [2006] ovat rakentaneet C-JDBC:n päälle järjestelmän, jossa esimerkiksi kaatuneiden tietokantanoodien korvaaminen uusilla tapahtuu automaattisesti. Ratkaisu koostuu *sensoreista* (sensors), jotka havaitsevat esimerkiksi tietokantanoodin kaatumisen. *Analyysi-* ja *päätöksentekokomponentit* vastaavat uudelleenkonfiguroinnin asetuksista, esimerkiksi kaatuneen noodin korvaamisen uudella. *Toimijakomponentit* (actuators) vastaavat uuden konfiguraation käytäntöönpanosta.

IBM DB2 on IBM:n tietokantajärjestelmä, jonka kehitys on aloitettu 1970-luvulla. Tuoteperheeseen kuuluu useita eri suuruisille yrityksille kohdennettuja julkaisuja. DB2 tukee taulun partitiointia useammalle noodille [Ahuja 2006]. Lisäksi tuetaan asynkronista lokipohjaista replikointia [Nikolopoulou 2004]. *Materialisoidun kyselyn taulut* (materialized query table) ovat tauluja, joiden sisältö muodostetaan näkymien tapaan tietokantaan tehtävän kyselyn tuloksista (niillä on kuitenkin näkymistä poikkeavia ominaisuuksia kuten mahdollisuus sallia taulun suora päivitys SQL-lauseilla) [Melnyk 2005]. Materialisoidun kyselyn tauluilla voidaan toteuttaa osittainen replikaatio luomalla niitä siten, että lähdedatana on toisella partitiolla sijaitsevan taulun sisältö [IBM DB2].

Oraclella ei ole varsinaista resursseja jakamatonta hajautettua ratkaisua. Oracle RAC:ssa (Oracle Real Application Clusters) tietokannan tallennustila on jaettu noodien kesken. Järjestelmän noodeja kutsutaan *instansseiksi*. Näillä on omat muistiresurssinsa, mutta ne jakavat välimuistinsa. *Interconnect* on erillinen lähiverkko instanssien välillä, jossa järjestelmän sisäinen liikenne tapahtuu. Jos instanssi tarvitsee toisen instanssin resursseja, se ottaa yhteyden *globaaliin välimuistipalveluun* (GCS - Global Cache Service). *Datalohko* (data block) on sivua vastaava käsite Oraclen tietokantajärjestelmässä. GCS seuraa datalohkoja, joille voidaan myös antaa lukituspyyntöjä. GCS lisäksi vastaanottaa instanssien resurssipyyntöjä. [Gupta 2015]

Instanssit voivat myös lähettää toisilleen pyyntöjä datalohkoista. Mikäli instanssi lähettää toiselle datalohkon ennen kuin sitä on kirjoitettu levylle, se jättää itselleen *lohkon vanhan version* (PI – past image). Näistä versioista voidaan tarvittaessa palauttaa datalohko häiriötilanteessa. Kun lohko kirjoitetaan levylle, voidaan kaikki siihen liittyvät PI:t poistaa. *Konsistenssi luku* (CR – consistent read) puolestaan tapahtuu, jos instanssi

haluaa lukea toisen instanssin varaamaa lohkoa. Varaava instanssi lähettää tällöin varsinaisen datalohkon sijasta CR-kuvan lohkosta. Tämä on lohkon versio ilman varaavan instanssin mahdollisesti siihen tekemiä muutoksia. [Gupta 2015]

Verrattuna resursseja jakamattomiin ratkaisuihin, Oraclen ratkaisua on arvosteltu sen kalleudesta ja riittämättömästä skaalautuvuudesta suurimpien internetsivustojen käyttöön. Koska Oraclen RAC pitää kiinni ACID-periaatteista, aiheuttaa instanssien välinen katkos kirjoitusristiriitojen välttämiseksi jommankumman instanssin alasajon. Useissa verkkosovelluksissa halutaan kuitenkin pitää yllä korkeaa saatavuutta tarvittaessa väljentämällä tietokannan eheyden vaatimuksia. [Harrison 2015]

6. Optimaalinen taulujen hajautus

Tässä luvussa esittelen tarkemmin tapoja, joilla tietokannan taulut voidaan partitioida niissä järjestelmissä, joissa hajautus tapahtuu taulujen tasolla. Partitionnin rakenne voidaan tehdä suunnittelijan toimesta manuaalisesti ennen tietokantajärjestelmän käyttöönottoa tai se voidaan laskea automaattisesti esimerkkikuorman perusteella.

Eri ratkaisujen tehokkuuden testaamiseen on usein käytetty TPC:n (Transaction Processing Performance Council) suorituskykytestejä. Testit *TPC-C* ja *TPC-E* on suunniteltu OLTP-järjestelmien suorituskyvyn mittaamiseen ja *TPC-H* puolestaan OLAP-järjestelmien testaamiseen [TPC 2019]. Lisäksi OLAP-järjestelmien mittaamiseen on jo vanhentunut testi *TPC-R* [TPC 2019]. Myös Yahoo! Cloud Serving Benchmark (*YCSB*) on usein käytetty suorituskykytesti. YCSB on kehitetty pilvipalveluina toimivien tietokantajärjestelmien testaukseen [Cooper et al. 2010].

Optimaalisen partitioinnin ja replikoinnin löytäminen on NP-täydellinen ongelma, ja sen ratkaisemiseksi on ehdotettu useita eri hakualgoritmeja. Tyypillinen automaattinen optimaalisen hajautuksen haku koostuu järjestelmän lopullisesta käyttötarkoituksesta riippumatta seuraavista askeleista (esimerkiksi [Pavlo et al. 2012] [Bellatreche and Benkrid 2009]):

1. Haetaan esimerkkityökuorman perusteella yksinkertainen hajautusehdotus tai useita ehdotuksia.
2. Käytetään hakualgoritmia, jolla haetaan hajautusmalleja muuntelemalla 1. kohdan hajautusehdotuksia tai yhdistelemällä niitä.
3. Verrataan kohdassa 2 löydettyjä uusia malleja suorituskyvyltään kohdan 1. malleihin ja mikäli löydetään parempia malleja, käytetään näitä jatkossa suorituskyvyn vertailukohtina.
4. Toistetaan kohtia 2 ja 3 kunnes haulle etukäteen määritetyt aikarajat täyttyvät.
5. Valitaan haun aikana löytynyt suorituskyvyltään paras ratkaisu hajautusmalliksi.

Yksittäisten taulujen replikointi soveltuu parhaiten pieniin ja lukuintensiivisiin tauluihin. Replikoiduissa tauluissa lisäys- ja päivitysoperaatiot ovat raskaampia, koska päivitykset pitää jakaa myös replikoihin. Lukukysely voi sen sijaan lukea taulun sisällön siltä replikalta, jolla muut kyselyn osat ajetaan. Tällöin sitä ei tarvitse lähettää alikyselynä toiselle noodille ja vältetään monimutkaisempi kyselynkäsittely.

Myös tallennettujen proseduurien toiminta vaikuttaa optimaaliseen hajautukseen. Tallennettujen proseduurien toiminta riippuu sekä niille annetuista parametreista että tietokantataulujen sisällöstä, eikä järjestelmä yleensä tiedä etukäteen proseduurin ajamien kyselyiden tarkkaa sisältöä tai käsiteltävien rivien partitioita.

6.1. Hajautus OLTP-järjestelmässä

Reaaliaikaisissa järjestelmissä korostuu tarve hajauttaa tietokanta niin, että minimoidaan hajautettujen kyselyiden määrä. Erilaiset tässä kappaleessa esitellyt ratkaisut on koottu taulukkoon 6. Näistä kehittynein ratkaisu on Horticulture, joka perustuu osittain vanhemman graafeihin perustuvan hajautusratkaisun Schismin käyttämiin tekniikoihin.

Taulukko 6. Hajautusratkaisuja OLTP-järjestelmään

Ratkaisu	Toimintatapa
Partition advisor	Partitiointisuunnitelman luonti virtuaalisten partitioiden ja kyselysuunnitelmien avulla
Schism	Partitiointi ja replikointi käyttögraafien avulla
Horticulture	Partitiointi ja replikointi käyttögraafien avulla. Partitiointimallin etsintä LNS-haulla.

Rao et al. [2002] ovat kehittäneet IBM:n DB2-järjestelmän päälle *partitioneuvojan* (partition advisor). Järjestelmässä käytetään laajennettua kyselyn optimoijaa optimaalisen partitioinnin laskentaan. Laajennettu optimoija sisältää mahdollisuuden partitiointiskeemojen suositteluun ja arviointiin. Syötteenä partitioneuvoja saa kyselyistä koostuvan työkuorman ja eri kyselyiden esiintymistiheyden työkuormassa. Optimoijan *suosittelutilaa* (RECOMMEND mode) käytettäessä kyselyt syötetään sille yksi kerrallaan, ja tämä palauttaa optimaalisen partitiointiskeeman käsiteltävälle kyselylle. Nämä kyselykohtaiset optimaaliset suunnitelmat kirjoitetaan `CANDIDATE_PARTITION` tauluun. *Luettelointialgoritmin* tehtävänä on huolehtia, että jokaiselle taululle on vain yksi partitiointikandidaatti `CANDIDATE_PARTITION` taulussa. Tämän jälkeen työkuormaa testataan taulujen partitioiden erilaisilla kombinaatioilla optimoijan *arviointitilassa* (EVALUATE mode). Tässä tilassa luodaan virtuaalisia partitioita, joiden perusteella kyselynkäsittelijä luo kyselysuunnitelman, mutta työkuorman kyselyjä ei kuitenkaan varsinaisesti ajeta. Lopulta raportoidaan arviointitilan perusteella valitut parhaat partitiointisuunnitelmat kullekin taululle.

Agrawal et al. [2004] ovat kehittäneet vastaavanlaisen automaattisen optimaalisen partitioinnin haun hajauttamattomalle järjestelmälle, jota on testattu Microsoft SQL Serverillä. Haun tuottama hajautusmalli yhdistää vertikaalista ja horisontaalista

partitiointia. Ratkaisussa käytetään syötteenä työkuormaa, jossa jokaiselle kuorman kyselylle on annettu painoarvo. Järjestelmä toimii seuraavasti:

1. Sarakkeet jaetaan *sarakeryhmiin* (column-groups), joista ainoastaan työnkuormaan merkittävästi vaikuttavat ryhmät pidetään mukana jatkokäsittelyssä.
2. Jokainen kysely syötetään kyselynoptimoijalle, joka antaa optimaalisen hajautusmallin kyselylle. Näistä muodostuu ryhmä *kandidaattikonfiguraatioita*.
3. Koska edellisessä vaiheessa saadut hajautusmallit on valittu vain yksittäisten kyselyiden perusteella, yhdistellään malleja ja lisätään näitä yhdistelmiä kandidaattikonfiguraatioiden ryhmään.
4. Etsitään kandidaattikonfiguraatioiden ryhmästä optimaalisin ratkaisu.

Schism on graafeihin perustuva partitiointi- ja replikointiratkaisu. Syötteenä se saa tietokannan, esimerkkitapahtumista koostuvan työkuorman ja halutun partitioiden lukumäärän. Näiden perusteella se muodostaa partitioinnin pohjaksi *graafiesityksen* (graph representation) tapahtumien ja rivien suhteista. Graafien partitiointiin Schism käyttää graafien käsittelyyn tarkoitettua MeTis-ohjelmistoa [Curino et al. 2010]. Schismin toiminnan vaiheet ovat seuraavat:

1. Graafiesitys
2. Graafipartitiointi
3. Selitys (tiivistys)
4. Lopullinen validointi

Graafiesityksessä tietokanta esitetään graafina, jossa taulujen yksittäiset rivit ovat solmuja. Ne rivit, joita käsitellään saman tapahtuman sisällä, yhdistetään särmillä toisiinsa. Särmän painoarvo kertoo yhdistävien tapahtumien määrän. Graafipartitioinnissa graafi jaetaan erillisiin partitioihin siten, että leikattujen särmien painoarvojen summa minimoidaan. Prosessia rajoitetaan kuitenkin niin, että datan määrä ja työkuorma partitioilla jakautuvat tasaisesti.

Esimerkki 4: Schismille annetaan syötteenä taulukossa 7 näkyvä `film`*-taulun sisältö sekä työkuorma, joka sisältää seuraavat yhden kyselyn sisältävät esimerkkitapahtumat.*
Kysely 1 palauttaa rivit 2, 4 ja 5:

```
SELECT * FROM film WHERE rating = 'G';
```

Kysely 2 palauttaa rivit 1 ja 2:

```
SELECT * FROM film WHERE film_id IN (1,2);
```

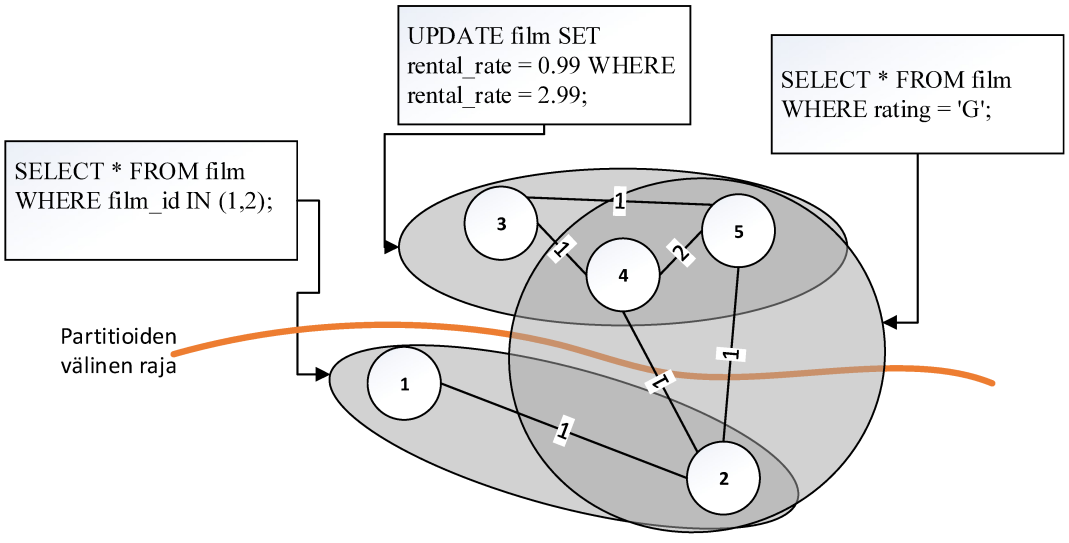
Kysely 3 päivittää rivit 3,4 ja 5:

```
UPDATE film SET rental_rate = 0.99
WHERE rental_rate = 2.99;
```

Taulukko 7. Taulun `film` sisältö esimerkissä 4, josta epäolennaiset sarakkeet on jätetty pois.

film_id	title	rating	rental_rate
1	ACADEMY DINOSAUR	PG	0.99
2	ACE GOLDFINGER	G	4.99
3	ADAPTATION HOLES	NC-17	2.99
4	AFFAIR PREJUDICE	G	2.99
5	AFRICAN EGG	G	2.99

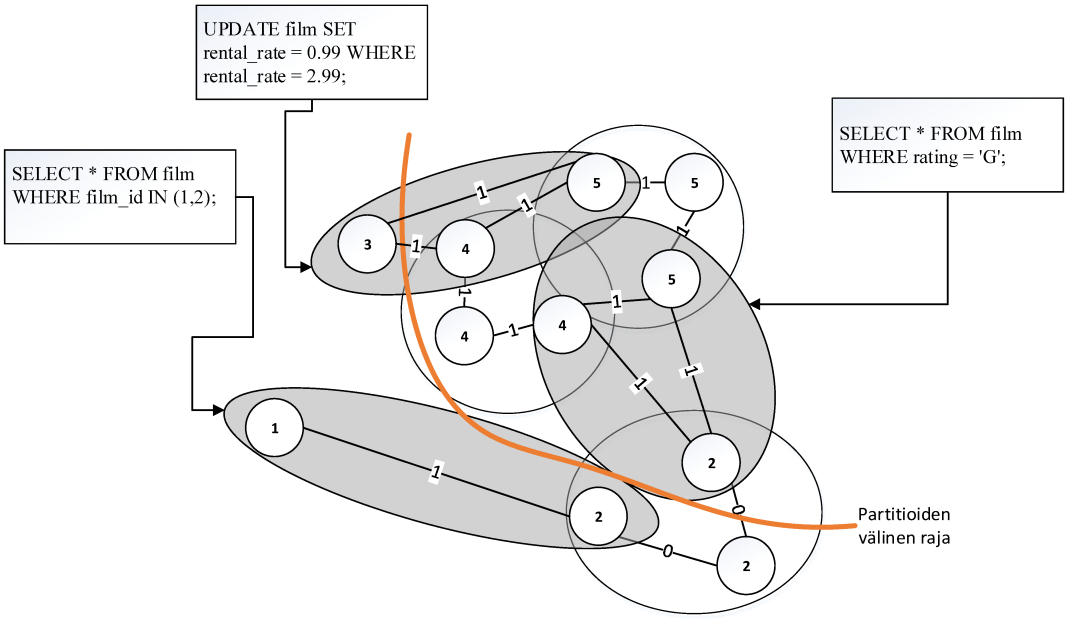
Esimerkkikyselyjen perusteella voidaan leikattujen särmien painoarvojen summa minimoida partitioimalla kuvan 12 osoittamalla tavalla partitioihin rivien jakautuessa seuraavasti: partitio A(1,2), partitio B(3,4,5).



Kuva 12. Schism ja graafipartitiointi esimerkin 4 kyselysyötteen perusteella.

Malliin voidaan lisätä myös replikointi, jolloin jokainen solmu muutetaan tähtikuvioksi, jossa on keskussolmun lisäksi *n* sakaraa (yksi sakara jokaista solmuun

kohdistuvaa tapahtumaa kohden). Tähtikuvion sisällä esiintyviin särmiin liittyvä painoarvo kertoo solmua päivittävien tapahtumien määrän. Jokainen päivitys replikoidulle riville aiheuttaa hajautetun kyselyn tietokantaan. Kuvassa 13 on esimerkin 4 graafimalliin lisätty replikointi ja yksi mahdollinen partitiointimalli, jossa rivi 2 on replikoitu ja partitiot ovat A(1,2,3) ja B(2,4,5). Tällöin esimerkkikyselyistä tarvitsee hajauttaa vain yksi, ja rivit saadaan jaettua tasaisesti partitioille.



Kuva 13. Schism sekä yhdistetty partitiointi ja replikointi esimerkin 4 kyselysyötteen perusteella.

Selitysvaiheessa pyritään löytämään suppea malli, joka onnistuu yksinkertaisilla säännöillä toisintamaan partitiointivaiheen valinnat rivien jaosta partitioihin. Tässä vaiheessa käytetään *päätöspuuta*, joka luokittelee annetun rivin kulkemalla puun juuresta *sisäsolmuja* pitkin *lehteen*. Sisäsolmut sisältävät testejä syötteenä annetun rivin arvoista, jonka mukaan rivi ohjataan eteenpäin puussa. Lehti määrittelee riville annettavan partition. Puun muodostuksen perusteena on kokoelma rivi-partitio -pareja. Jos tiedetään rivin sisältämät arvot, saadaan partitio selville kulkemalla puuta ja vertaamalla arvoa ehtoihin. Replikoiduille riveille annetaan erityinen *replikaatiotunniste*, joka edustaa niiden noodien joukkoa, joiden pitäisi sisältää kyseinen rivi.

Kun päätöspuun käyttö onnistuu, löydetään yksinkertaisilla säännöillä olennaisimmat osat aikaisemmin löydetystä partitioinnista. Esimerkiksi yllä olevien partitioiden säännöt voidaan yksinkertaistaen ilmaista seuraavasti:

Partitio A: `film_id` <= 2

Partitio B: `film_id` >= 3

Tällaisten yksinkertaisten sääntöjen tuottaminen ei ole kuitenkaan aina mahdollista ja kaikki säännöt eivät välttämättä ole hyödyllisiä. Sääntö on hyödyllinen vain, jos se

a) perustuu usein kyselyssä käytettyihin sarakkeisiin,
b) ei vähennä liikaa partitioinnin laatua luokittelemalla rivejä väärin,
c) ja toimii myös muille kuin syötteenä annetuille esimerkkikyselyille.

Schismin käyttämää graafiesitystä on myöhemmin arvosteltu tehottomaksi [Serafini et al. 2016].

Horticulture generoi esimerkkityökuorman perusteella partitiointi- ja replikointimalleja, jotka minimoivat jaettujen kyselyjen määrän tasapainottamalla kyselyjen yhteydenottoja noodeille. Horticulture ottaa huomioon myös kyselykuorman ajallisen vääristymän. Partitioinnin sijaan taulu replikoidaan tietokantapartitioiden välillä, jos siitä lähinnä luetaan tietoja ja muutokset ovat harvinaisia. Näiden keinojen lisäksi käytetään *toissijaisia indeksejä* partitioitujen taulujen niissä kentissä, joita ei olla käytetty partitioinnin perustana. Tällöin kyselyt, jotka käyttävät kyseessä olevia sarakkeita voidaan rajoittaa indeksin perusteella vain osalle noodeista. Schismin tavoin Horticulture käyttää käyttögraafeja. Partitioinnissa tuetaan arvoväli- ja hajautuspartitiointia. [Pavlo et al. 2012]

Parametrina Horticulture saa tietokannan rakenteen, tallennettujen proseduurien määrittelyt ja esimerkkityökuorman. Hajautusta etsitään *lokaalilla haulla*, joka ei pyri käymään läpi kaikki mahdollisia hajautusmalleja, vaan keskittyy ensimmäisenä muodostettavaa mallia jossain määrin muistuttaviin malleihin. Horticulture käyttää *LNS-hakua* (large neighbourhood search), joka suoritetaan seuraavissa vaiheissa:

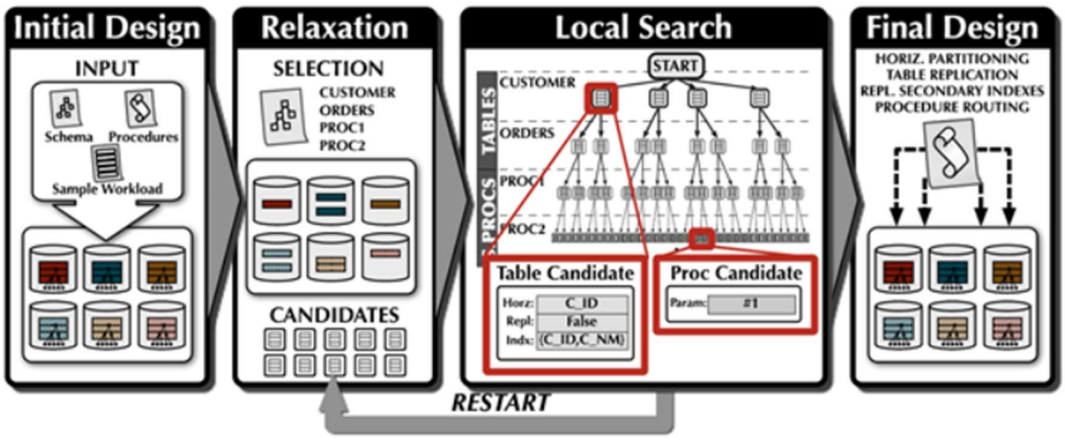
1. Esimerkkityökuorma analysoidaan hakuprosessin helpottamiseksi
2. Luodaan *ensimmäinen hajautusmalli* (initial design) D_{best} perustuen tietokannan käytetyimpiin sarakkeisiin
3. *Löyhennyksessä* (relaxation) luodaan vajaa malli D_{relax} käyttämällä osittain mallia D_{best}. Osittainen malli luodaan nollaamalla satunnaisten tietokantataulujen partitiointiasetukset mallissa D_{best}.
4. Tehdään lokaali haku käyttäen vajaata mallia aloituspisteenä. Mikäli löydetään malli, jolla on pienempi kokonaiskustannus kuin mallilla D_{best}, tehdään

löydetystä mallista uusi D_{best}. Haku päättyy, kun on käyty läpi tietty määrä malleja tai kun D_{relax}-mallin läheisyydessä ei ole enää malleja jäljellä.

5. Mikäli algoritmille annettu aikaraja täyttyy, pysäytetään haku ja palautetaan D_{best}. Muussa tapauksessa muodostetaan uusi D_{relax} tämän hetkisen D_{best}-mallin perusteella

Ensimmäinen hajautusmalli haetaan yksinkertaisilla säännöillä, jotta saadaan muodostettua alkupiste haulle ja karkea kustannusyläraja, jonka ylittävät mallit voidaan suoraan hylätä:

1. Jokaiselle taululle haetaan käytetyin sarake ja partitioidaan tämän mukaan.
2. Replikoidaan kaikki taulut, joihin tehdään vain lukukyselyitä, mikäli ei ylitetä noodin tilarajoituksia.
3. Haetaan toiseksi käytetyin sarake, jonka tietoja ei päivitetä usein ja tehdään tälle toissijainen indeksi, mikäli ei ylitetä tilarajoituksia.
4. Tallennetuille proseduureille haetaan *reititysparametri*: proseduurin se parametri, jonka arvo määrittelee niiden kyselyiden toimintaa, joissa ovat mahdollisimman usein mukana partitioinnissa käytetyt sarakkeet.



Kuva 14. Horticulturen toimintakaavio [Pavlo et al. 2012]

Haun jälkeen saadaan lopullinen malli, johon sisällytetään myös partitioinnin ja replikoinnin lisäksi myös toissijaiset indeksit ja tallennettujen proseduurien reitityspparametrit. Horticulturen toimintaprosessin kohdat näkyvät kuvassa 14: vasemmanpuoleisessa osiossa analysoidaan työkuorma ja muodostetaan ensimmäinen hajautusmalli. Seuraavassa osiossa eli löyhennyksessä luodaan vajaa malli ja kolmannessa osiossa suoritetaan tämän pohjalta lokaali haku. Neljännessä osiossa

muodostetaan lokaalin haun perusteella löytynyt paras malli, joka sisältää tiedot partitioitavista ja replikoitavista tauluista, luotavista toissijaisista indekseistä sekä *tallennettujen proseduurien reitityksistä* (procedure routing) reititysparametrien perusteella.

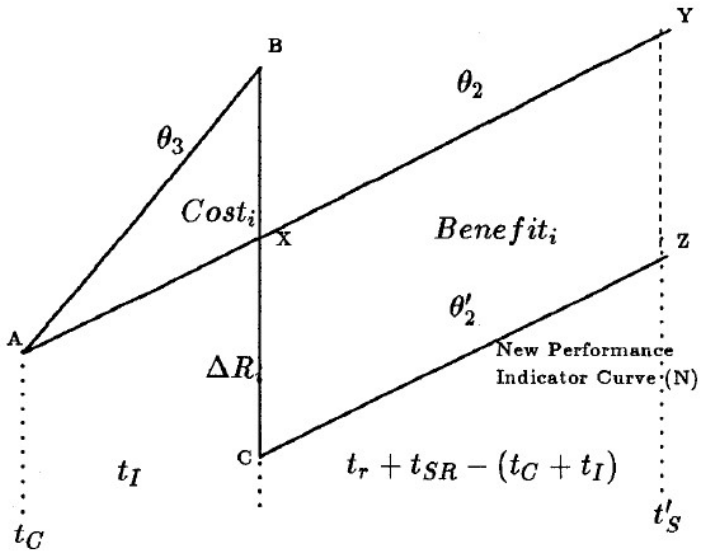
6.2. Uudelleenhajautus OLTP-järjestelmässä

Uudelleenhajautuksessa tietokannan tietoa siirretään noodien välillä. Tällä voidaan tasoittaa työkuormaa, mikäli partitioidun taulun useimmin käytetyt rivit ovat keskittyneet vain osalle noodeista, tai jos noodeissa muuten ilmenee vääristymää [Baru and Zilio 1993]. Uudelleenhajautusta tarvitaan myös, mikäli halutaan horisontaalisesti skaalata ylöspäin siten, että uusille noodeille jaetaan osa olemassa oleviin riveihin kohdistuvasta kuormasta [Taft et al. 2014]. Taulukkoon 8 on koottu tässä kappaleessa tarkasteltavia uudelleenhajautukseen osallistuvia komponentteja.

Taulukko 8. Uudelleenhajautusratkaisuja

Ratkaisu	Ominaisuudet
E-Store	Kokonaisratkaisu, jossa mukana muun muassa Squall
Squall	Rivien siirto osissa ilman käyttökatkoa
Clay	Siirrettävien rivien valinta kimpuissa lämpögraafin perusteella
Wildebeest	Rivien siirto osissa ilman käyttökatkoa

Baru ja Zilio [1993] ovat tutkineet käytössä olevan hajautetun tietokannan uudelleenorganisoinnin kustannuksia ja hyötyjä. Tuotantokäytössä olevan tietokannan suorituskyky huononee hiljalleen ajan kuluessa, kun erilaiset päivitys-, lisäys- ja poisto-operaatiot aiheuttavat vääristymää kannan tauluissa. Kuvan 15 suora AY kuvastaa tätä kasvavaa vasteaikaa, kun uudelleenhajautusta ei tehdä tarkastellulla aikavälillä. Mikäli datan hajautusta lähdetään optimoimaan ajan hetkellä t_C, aiheuttaa se hetkellisesti ylimääräistä kuormaa palvelimella (suora AB). Uudelleenhajautuksen loputtua vasteaika tippuu (piste C), jonka jälkeen se taas lähtee kasvamaan normaalien tietokantaoperaatioiden seurauksena (suora CZ). Kustannukset saadaan vertaamalla suoraa AB suoraan AX ja laskemalla kolmion ABX pinta-ala. Hyödyt taas saadaan laskemalla nelikulmion CXYZ pinta-ala.



Kuva 15. Datan uudelleenpartitioinnin kustannukset ja hyödyt. Vaaka-akselilla aika ja pystyakselilla keskimääräinen vasteaika [Baru and Zilio 1993]

H-Storen päälle rakennettu Squall toteuttaa rivien siirtämisen partitiolta toiselle ilman käyttökatkoa. Järjestelmän kuorman seuranta ja uuden partitiomallin laskenta hoidetaan toisaalla ja Squall aktivoituu vasta, kun se saa käskyn uudelleenpartitioinnin aloittamiseen. Uudelleenpartitiointi koostuu kolmesta vaiheesta:

Aloitusvaihe. Valitaan *johtajanoodi*, joka lähettää muille noodeille kutsun uudelleenkonfigurointiin. Jokainen noodi laskee uuden partitiointisuunnitelman pohjalta siltä lähtevät ja sille tulevat rivit.

Migraatiovaihe. Squall seuraa siirtyvien rivien sijaintia. Mikäli rivin sijainnista ei ole varmuutta, ohjataan kysely sille noodille, jolla rivin pitäisi olla uuden suunnitelman mukaan. Jos rivi ei ole siirtynyt, se vedetään tässä vaiheessa vanhalta noodilta uudelle. Lisäksi Squall siirtää rivejä järjestelmän normaalien tietokantaoperaatioiden taustalla uusille noodeille.

Lopetusvaihe. Jokainen noodi seuraa siirtyvien rivien tilaa. Kun poistuvat tai lisättävät rivit ovat kaikki siirtyneet se ilmoittaa johtajanoodille migraation päättymisestä sillä noodilla. Kun johtajanoodi on saanut ilmoituksen kaikilta noodeilta, se lähettää noodeille ilmoituksen uudelleenkonfigurointivaiheen päättymisestä.

[Elmore et al. 2015]

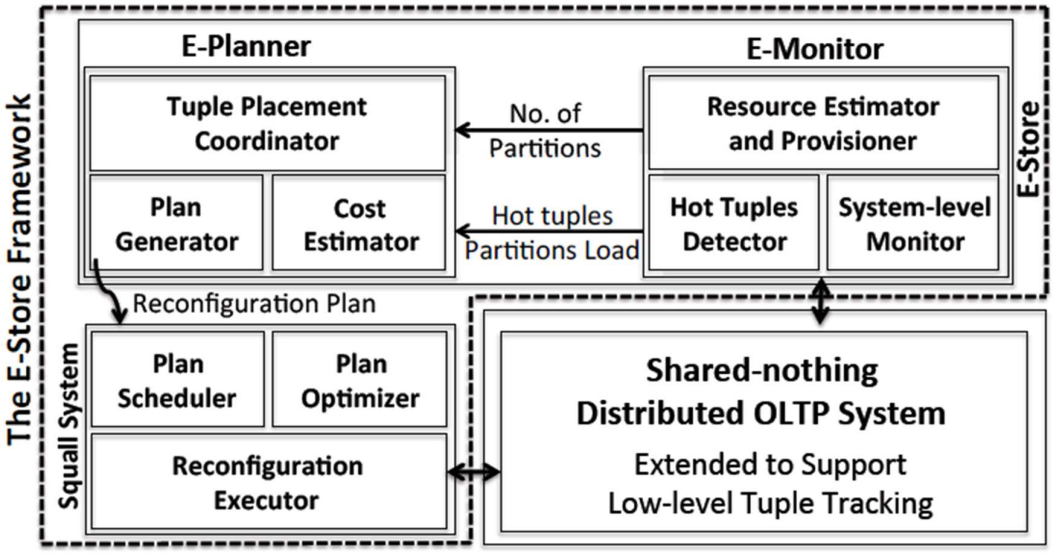
Samankaltaista prosessia on myös ehdotettu kokonaisten tietokantojen siirtoon pilviympäristössä toiselle instanssille, jolloin data siirtyy sivu kerrallaan ja kyselyt tietokantaan jaetaan hetken ajan kahden instanssin kesken [Elmore et al. 2011].

E-Store koostuu kolmesta alijärjestelmästä: E-Monitorista, E-Plannerista ja Squallista. E-Monitor tarkkailee hajautetun tietokannan työkuormaa ja vääristymää kuormassa. Mikäli vääristymä ylittää raja-arvot, käynnistetään rivien siirto vähemmän kuormitetulle partitiolle. E-Planner määrittää siirrettävät rivit sekä kohdepartition, ja Squall huolehtii rivien siirrosta samaan aikaan kuin järjestelmä vastaa normaalisti asiakasohjelmistoilta tuleviin tietokantakyselyihin. Myös järjestelmän automaattinen skaalaus ylöspäin tai alaspäin tätä ratkaisua käyttäen on mahdollista. E-Store toimii vain puumuotoiselle tietokantarakenteelle eli rakenne ei saa sisältää monen suhde moneen -viittauksia. E-Storen arkkitehtuuri on esitetty kuvassa 16.

E-Storessa käytetään kaksikerroksista partitiointia, jossa järjestelmän käytetyimmät, *kuumat rivit* (hot tuples) partitioidaan yksitellen. Vähemmän käytetyistä riveistä muodostettavat *kylmät lohkot* (cold chunks) voidaan partitioida esimerkiksi 100 000 rivin erissä. E-Monitor tunnistaa kuumat rivit ja toimittaa tiedon E-Plannerille, joka suunnittelee uuden partitiointiskeeman. Mikäli kokonaiskuorma ylittää kaikilla palvelimilla järjestelmään asetetun kuormituksen ylärajan, lisätään skeemaan lisää noodeja eli järjestelmää skaalataan ylöspäin.

Lähtötilanteessa E-Monitor seuraa prosessorin kuormitusta partitioilla. Mikäli kuormitus nousee ylärajan yli tai laskee alarajan alle, kytkeytyy tarkempi monitorointi päälle. Tarkemmassa monitoroinnissa seurataan yksittäisten rivien lisäystä, päivitystä ja lukua. Kuumista riveistä muodostetaan ensin lista. Koska yksittäisten rivien siirto on kevyt operaatio, uudelleenpartitioidaan nämä rivit ensin. Mikäli kuormassa on yhä vääristymää, uudelleenpartitioidaan kylmiä lohkoja työkuorman tasaamiseksi. [Taft et al. 2014]

Uudelleenprovisiointia varten on kehitetty kaksi versiota yleisestä *pakkauksen lajittelu -algoritmista* (bin packing algorithm), joissa pyritään lohkoja ja rivejä yhdistelemällä saavuttamaan keskiarvoa lähellä oleva kuorma noodilla samalla minimoiden siirrettävien rivien määrän. Koska useimpiin tilanteisiin nämä algoritmit ovat liian raskaita, on järjestelmään toteutettu myös kolme kevyempää algoritmia: *ahne algoritmi* (greedy), *laajennettu ahne algoritmi* (greedy extended) ja *ensimmäinen sopiva* -algoritmi (first fit). Ahne algoritmi siirtää käytetyimpiä rivejä ylikuormitetuilta palvelimilta pienimpien kuormien palvelimille. Laajennettu ahne algoritmi tekee saman, mutta siirtää myös käytetyimmät kylmät lohkot. Ensimmäinen sopiva -algoritmi hajauttaa uudelleen koko taulun jakamalla sen tasaisesti noodien kesken.



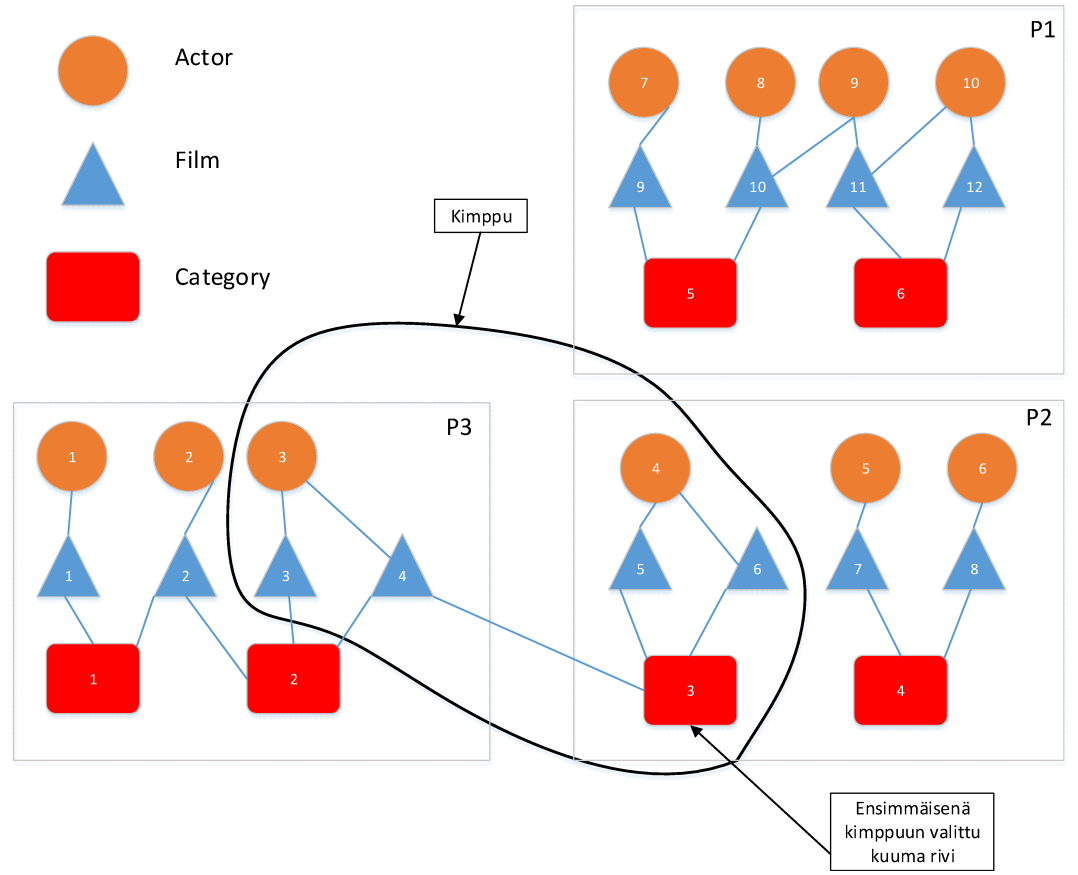
Kuva 16. E-Storen arkkitehtuuri, joka koostuu E-Plannerista, E-Monitorista ja Squallista. [Taft et al. 2014]

Clay mahdollistaa olemassa olevien rivien uudelleenjärjestämisen ja se toimii myös muille kuin puumuotoisille tietokantaskeemoille. Riveistä muodostetaan sellaisten rivien ryhmiä, joita käsitellään samoissa kyselyissä. Näitä ryhmiä kutsutaan *kimpuiksi* (clumps). Kimpun muodostaminen aloitetaan kuumasta rivistä, joka halutaan siirtää pois ylikuormittuneelta partitiolta. Tämän rivin lisäksi kimppuun kerätään muita rivejä, joita yleensä haetaan kuuman rivin yhteydessä. Tällä vältetään hajautettujen kyselyjen syntyminen kimppua siirrettäessä. Kimppu voi ulottua usealle partitiolle.

Kun ylikuormitusta havaitaan, Clay tarkkailee järjestelmän kuormaa muutaman sekunnin ajan. Tämän kuormituksen perusteella Clay muodostaa *lämpögraafin*. Graafiin solmuja ovat ne rivit, joita on käytetty havaintojakson aikana. Rivit yhdistyvät särmillä Schism-järjestelmän tapaan siten, että samassa kyselyssä käytetyt rivit yhdistetään. [Serafini et al. 2016]

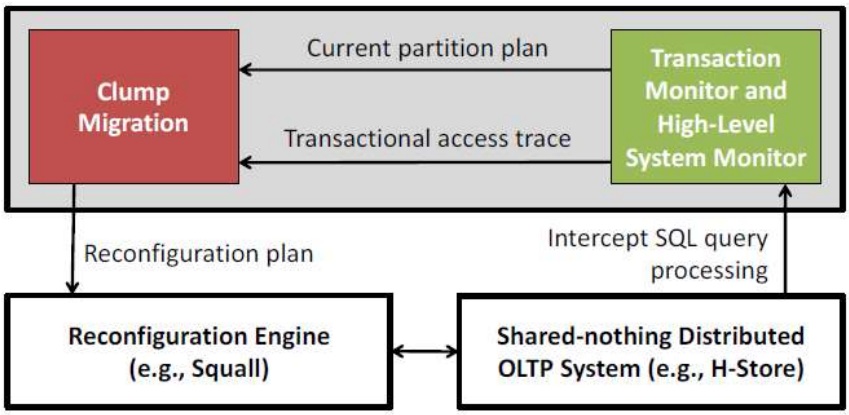
Esimerkki 5: taulujen `film`, `actor` *ja* `category` *sisältö on jakautunut partitioille P1, P2 ja P3 (kuva 17). Taulun* `film` *ja kahden muun taulun välinen yhteys on monen suhde moneen -tyyppinen. Kun partitio P2 ylikuormittuu, kuorman laatua tarkkaillaan muutaman sekunnin ajan. Tämän perusteella valitaan* `category`*-taulun rivi 3 ja sen ympärille kerätään samoissa kyselyissä esiintyviä rivejä kimppuun, joka ulottuu myös noodille P3. Lopuksi tämä kimppu siirretään partitiolle P1*

.



Kuva 17. Clay-järjestelmä ja kimpun määrittyminen

Clayn voi liittää resursseja jakamattoman hajautetun tietokantajärjestelmän yhteyteen. Kuvassa 18 on esitetty Clayn arkkitehtuuri: Squallia käytetään rivien siirron suunnitteluun, tosin tähän voitaisiin käyttää jotain toistakin ohjelmistoa: esimerkiksi aiemmin esitelty E-Planner toimii samanlaisena uudelleenhajautuksen suunnittelukomponenttina. Clay on suunniteltu toimimaan minkä tahansa OLTP-järjestelmän kanssa ja sitä on testattu H-Storen kanssa. [Serafini et al. 2016]



Kuva 18. Clayn toiminta (clump migration) suhteessa muihin järjestelmiin. [Serafini et al. 2016]

Dtxn:n osaksi on rakennettu uudelleenpartitioinnin mahdollisuus kuorman vääristyessä Wildebeest-komponentilla, joka mahdollistaa taulujen siirron, uudelleenpartitioinnin tai partitioiden yhdistämisen tuotantokäytössä olevassa OLTP-järjestelmässä. Siirron alkaessa luodaan kohdenoodille taulu ja tarvittavat indeksit. Tämän jälkeen Wildebeest odottaa avointen kyselyjen loppumista lähdenoodilla. Kun avoimet tapahtumat ovat loppuneet lähdenoodilla, lähetetään uudet kyselyt kohdenoodille. Koska kohteessa ei ole vielä rivejä, kyselyyn tarvittavat rivit siirretään kohdenoodille. Kaikkia tietoa päivittävät tai lisäävät kyselyt tapahtuvat kohdenoodilla ja taulun rivit siirtyvät hiljalleen kohdenoodille sen palvellessa sisään tulevia kyselyjä. Lisäksi noodilla ajetaan taustaprosessia, joka hakee säännöllisesti puuttuvia rivejä lähteestä. [Jones 2012]

6.3. Hajautus OLAP-järjestelmässä

OLAP-tietokantojen yhteydessä on usein tehokkaampaa hajauttaa siten, että voidaan käyttää hyväksi kyselyjen sisäistä rinnakkaisuutta kyselyjen välisen rinnakkaisuuden sijaan, koska tällöin saadaan vähennettyä pitkään ajavien kyselyjen suoritusaikaa.

Röhm et al. [2000] vertailivat TPC-R testin kanssa kahta vaihtoehtoista hajautusstrategiaa:

1. kaikkien taulujen replikointia
2. suurimman faktataulun partitiointia ja muiden taulujen replikointia kaikkien noodien kesken (*hybridimalli*).

Vaihtoehto 2 suoriutui paremmin sekä vasteaikojen että suoritustehon suhteen. Tämä vastaa OLTP-järjestelmien yhteydessä käytettyä strategiaa replikoida pienemmät taulut, joihin kohdistuu paljon lukukyselyitä: tietovarastossa faktataulu on yleensä kannan suurin taulu.

OLAP-kyselyjen tueksi voidaan käyttää *virtuaalisia partitioita*. Tällöin taulua ei varsinaisesti ositeta, mutta kyselyt jaetaan alikyselyiksi ja ohjataan partitiointiattribuutin mukaan tietyille noodeille ikään kuin partitiointi olisi tehty. Tätä tapaa partitioida virtuaalisesti on myös kutsuttu *yksinkertaiseksi virtuaaliseksi partitioinniksi* (SVP – single virtual partitioning) erotuksena muista virtuaalisen partitioinnin muodoista. Ositus kannattaa tehdä sitten, että alikyselyiden ajoaika on suunnilleen sama. Testattaessa on havaittu, että kysely on optimaalista jakaa kahdeksaan osaan, kun klusterissa on vähintään sen verran noodeja. [Akal et al. 2002]

Virtuaalisten partitioiden muodostus on ongelmallista, koska partitioiden arvovälejä on vaikea muodostaa siten, että partitioista muodostuisi yhtä suuria. Lisäksi edellä esiteltyä ratkaisua on myöhemmin arvosteltu siitä, ettei yksittäisillä noodeilla tueta hajautusominaisuuksia. Tällöin tietokantajärjestelmä noodilla saattaa käyttää arvovälin hakuun indeksoimatonta hakua ja lukea koko virtuaalisesti partitioidun taulun sisällön. Koska kyselyiden ajoon ei voida puuttua ajon alettua, ei kuorman tasaaminen myöskään onnistu kesken tapahtuman. [Lima et al. 2004]

Myös *hienojakoisessa virtuaalisessa partitioinnissa* kysely jaetaan alikyselyiksi virtuaalisten partitioiden mukaan, mutta partitioita voi olla useampi kuin yksi jokaista noodia kohden. Jakamalla kysely pienempiin alikyselyihin pyritään välttämään indeksittömät haut noodeilla. [Lima et al. 2004]

Cuzzocrea et al. [2013] ovat tutkineet OLAP-ratkaisun tilavaatimuksia neljällä eri partitiointimallilla:

1) ei partitiointia eikä replikointia,
2) koko tietokannan replikointi,
3) faktataulun partitiointi ja dimensiotaulujen replikointi,
4) tai valitun dimensiotaulun partitiointi ja faktataulun johdettu partitiointi tämän dimensiotaulun mukaan.

Samalla he ovat kehittäneet myös OLAP* ohjelmistokehyksen (OLAP* framework), joka toimii väliohjelmistona OLAP-asiakasohjelmiston ja OLAP-palvelimen välillä.

Partitioinnin pohjaksi on ehdotettu myös moniulotteista *m-Q-puu*-indeksiä, jonka avulla partitiointi tehdään useamman sarakkeen perusteella. Indeksi käsittelee *n*-sarakkeisen taulun rivejä pisteinä *n*-ulotteisessa kuutiossa, jossa yksi ulottuvuus on yhden sarakkeen arvoalue. Indeksointia varten kuutio jaetaan *alikuutioihin* (sub-cubes) ja

kuution jaottelua varten käytetään hyväksi *painopisteen* käsitettä: Jos kolmisarakkeisessa taulussa painopisteenä on rivi $X(a_1,b_2,c_3)$, voidaan kuutio jakaa kahteen osaan painopisteen sarakkeen A arvon toimiessa keskipisteenä:

osa 1: $a_n <= a_1$

osa 2: $a_n > a_1$.

Mikäli yhä jaetaan kuutio osiin muidenkin sarakkeiden arvojen mukaan, syntyy yhteensä 2^3 alikuutiota. Yleisesti kuutio voidaan jakaa 2^n osaan, missä *n* on sarakkeiden lukumäärä. Painopiste pitäisi valita siten, että rivit jakautuisivat mahdollisimman tasaisesti alikuutioihin. Jottei syntyisi suurta määrää alikuutioita vain muutaman rivin sisällöllä on m-Q-puussa erillinen *ylijäämäkuutio*, johon kaikkien vajaaksi jäävien alikuutioiden sisältö yhdistetään. [Polo et al. 1995]

Partitiointia ja replikointia suunniteltaessa voi myös olla hyödyllistä huomioida kyselyjen käsittelyn ongelmat. *F&A&R* (fragmentation, allocation, replication) on suunnittelumalli, jossa partitiointi, replikointi ja tiedon sijoittaminen tehdään yhtäaikaisesti. Jokainen vaihtoehtoisen partitiomallin edullisuus tiedon sijoittamiseen, replikointiin, kuormantasaukseen sekä kyselynkäsittelyyn testataan, ja pienimmän kustannuksen ratkaisu valitaan tietovaraston malliksi. [Benkrid et al. 2014]

Syötteenä F&A&R saa tiedot käsiteltävän tietovaraston relaatiomallista, yksittäisistä noodeista, OLAP-kyselyistä koostuvaan esimerkkityökuorman, kyselyjen frekvenssit kuormassa, sekä erilaisia raja-arvoja partitioinnille ja replikoinnille. Tämän lisäksi järjestelmälle annetaan tilastollista tietoa varaston tietomallista.

Partitiointimalli tuotetaan F&A&R:ssä käyttämällä *geneettistä algoritmia*, jossa käsitellään mahdollisia partitiointimalleja eli *partitiointimallikandidaatteja* esittäviä *kromosomeja*. Kromosomi voidaan esittää moniulotteisena taulukkona, joka koostuu taulukkoina esitetyistä partitiointisarakkeista. Yhdestä sarakkeesta muodostuvan taulukon solut ovat partitioihin kuuluvia arvoalueita.

Esimerkki 6: Jos partitiointimallikandidaatissa partitioidaan taulut `film` *sekä* `rental` *ja näistä edellinen partitioidaan* `film_id` *sarakkeen mukaan kahteen partitioon:*
P1: `film_id` *<= 500*
ja
P2: `film_id` *> 500,*
ovat partitioiden P1 ja P2 arvoalueet moniulotteisena taulukkona esitetyn kromosomin soluja.

Mallin alussa haetaan dimensiotaulukohtaisia partitiointisarake-kandidaatteja seuraavasti:

1. Erotellaan kaikki hakuehdot, joita käytetään esimerkkikuorman rivien valinnassa.
2. Yhdistetään dimensiotauluille ne ehdot, joissa kyseisen taulun tietoja käytetään.
3. Hylätään ne taulut, joiden tietoja ei ole yhdessäkään ehdossa.
4. Haetaan mahdolliset partitiointisarakkeet.
5. Poistetaan joukosta sarakkeet, joissa on korkea vääristymä.
6. Hajotetaan jokaisen partitiointiattribuutin arvojoukko aliarvojoukkoihin 1. kohdan hakupredikaattien perusteella.

Kun partitiointisarake-kandidaatit on haettu, geneettinen algoritmi luo sattumanvaraisen *populaation*, joka koostuu useista kromosomeista. Populaation laatua parannetaan *risteytys*- ja *mutaatio*-operaatioilla. Risteytyksessä eri kromosomit vaihtavat keskenään partitiointisarakkeille määriteltyjä arvoalueita. Mutaatiossa partitiointisarakkeiden arvoalueisiin tehdään etukäteen asetetulla todennäköisyydellä sattumanvaraisia muutoksia. Tiedon sijoittamisen tehdään kullekin kromosomille, minkä jälkeen kyselynkäsittelyn kustannukset lasketaan ja pienimmän kustannuksen kromosomi valitaan partitiointiskeemaksi. [Bellatreche and Benkrid 2009]

Tiedon sijoitukseen käytetään *sumean klusteroinnin menetelmää* (fuzzy clustering method), jossa klusterilla ei tarkoiteta tietokantaklusteria, vaan rypästä yhteen liittyviä fragmentteja. Sumeassa klusteroinnissa taulun partitiot jaetaan klustereihin, ja partitiot voivat kuulua useampaan klusteriin tietyllä *jäsenyysasteella* (membership degree). Jäsenyysaste on luku arvovälillä [0,1] ja se lasketaan *sumealla k-means klusterointialgoritmilla* (fuzzy k-means clustering algorithm), jonka esittely jätetään tämän tutkielman ulkopuolelle. Käsiteltävät partitiot jaetaan *M*:ään klusteriin, missä *M* on noodien määrä järjestelmässä. Fragmentteja (taulun partitioita) käsitellään jaottelun muodostamista varten seuraavasti:

1. Luodaan *fragmenttien käyttömatriisi*, jossa sarakkeina ovat fragmentit ja riveinä esimerkkikuorman kyselyt. Matriisin alkion arvo on 1, mikäli kysely käyttää vastaavan sarakkeen fragmenttia ja muuten 0.
2. Luodaan *fragmenttien jäsenyysmatriisi*, joka muodostuu fragmenteista sarakkeina ja klustereista riveinä. Alkion arvo on fragmentin jäsenyysaste vastaavassa klusterissa.
3. Fragmenttien klusterointi suoritetaan järjestelmällä kohdassa 2 fragmenttiin liittyvät klusterit laskevaan järjestykseen jäsenyysasteen mukaan ja lisäämällä

fragmentti *R*:ään ensimmäiseen klusteriin, missä *R* on *replikaatioaste* eli replikoiden määrä jokaiselle fragmentille järjestelmässä.

4. Muodostetaan *fragmenttien sijoitusmatriisi*, jossa riveinä ovat fragmentit ja sarakkeina noodit. Alkio saa arvon 1, jos fragmentti sijoitetaan kyseiselle noodille ja muuten arvon 0. Fragmenttien sijoittelu tapahtuu kohdan 3 klustereissa ja tapahtuu kiertovuorotteluperiaatteella.

[Bellatreche et al. 2010]

6.4. Uudelleenhajautus OLAP-järjestelmässä

Mukautuvassa virtuaalisessa partitioinnissa (AVP - adaptive virtual partitioning) virtuaalisten partitioiden kokoa muutetaan tarpeen mukaan. AVP pohjautuu edellä esiteltyyn hienojakoiseen virtuaaliseen partitiointiin. Se pyörii erikseen jokaisella noodilla eivätkä noodit kommunikoi toistensa kanssa. Järjestelmän käynnistyessä jokainen noodi saa itselleen kuuluvan virtuaalisen partition tiedon, ja ositus noodeille on tehty tässä vaiheessa kuten SVP:llä. Tämän jälkeen jokaisella noodilla tehdään seuraavat toimenpiteet:

1. Valitaan hyvin pieni arvoväli saadun virtuaalisen partition arvovälialueen alusta.
2. Ajetaan alikysely, joka käsittelee ainoastaan tätä arvoväliä.
3. Suurennetaan partitiokokoa ja ajetaan alikysely tälle arvovälille. Tätä toistetaan niin pitkään kuin ajoajan pidennys on suhteellisesti pienempi kuin partitiokoon suurennus.
4. Lopetetaan partitiokoon suurentaminen.
5. Tarkkaillaan kyselyjen ajoaikaa, jotta havaitaan suorituskyvyn huonontuminen.
6. Jos suorituskyky huonontuu pidemmäksi ajaksi, pienennetään partitiokokoa ja siirrytään kohtaan 2.

[Lima et al. 2004]

Furtado et al. [2005] yhdistävät edellisen luvun hybridimallin mukautuvaan virtuaaliseen partitiointiin: suurimmat taulut partitioidaan ja pienemmät taulut replikoidaan noodien välillä. Suurimpien taulujen partitiot lisäksi partitioidaan virtuaalisesti, jolloin noodin sisällä voidaan hyödyntää hienojakoisen virtuaalisen partitioinnin kyselyjen välistä rinnakkaisuutta. Tarvittaessa taulupartitiot voidaan vielä kahdentaa toisille noodeille. Tällöin ratkaisussa on mahdollista käyttää *dynaamista kuormantasausta* kyselyn ajoaikana, missä vapaa noodi voi auttaa kuormittunutta noodia joihinkin partitioihin liittyvien kyselyn osien käsittelyssä.

Lima et al. [2009] ovat kehittäneet *ketjutetun hajautuksen* (chained declustering), missä relaation ensisijaisen noodin vierekkäisille noodeille sijoitetaan

varmuuskopioreplikoita. Replikoiden määrän kasvaessa levytilan tarve kasvaa, mutta dynaaminen, AVP:hen perustuva kuormantasaus tehostuu. Virtuaalisten partitioiden käytön yhteydessä replikointi on suotavaa varsinkin, jos kyselyt jakautuvat hyvin epätasaisesti eri noodeille.

7. Optimaalinen kyselyjen käsittely

Optimaaliseen kyselyjen käsittelyyn liittyvät rinnakkaisessa tietokannassa kysymykset rinnakkaisuuden hallinnasta ja mahdollisesta lukkojen käsittelystä. Mikäli kaikkia kyselyn tarvitsemia rivejä ei löydy yhdeltä noodilta, pitää kysely hajauttaa ja rinnakkaisuuden hallinta monimutkaistuu.

Tiedon säilyvyydestä voidaan puolestaan levylle kirjoitettavan tapahtumalokin sijaan huolehtia pelkästään replikoiden avulla, jolloin kaatunut noodi palautuu ajantasaiseen tilaan hakemalla muilta replikoilta viimeisimpien päivitysten tiedot [Stonebraker et al. 2007].

Luvussa 5 esitelty Dtxn tarjoaa *spekulatiivisen rinnakkaisuuden käsittelyn* (speculative concurrency control), joka odottaessaan usealle partitiolle jaetun kyselyn vahvistusta tietyllä partitiolla ajaa kyseisellä noodilla muita kyselyitä. Kyselyitä ei voi lähettää noodin ulkopuolelle, koska jos odotettava kysely peruutetaan, pitää myös odotusaikana ajetut kyselyt peruuttaa. Mikäli odotettava kysely vahvistetaan, voidaan vahvistaa välittömästi myös kaikki odotusaikana ajetut kyselyt. Spekulatiivinen rinnakkaisuuden käsittely hyödyntää sitä, että tyypillisesti vain pieni osa tapahtumista peruutetaan. [Jones 2012]

Mikäli peruutettavia tapahtumia on työkuormassa paljon, ei spekulatiivinen rinnakkaisuuden käsittely toimi hyvin. Lukkojen käyttö toimii paremmin, jos järjestelmässä ajetaan paljon usean partition kyselyjä. [Jones et al. 2010]

Tapahtumien jakamisessa voidaan painottaa kuormantasauksen näkökulmaa, jolloin kyselyt jaetaan noodeille niiden senhetkisen työkuorman perusteella. Toisaalta tällöin useampia kyselyjä joudutaan hajauttamaan. Jos tapahtuman tarvitsevat rivit tiedetään etukäteen ja noodi valitaan siitä löytyvän datan perusteella, voidaan noudattaa seuraavia optimointeja [Pavlo et al. 2011]:

1. Jos kyselynkäsittelynoodi sisältää myös dataa, valitaan käsittelijäksi noodi, jossa on tapahtuman eniten käyttämä partitio. Mikäli koko tapahtuma saadaan ajettua tällä noodilla (*yhden noodin tapahtuma*), nopeutuu käsittelyaika huomattavasti.
2. Lukitaan vain ne partitiot, joita tapahtuma käsittelee. Mikäli tapahtuma tarvitsee tietoja usealta noodilta, lukitaan ne noodit kokonaan muilta tapahtumilta. Tätä varten tarvitaan tarkkaa ennustusta siihen, mitä noodeja tapahtuma käsittelee. Mikäli myöhemmin tapahtuma hakee tietoja lukitsemattomalta noodilta, joudutaan koko tapahtuma peruuttamaan ja käynnistämään uudelleen. Toisaalta jos lukitaan noodeja, joita tapahtuma ei tarvitse, tuhlataan resursseja pitämällä noodeja varattuina.

3. Peruutusloki otetaan pois päältä. Jos tiedon säilyvyydestä ei huolehdita tapahtumalokin avulla, vaan lokia tarvitaan ainoastaan peruuttamaan peruutettavan tapahtuman tekemät muutokset, voidaan lokin käytöstä luopua tapahtumalla, josta voidaan varmistua, ettei sitä peruuteta.
4. Spekulatiivinen rinnakkaisuuden käsittely otetaan käyttöön niillä partitioilla, joiden tietoa tapahtuma ei enää käsittele.

Täysin replikoidun järjestelmän ollessa kyseessä kyselyt voidaan jakaa kiertovuorottelu periaatteella. Yksinkertaisessa reititysstrategiassa kyselyt voidaan esimerkiksi pyrkiä lähettämään sille noodille, jossa on vähiten aktiivisia kyselyitä. Toisena mahdollisuutena on arvioida kyselystä aiheutuvat tiedonsiirto- sekä laskentakuormat ja tämän perusteella valita noodi, jolla on paras ennakoitu vasteaika [Carey et al. 1984].

Affiniteetti-pohjaisessa kyselyjen reitityksessä pyritään suorittamaan samaa dataa lukevat kyselyt samalla noodille. Tällä haetaan pienentynyttä tiedonsiirron kustannusta [Yu et al. 1986]. Ongelmana on kuitenkin se, että kyselyiden tiedonhakustrategian ollessa erilainen (haku ilman indeksiä ja haku indeksistä) ne eivät hyödytä toisiaan ja voivat jopa hidastaa kokonaishakuaikoja [Röhm et al. 2000]. Mikäli OLAP-tietokannassa kyselyiden joukossa on myös lyhyitä kyselyitä, voidaan niille antaa kyselyjonossa prioriteetti [Röhm et al. 2000].

Jhingran et al. [1993] ovat tutkineet kyselyjen optimointia rinnakkaisessa tietokannassa. Kyselyiden kustannus voidaan laskea määrittämällä *vasteajan peitto* (response time envelope). Tämä koostuu yksittäisten relaatioiden haun hinnasta ja ajoitettujen liitosoperaatioiden hinnasta. Liitosoperaatiot on ajoitettu käyttämällä *hienojakoista ajoitusta* (fine-grained scheduling), jolla pyritään siihen, että operaatiot eri noodeilla valmistuvat samaan aikaan. Tätä ajankohtaa kutsutaan *vesilinjaksi* (water line). Alikyselysuunnitelmia vertaillaan niiden vesilinjojen mukaan – korkeamman vesilinjan suunnitelma hylätään.

Kyselyiden rinnakkaisella ajolla saadaan parannettua kyselyn nopeutta varsinkin, jos kyselyssä ei käytetä indeksiä. Indeksin käyttö vähentää luettavien tietojen määrää, jolloin hajautetun kyselyn vaatimalla raskaammalla kyselynkäsittelyllä on suurempi vaikutus kokonaissuorituskykyyn [Rahm 1996]. Lisäksi suurimman relaation koko rajoittaa kyselyjen nopeutumista: mikäli tietokannassa ei ole yhtään suurta taulua, kyselyn hajottaminen eri partitioille ei kannata [Röhm et al. 2000].

Houdini on ohjelmistokehys, joka ennustaa tapahtumien käyttäytymistä *Markovin mallia* hyväksi käyttäen. Markovin malli on tilastollinen malli, joka tapahtuman viimeksi ajetun kyselyn perusteella mallintaa todennäköisyysjakauman tapahtuman mahdollisista

seuraavista toiminnoista. Ohjelmistokehys rakentaa tilastollisen mallin esimerkkityökuormasta. Mallin perusteella valitaan pohjanoodi ja arvioidaan tapahtuman aikana tarvittavat noodit. [Pavlo et al. 2011]

Sadat ja Lecca [2009] ovat tutkineet kyselyn hajauttamisen vaikutusta liitosoperaation suorituskykyyn. Mikäli kyseessä on yhtäläisyysliitos, voidaan liitettävät taulut jakaa käsiteltäviin partitioihin liitosehdon sarakkeiden arvojen mukaan. Jos kyseessä on muunlainen liitos, voidaan taulut yhä partitioida, mutta tällöin kaikkia taulujen partitioita pitää verrata keskenään.

OLAP-tietokannoissa tehtävien raskaampien kyselyiden optimoinnissa on kiinnitettävä huomiota myös kyselyn sisäiseen rinnakkaisuuteen. *QueCh* (Query processing with Chained declustering replication) on kolmivaiheinen prosessi, jossa ensimmäisessä vaiheessa kysely jaetaan faktataulun partitioiden mukaan eri noodeille. Toisessa vaiheessa kyselyt jaetaan vielä virtuaalisten partitioiden mukaan siten, että jokaisella virtuaalipartitioilla ajetaan yksi alikysely. Tämä mahdollistaa alikyselyiden siirron niille vapaille partitioille, joista löytyy taulun kaksinnus. Kolmannessa vaiheessa kuormitetuilta noodeilta siirretään tarvittaessa alikyselyitä vapaille replikanoodeille. [Lima et al. 2009]

8. Johtopäätökset

Tällä hetkellä hajautettujen tietokantojen tutkimus keskittyy erilaisten partitiointi- ja replikointimallien ja niiden suorituskyvyn tutkimiseen. Kaupalliset ratkaisut tukeutuvat toistaiseksi yksinkertaisiin hajautusmalleihin, joissa taulut voidaan hajauttaa esimerkiksi arvovälin mukaan. Kaupallisissa järjestelmissä ei myöskään automaattisesti tehdä työkuorman perusteella uudelleenpartitiointia. Yleisimmistä tietokannoista vain MySQL:n sekä IBM:n DB2:n klusterit tukevat sekä partitiointia että hajautettua replikointia. Keskitettyä replikointia tukevat kaikki tunnetuimmat tietokantajärjestelmät. Microsoft SQL Server tarjoaa vain täyden replikoinnin. Oraclella ei puolestaan ole tarjota varsinaista resursseja jakamatonta järjestelmää, vaan sen klusteri jakaa levyn.

Jotta hajautettua järjestelmää voidaan skaalata, pitää sen tarjota sekä osittaisen replikoinnin että partitioinnin mahdollisuus. Jotta ratkaisu sopisi tietovarastoksi, jossa ajetaan raskaampia OLAP-kyselyjä, sen pitäisi myös tukea kyselyjen sisäistä rinnakkaisuutta. MySQL NDB Clusterista ei tätä ominaisuutta löydy, joten se ei sovellu skaalautuvaksi OLAP-järjestelmäksi.

Suurimpien verkkosovellusten MySQL-ratkaisuissa tiedot on jaettu tuhansien palvelinten kesken. Tällöin yhtenäisyyden vaatimuksia joudutaan usein lieventämään. Yleinen versio MySQL:stä ei myöskään tue näin suuria hajautettuja tietokantoja, minkä vuoksi esimerkiksi Facebook onkin tehnyt oman versionsa MySQL:stä tämänkaltaista käyttöä varten.

Pienemmissä ratkaisuissa hajauttaminen saattaa johtaa järjestelmän monimutkaistumiseen ilman, että saavutetaan kaivattuja hyötyjä. Mikäli OLTP-järjestelmä hajautetaan siten, että seurauksena on suuri määrä hajautettuja kyselyjä, saattaa monimutkaistunut kyselyjen käsittely hidastaa koko järjestelmän käyttökelvottomaksi.

Tietokantaklusterin hajautukseen liittyvien erityiskysymysten, ja valmiiden ratkaisuiden vähyyden vuoksi tehokas hajauttamaton tietokanta on usein helpommin hallinnoitava ratkaisu tilanteissa, missä horisontaalinen skaalautuvuus ei ole välttämätöntä. Suurimmilla verkkopalveluiden tuottajilla kuten Facebookilla on riittävät resurssit tarvittavaan valmiiden ratkaisuiden räätälöintiin yrityksen tarpeita vastaaviksi.

Tutkimuskäyttöön tehdyt ohjelmistot mahdollistavat usein monimutkaisemmat hajautusarkkitehtuurit. H-Store-klusterin yhteyteen on kehitelty hienovaraisempia työkuormaan perustuvia partitiointimalleja, työkuorman tarkkailua ja automaattista uudelleenhajautusta, mikäli järjestelmä arvioi sen tarpeelliseksi. OLAP-järjestelmien hajautusta pyritään arvioimaan samankaltaisilla operaatioilla kuin OLTP-järjestelmissä painopisteen ollessa kuitenkin raskaampien kyselyiden optimoinnissa. Uudelleenhajautuksen tekniikasta saattaa olla hyötyä myös, vaikkei kokonaiskuorman

suuruus olisikaan muuttunut. Tällöin uudelleenhajautuksella vähennetään järjestelmään syntynyttä vääristymää.

Lähes optimaalista hajautusta laskevat järjestelmät käyttävät usein erilaisia hakualgoritmeja parhaimman hajautusmallin hakuun. Myös tämä poikkeaa nykyisistä tietokantaratkaisuista, joissa hajautuksen tehokkuutta ei pyritäkään mallintamaan, vaan suorituskyvyn kannalta parhaiden hajautusratkaisujen suunnitteleminen jätetään järjestelmän ylläpitäjän vastuulle.

Monien tutkimuskäyttöön kehitettyjen hajautettuun arkkitehtuuriin tähtäävien järjestelmien tai väliohjelmistojen kehitys on muutaman tutkimuksen jälkeen pysähtynyt. Tämä näkyy yleensä siten, ettei ohjelmistosta tarjota viimeisteltyä asennettavaa versiota, tai ettei sitä ole enää lainkaan julkisesti saatavilla. Jotta jokin tutkimuksissa käytetty ohjelmisto voisi toimia kaupallisessa käytössä esimerkiksi verkkosovellusten alustana, pitäisi sen kehittyä tuotteistetuksi ja käyttäjän kannalta viimeistellyksi ratkaisuiksi. Esimerkkinä tästä on VoltDB, joka perustuu tutkimuskäyttöön kehitetyn H-Storen arkkitehtuuriin.

Horisontaalisesti skaalautuvassa järjestelmässä on myös mahdollista monimutkaisemman hajautusarkkitehtuurin käytön sijaan yksinkertaisesti lisätä järjestelmään noodeja, jolloin saadaan enemmän resursseja käyttöön. Esimerkiksi Facebookin järjestelmissä tämä noodien hallinta on pitkälti pyritty automatisoimaan [Priymak 2013]. Pienemmissä kymmenien palvelimien klustereissa saattaisi tarkemmasta rivikohtaisesta työkuorman tarkkailusta ja tarvittaessa rivien uudelleenhajautuksesta olla enemmän hyötyä.

Hajautettu tietokantajärjestelmä on helpompi mahduttaa muistiin, koska tarvittava muistimäärä on pienempi ja skaalautumista voidaan hallita lisäämällä noodeja. Nopeat SSD-levyt voivat kuitenkin vähentää tarvetta mahduttaa tietokannan sisältö keskusmuistiin suorituskyvyn vuoksi.

Pilvipalveluihin yhdistettyinä hajautetusta tietokantajärjestelmästä on mahdollisuus rakentaa lähes vapaasti ylöspäin skaalautuvia ratkaisuja, jotka tällä hetkellä tarvitsevat kuitenkin hienosäätöä ja ohjelmiston räätälöintiä kokoluokan kasvaessa ulos tavanomaisista ratkaisuista. Nykyään pilviympäristöt myös usein mahdollistavat helpon vertikaalisen skaalautuvuuden, koska pilveen asennetulle palvelimelle voidaan yleensä antaa lisäresursseja nopealla vasteella.

Järjestelmän skaalautuessa horisontaalisesti ylöspäin ja noodien määrä kasvaessa kasvaa OLTP-järjestelmässä todennäköisyys siihen, että ajetaan usealta noodilta tietoja tarvitsevia hajautettuja kyselyitä. Monissa järjestelmissä onkin tutkittu hajautettujen kyselyiden aiheuttaman haitan minimoimista. Siitä huolimatta ne ovat yhä ACID-säännöt täyttävän hajautetun tietokannan ongelmallisin osa-alue. Tietovarastojen yhteydessä

puolestaan painottuu tarve kyselyn sisäiselle rinnakkaisuudelle, ja monessa ehdotetussa ratkaisussa hajautetaankin kyselyt virtuaalisilla partitioilla, jolloin noodi palvelee kyselyjä vain sille määriteltyyn taulun osaan, vaikka se sisältäisikin koko taulun sisällön.

NoSQL-järjestelmiin verrattuna horisontaalisesti skaalautuvat hajautetut kaupalliset relaatiotietokantaratkaisut eivät toistaiseksi ole laajassa käytössä. Niistä myös puuttuu tutkimusjärjestelmissä esiteltyjä ominaisuuksia, jotka mahdollistaisivat esimerkiksi automaattisen skaalauksen.

Viiteluettelo

[Agrawal et al. 2004] Sanjay Agrawal, Vivek Narasayya, Beverly Yang, Integrating vertical and horizontal partitioning into automated physical database design, *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data,* 2004, 359-370

[Ahuja 2006] Rav Ahuja, Introducing DB2 9, part 2: table partitioning in DB2 9, https://developer.ibm.com/tutorials/dm-0605ahuja2/, 2006. Viitattu: 14.12.2018.

[Akal et al. 2002] Fuat Akal, Klemens Böhm, Hans-Jörg Schek, OLAP query evaluation in a database cluster: a performance study on intra-query parallelism, *Advances in Databases and Information Systems. ADBIS 2002. LNCS 2435,* 2002, 218-231

[Alsberg and Day 1976] Peter A. Alsberg, John D. Day, A principle for resilient sharing of distributed resources, *Proceedings of the 2nd international conference on Software engineering,* 1976, 562-570

[Aslett 2011a] Matt Aslett, How will the database incumbents respond to NoSQL and NewSQL?, The 451 Group, April 2011

[Aslett 2011b] Matthew Aslett, What we talk about when we talk about NewSQL, https://blogs.the451group.com/information_management/2011/04/06/what-we-talk-about-when-we-talk-about-newsql/, 2011. Viitattu: 14.12.2018.

[Baru and Zilio 1993] Chaitanya Baru, Daniel C. Zilio, Data reorganization in parallel database systems, *Proceedings of the 1993 IEEE Workshop on Advances in Parallel and Distributed Systems,* 1993, 102 - 107

[Bellatreche and Benkrid 2009] Ladjel Bellatreche, Soumia Benkrid, A joint design approach of partitioning and allocation in parallel data warehouses, *International Conference on Data Warehousing and Knowledge Discovery, LNCS 5691,* 2009, 99-110

[Bellatreche et al. 2010] Ladjel Bellatreche, Alfredo Cuzzocrea, Soumia Benkrid, Query optimization over parallel relational data warehouses in distributed environments by simultaneous fragmentation and allocation, *Algorithms and Architectures for Parallel Processing. ICA3PP 2010. LNCS 6081,* 2010, 124-135

[Benkrid et al. 2014] Soumia Benkrid, Ladjel Bellatreche, Alfredo Cuzzocrea, A global paradigm for designing parallel relational data warehouses in distributed environments, *Transactions on Large-Scale Data- and Knowledge-Centered Systems XV. LNCS 8920,* 2014, 64-101

[Booth 1972] Grayce M. Booth, The use of distributed data bases in information networks, *Proceedings of the First International Conf on Computer Communication: Impacts and Implications,* 1972, 371-376

[Carey et al. 1984] Michael J. Carey, Miron Livny, Hongjun Lu, *Computer Sciences Technical Report #556,* Computer Sciences Department, University of Wisconsin, 1984

[Cattell 2011] Rick Cattell, Scalable SQL and NoSQL data stores, *ACM SIGMOD Record,* 39(4), December 2010, 12-27

[Cecchet 2004] Emmanuel Cecchet, C-JDBC: a middleware framework for database clustering, *IEEE Data Eng. Bull.* 27, 2004, 19-26

[C-JDBC 2008] ObjectWeb Consortium, C-JDBC: clustered JDBC, http://c-jdbc.ow2.org/, 2008. Viitattu: 14.12.2018.

[Codd 1970] Edgar Codd, A relational model of data for large shared data banks, *Communications of the ACM*, 13(6), June 1970, 377-387

[Continuent 2018] Continuent Ltd, Products and solutions, https://www.continuent.com/solutions/, 2018. Viitattu: 14.12.2018.

[Cooper et al. 2010] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, Russell Sears, Benchmarking cloud serving systems with YCSB, *Proceedings of the 1st ACM symposium on Cloud computing,* 2010, 143-154

[Copeland et al. 1988] George P. Copeland, William Alexander, Ellen E. Boughter, Tom W. Keller, Data placement in Bubba, *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data,* 1988, 99–108

[Curino et al. 2010] Carlo Curino, Evan Jones, Yang Zhang, Sam Madden, Schism: a workload driven approach to database replication and partitioning, *Proceedings of the VLDB Endowment*, 3(1-2), 2010, 48-57

[Cuzzocrea et al. 2013] Alfredo Cuzzocrea, Rim Moussa, Guandong Xu, OLAP*: effectively and efficiently supporting parallel OLAP over big data, *Model and Data Engineering. MEDI 2013. LNCS 8216,* 38-49

[Deppe and Fry 1976] Mark E. Deppe, James P. Fry, Distributed data bases : a summary of research, *Computer Networks* 1(2), 1976, 130-138

[Elmasri and Navathe 2011] Ramez Elmasri, Shamkant B. Navathe, *Fundamentals of Database Systems*, 6. painos, Addison-Wesley Publishing Company, USA 2010

[Elmore et al. 2011] Aaron J. Elmore, Sudipto Das, Divyakant Agrawal, Amr El Abbadi, Zephyr: live migration in shared nothing databases for elastic cloud platforms, *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data,* 2011, 301-312

[Elmore et al. 2015] Aaron J. Elmore, Vaibhav Arora, Rebecca Taft, Andrew Pavlo, Divyakant Agrawal, Amr El Abbadi, Squall: fine-grained live reconfiguration for partitioned main memory databases, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data,* 2015, 299-313

[Furtado et al. 2005] Camille Furtado, Alexandre A. B. Lima, Esther Pacitti, Patrick Valduriez, Marta Mattoso, Physical and virtual partitioning in OLAP database clusters, *17th International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD,* 2005, 143-150

[Galera 2018] Codership Ltd, Galera Cluster Documentation, http://galeracluster.com/documentation-webpages, 2018. Viitattu: 14.12.2018.

[Gray 1981] Jim Gray, The transaction concept: virtues and limitations, *Proceedings of the Seventh International Conference on Very Large Data Bases,* 1981, 144 - 154

[Gray et al. 1996] Jim Gray, Pat Helland, Patrick O'Neil, Dennis Shasha, The dangers of replication and a solution, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, 25(2),* 1996, 173-182

[Gupta 2015] Rohit Gupta, Oracle RAC Cache Fusion, http://www.dba-oracle.com/t_gupta_oracle_rac_cache_fusion.htm, 2015. Viitattu: 14.12.2018.

[Harrison 2015] Guy Harrison, *Next Generation Databases*, Springer Science+Business Media New York, 2015

[H-Store 2018] Brown University, Carnegie Mellon University, Massachusetts Institute of Technology, Yale University, H-Store: Next Generation OLTP Database Research, http://hstore.cs.brown.edu/, 2018. Viitattu: 14.12.2018.

[IBM DB2 2018] IBM Corporation, IBM Db2 Version 11.1 Knowledge Center, https://www.ibm.com/support/knowledgecenter/en/SSEPGG_11.1.0/com.ibm.db2.luw.welcome.doc/doc/welcome.html, 2018. Viitattu: 14.12.2018.

[IBM InfoSphere 2017] IBM Corporation, IBM InfoSphere Data Replication V11.4 documentation, https://www.ibm.com/support/knowledgecenter/en/SSTRGZ_11.4.0/com.ibm.idr.frontend.doc/pv_welcome.html, 2017. Viitattu: 16.1.2019.

[Jhingran et al. 1993] Anant Jhingran, Sriram Padmanabhan, Ambuj Shatdal, Join query optimization in parallel database systems, *Proceedings of the 1993 IEEE Workshop on Advances in Parallel and Distributed Systems,* 1993, 114 - 119

[Jones 2012] Evan P. C. Jones, *Fault-Tolerant Distributed Transactions for Partitioned OLTP Databases*, Massachusetts Institute of Technology, February 2012

[Jones et al. 2010] Evan P. C. Jones, Daniel J. Abadi, Samuel Madden, Low overhead concurrency control for partitioned main memory databases, *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data,* 2010, 603-614

[Kaitsa et al. 2007] Maria Kaitsa, Ilias Stavrakas, Theophanis Kontogiannis, Ilias Daradimos, Marios Panaousis and Dimos Triantis, Load balancing incoming IP

requests across a farm of clustered MySQL servers, *EUROCON 2007 The International Conference on "Computer as a Tool" Warsaw,* 2007, 546-550

[Kallman et al. 2008] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, Daniel J. Abadi, H-store: a high-performance, distributed main memory transaction processing system, *Proceedings of the VLDB Endowment,* 1(2), 2008, 1496-1499

[Kapila 2015] Amit Kapila, Well-known Databases Use Different Approaches for MVCC, https://www.enterprisedb.com/node/3412, 2015. Viitattu: 14.12.2018.

[Kemme and Alonso 2000] Bettina Kemme, Gustavo Alonso, Don't be lazy, be consistent - Postgres-R, a new way to implement database replication, *Proceedings of the 26th VLDB conference,* 2000, 134-143

[Kimball and Ross 2013] Ralph Kimball, Margy Ross, *The Data Warehouse Toolkit – The Definitive Guide to Dimensional Modeling 3rd ed.*, John Wiley & Sons, Inc. Indianapolis IN, USA, 2013

[Lake and Crowther 2013] Peter Lake, Paul Crowther, *Concise Guide to Databases*, Springer-Verlag London, 2013

[Lima et al. 2004] Alexandre A. B. Lima, Marta Mattoso, Patrick Valduriez, Adaptive virtual partitioning for OLAP query processing in a database cluster, *Anais/Proceedings XIX Simpósio Brasileiro de Bancos de Dados,* 2004, 92-105

[Lima et al. 2009] Alexandre A. B. Lima, Camille Furtado, Patrick Valduriez, Marta Mattoso, Parallel OLAP query processing in database clusters with data replication, *Distributed and Parallel Databases,* 25(1–2), April 2009, 97–123

[LSST 2018] LSST Corporation, LSST – About – Data Management, https://www.lsst.org/about/dm, 2018. Viitattu: 14.12.2018.

[Malkowski et al. 2009] Simon Malkowski, Markus Hedwigt, Deepal Jayasinghe, Junhee Park, Yasuhiko Kanemasal, Calton Pu, A new perspective on experimental analysis of N-tier systems: evaluating database scalability, multi-bottlenecks, and economical operation, *5th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom),* 2009

[Matsunobu 2016] Yoshinori Matsunobu, MyRocks: A space- and write-optimized MySQL database, https://code.fb.com/core-data/myrocks-a-space-and-write-optimized-mysql-database/, 2016. Viitattu: 14.12.2018.

[MaxDB 2018] SAP SE, What Is an SAP MaxDB Database? http://maxdb.sap.com/doc/7_7/6d/a02e727c6f4c3aa3a0c6468b9750a6/content.htm, 2018. Viitattu: 14.12.2018.

[MaxScale 2018] MariaDB Foundation, MaxScale Configuration & Usage Scenarios, https://mariadb.com/kb/en/mariadb-enterprise/mariadb-maxscale/maxscale-configuration-usage-scenarios/, 2018. Viitattu: 14.12.2018.

[Melnyk 2005] Roman Melnyk, An introduction to materialized query tables, https://www.ibm.com/developerworks/data/library/techarticle/dm-0509melnyk/index.html, 2005. Viitattu: 16.1.2019.

[MySQL NDB 2018] Oracle Corporation, NDB Cluster Internals - 3.1 NDB Protocol Overview https://dev.mysql.com/doc/ndb-internals/en/ndb-internals-ndb-protocol-overview.html, 2018. Viitattu: 14.12.2018.

[MySQL Ref. Manual 2018] Oracle Corporation, MySQL 5.7 Reference Manual, https://dev.mysql.com/doc/refman/5.7/en/, 2018. Viitattu: 14.12.2018.

[Nazaruk and Rauchman 2013] Alex Nazaruk, Michael Rauchman, Big Data in Capital Markets, SIGMOD 2013 esityskalvot, https://pdfs.semanticscholar.org/8a7a/28930d59931a0c8375aec77fcf242fe2995e.pdf, 2013. Viitattu: 14.12.2018.

[Ngamsuriyaroj and Pornpattana 2010] Sudsanguan Ngamsuriyaroj, Rangsan Pornpattana, Performance evaluation of TPC-H queries on MySQL cluster, *IEEE 24th International Conference on Advanced Information Networking and Applications Workshops (WAINA),* 2010, 1035-1040

[Nikolopoulou 2004] Mika Nikolopoulou, Replication setup for DB2 Universal Database, https://www.ibm.com/developerworks/data/library/techarticle/dm-0405nikolopoulou/index.html, 2004. Viitattu: 14.12.2018.

[Paes et al. 2008] Melissa Paes, Alexandre A.B. Lima, Patrick Valduriez, Marta Mattoso, High-performance query processing of a real-world OLAP database with ParGRES, *High Performance Computing for Computational Science - VECPAR 2008. LNCS 5336,* 188-200

[Page et al. 1985] Thomas W. Page, Jr., Matthew J. Weinstein, and Gerald J. Popek, Genesis: a distributed database operating system, *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data,* 1985, 374-387

[Pan and Zymbler 2013] Constantin S. Pan, Mikhail L. Zymbler, Taming elephants, or how to embed parallelism into PostgreSQL, *DEXA 2013, Part I, LNCS 8055,* 2013, 153–164

[Pavlo et al. 2011] Andrew Pavlo, Evan P.C. Jones, Stanley Zdonik, On predictive modeling for optimizing transaction execution in parallel OLTP systems, *Proceedings of the VLDB Endowment*, 5(2), 2011, 85-96

[Pavlo et al. 2012] Andrew Pavlo, Carlo Curino, Stan Zdonik, Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems, *Proceedings of the*

2012 ACM SIGMOD International Conference on Management of Data, 2012, 61-72

[Pavlo and Aslett 2016] Andrew Pavlo, Matthew Aslett, What's really new with NewSQL?, *SIGMOD Record,* 45(2), June 2016, 45-55

[pgpool-II 2018] Pgpool Wiki, http://www.pgpool.net/mediawiki/index.php/Main_Page, 2018. Viitattu: 14.12.2018.

[Polo et al. 1995] Antonio Polo Márquez, Manuel Barrena García, Juan Hernández Núñez, Juan Miguel Martínez, Pedro de Miguel, Manuel M. Nieto Rodríguez, Multi-dimensional partitioning for massively parallel database machines, *Proceedings Euromicro Workshop on Parallel and Distributed Processing,* January 1995, 25-27

[PostgreSQL 2018] The PostgreSQL Global Development Group, PostgreSQL 10 Documentation, https://www.postgresql.org/docs/10/static/, 2018. Viitattu: 14.12.2018.

[PostgreSQL Wiki 2018] PostgreSQL Wiki - Replication, Clustering, and Connection Pooling, https://wiki.postgresql.org/wiki/Replication,_Clustering,_and_Connection_Pooling, 2018. Viitattu: 14.12.2018.

[Postgres-XL 2018] The PostgreSQL Global Development Group, Postgres-XL Documentation, https://www.postgres-xl.org/documentation/, 2018. Viitattu: 14.12.2018.

[PowerDB 2018] ETH Zürich The Database Research Group, PowerDB - ACE: Autonomic Cluster Environment, http://www.dbs.ethz.ch/research/cont_23.html, 2018 (arkistoitu sivu). Viitattu: 14.12.2018.

[Priymak 2013] Shlomo Priymak, Under the hood: MySQL Pool Scanner (MPS), https://www.facebook.com/notes/facebook-engineering/under-the-hood-mysql-pool-scanner-mps/10151750529723920/, 2013. Viitattu: 14.12.2018.

[Rahm 1996] Erhard Rahm, Dynamic load balancing in parallel database systems, *Euro-Par'96 Parallel Processing, LNCS 1123,* 2005, 37-52

[Rao et al. 2002] Jun Rao, Chun Zhang, Guy Lohman, Nimrod Megiddo, Automating physical database design in a parallel database, *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data,* 2002, 558-569

[Ringer 2015] Craig Ringer, When are we going to contribute BDR to PostgreSQL?, http://blog.2ndquadrant.com/when-are-we-going-to-contribute-bdr-to-postgresql/, 2015. Viitattu: 14.12.2018.

[Rothnie et al. 1980] James B. Rothnie, Philip A. Bernstein, Stephen Fox, Nathan Goodman, Michael Hammer, Terry A. Landers, Christopher L. Reeve, David W.

Shipman, Eugene Wong, Introduction to a system for distributed databases (SDD-1), *ACM Transactions on Database Systems (TODS)*, 5(1), March 1980, 1-17

[Röhm et al. 2000] Uwe Röhm, Klemens Böhm, Hans-Jörg Schek, OLAP query routing and physical design in a database cluster, *Advances in Database Technology — EDBT 2000. LNCS 1777,* March 2000, 254-268

[Sadat and Lecca 2009] A.B.M.Rubaiyat Islam Sadat, Paola Lecca, On the performances in simulation of parallel databases: an overview on the most recent techniques for query optimization, *International Workshop on High Performance Computational Systems Biology, HIBI '09,* 2009, 113 - 117

[Serafini et al. 2016] Marco Serafini, Rebecca Taft, Aaron J. Elmore, Andrew Pavlo, Ashraf Aboulnaga, Michael Stonebraker, Clay: fine-grained adaptive partitioning for general database schemas, *Proceedings of the VLDB Endowment,* 10(4), 2016, 445-456

[Singh 2018] Ajit Singh, *New York Stock Exchange Oracle Exadata – Our Journey*, http://www.oracle.com/technetwork/database/availability/con8821-nyse-2773005.pdf, 2018. Viitattu: 14.12.2018.

[solidIT 2018] solidIT consulting & software development gmbh, *DB-Engines Ranking*, https://db-engines.com/en/ranking, 2018. Viitattu: 14.12.2018.

[SQL Server Av. Groups 2016] Microsoft Corporation, Overview of AlwaysOn Availability Groups (SQL Server) https://docs.microsoft.com/en-us/sql/database-engine/availability-groups/windows/overview-of-always-on-availability-groups-sql-server, 2016. Viitattu: 14.12.2018.

[SQL Server Replication 2017] Microsoft Corporation, SQL Server Replication https://docs.microsoft.com/en-us/sql/relational-databases/replication/sql-server-replication, 2017. Viitattu: 14.12.2018.

[SQL Server Warm Standby 2012] Microsoft Corporation, How to: Set Up, Maintain, and Bring Online a Warm Standby Server (Transact-SQL), https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008-r2/ms178034(v=sql.105), 2012. Viitattu: 14.12.2018.

[Stack Exchange 2018] Stack Exchange, Inc., *StackOverflow Developer Survey Results 2018*, https://insights.stackoverflow.com/survey/2018/, 2018. Viitattu: 14.12.2018.

[Stonebraker et al. 2007] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, Pat Helland, The end of an architectural era (it's time for a complete rewrite), *Proceedings of the 33rd international conference on Very large data bases,* 2007, 1150-1160

[Taft et al. 2014] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J. Elmore, Ashraf Aboulnaga, Andrew Pavlo, Michael Stonebraker, E-Store - fine-

grained elastic partitioning for distributed transaction processing systems, *Proceedings of the VLDB Endowment,* 8(3), 2014, 245-256

[Taton et al. 2006] Christophe Taton, Sara Bouchenak, Noël De Palma, Daniel Hagimont, Sylvain Sicard, Self-sizing of clustered databases, *International Symposium on a World of Wireless, Mobile and Multimedia Networks, WoWMoM,* 2006, 506-512

[TPC 2019] Transaction Processing Performance Council, www.tpc.org, 2019. Viitattu: 16.1.2019.

[Wang et al. 2011] Daniel L. Wang, Serge M. Monkewitz, Kian-Tat Lim, Jacek Becla, Qserv: a distributed shared-nothing database for the LSST catalog, *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis,* 2011

[Yu et al. 1986] Philip S. Yu, Douglas W. Cornell, Daniel M. Dias and Balakrishna R. Iyer, Analysis of affinity based routing in multi-system data sharing, *Performance Evaluation*, 7(2), June 1987, 87-109

[Özsu and Valduriez 2011] M. Tamer Özsu, Patrick Valduriez, *Principles of Distributed Database Systems* 3rd ed, Springer Science+Business Media, New York, 2011