Pauli Kujala

Korppi-opintotietojärjestelmän tietokantakyselyiden nopeuttaminen

Tietotekniikan
pro gradu -tutkielma
3.7.2008

Jyväskylän yliopisto

Tietotekniikan laitos

Jyväskylä

Tekijä: Pauli Kujala
Yhteystiedot: pauli.kujala@iki.fi
Työn nimi: Korppi-opintotietojärjestelmän tietokantakyselyiden nopeuttaminen
Title in English: Optimizing Database Queries of Korppi Study Information System
Työ: Tietotekniikan pro gradu -tutkielma
Sivumäärä: 137
Tiivistelmä: Tutkielma käsittelee tietojärjestelmän tehostamista tietokantakyselyiden muokkauksen, tietokantaan luotavien indeksien ja sovelluksen lähdekoodin muokkauksen avulla. Tehostamiskeinojen toimivuutta analysoidaan Jyväskylän yliopistossa kehitetyssä ja käytössä olevassa Korppi-opintotietojärjestelmässä.
English abstract: The thesis examines on how an information system can be optimized and tuned by reformulating SQL queries, by analysing database index usage, and by modifying program code. The optimization methods are applied to the Korppi study information system which is developed and used at the University of Jyväskylä.
Avainsanat: Hakemisto, indeksi, Korppi, kysely, nopeuttaminen, opintotietojärjestelmä, optimointi, PostgreSQL, relaatiotietokanta, lähdekoodi, SQL, suorituskyky, suoritussuunnitelma, tehostaminen, tietokannanhallintajärjestelmä, virittäminen.
Keywords: Database management system, index, Korppi, optimization, performance, PostgreSQL, program code, query, query plan, relational database, SQL, study information system, tuning.

Sisältö

1 Johdanto

Isossa organisaatiossa tietokantajärjestelmä (engl. *database system*) on yleensä osa tietojärjestelmää (engl. *information system*), johon kuuluvat kaikki organisaation tietojen keräykseen, hallintaan, käyttöön ja levittämiseen liittyvät resurssit. Näihin resursseihin lasketaan kuuluviksi mm. data, tietokannanhallintajärjestelmä sekä dataa hyödyntävät ja muokkaavat ohjelmistot. Resurssien optimoinnilla pyritään siihen, että tietokantoja hyödyntävät sovellukset suoriutuvat tehtävistään nopeammin, tietokantoihin tallennettuun dataan kohdistettujen kyselyiden vasteajat ovat lyhyempiä ja operaatioiden kokonaissuoritusteho lisääntyy. Tavoitteisiin päästään analysoimalla tietokantoihin tallennetusta datasta ja tietokantojen käytöstä kerättyjä tietokantojen sisäisiä tilastoja sekä optimoimalla indeksien käyttöä, tietokantojen rakennetta ja tietokantoihin kohdistuvia hakuja.

Korppi on monipuolinen Jyväskylän yliopistossa kehitetty ja käytössä oleva, pääosin WWW-selaimella käytettävä opintotietojärjestelmä. Se tarjoaa tietoa ja työvälineitä opettajille ja opintohallinnosta vastaaville sihteereille opetuksen hallintaan sekä opiskelijoille opintoihin ilmoittautumiseen ja opintojen seurantaan. Korppi-järjestelmä sijaitsee WWW-osoitteessa `https://korppi.jyu.fi/`.

Korppi-järjestelmä on ohjelmoitu Java-kielellä. Se hyödyntää PostgreSQL-tietokannanhallintajärjestelmää, Tomcat-nimistä servletti- ja JSP-moottoria sekä Apache-nimistä WWW-palvelinohjelmistoa Linux-käyttöjärjestelmän päällä. Käyttäjän WWW-selaimen tuottamat HTTP- ja HTTPS-pyynnöt Korppi-palvelimelle käsittelee Apache-ohjelmisto, joka välittää dynaamiset JSP-kutsut Tomcat-palvelinohjelmistolle. Tomcat käsittelee pyynnön ja palauttaa vastauksen Apachelle, joka lopulta palauttaa selaimen tuottaman pyynnön tuloksena HTML-sivun käyttäjän selaimelle.

Tutkielmassa tarkastellaan Korppi-opintotietojärjestelmän tietokantaa ja sen käyttötapoja sekä tietokantahakujen nopeuttamista sovelluksen lähdekoodin kehittäjän ja järjestelmän ylläpitäjän kannalta.

Luku 2 esittelee Korppi-järjestelmän historiaa ja tutkielman taustaa sekä tavoitteet ja tutkimusasetelman. Luvussa 3 kuvataan tutkielmassa käytetyt termit ja luvussa 4 tärkeimmät tietokantojen hallintaan liittyvät käsitteet. Luvussa 5 esitellään

tietokantakyselyissä käytetty SQL-kieli.

Luku 6 kuvaa tietokantojen suorituskykyyn vaikuttavat tekijät sekä sen, miten tietokannan suorituskykyä voidaan valvoa. Luvussa käytetään esimerkkinä Korppi-järjestelmää ja sen hyödyntämää PostgreSQL-tietokannanhallintajärjestelmää.

Luvun 7 tehostuskeinoja ja -menetelmiä voidaan hyödyntää SQL-kyselylauseisiin ja sovelluksen lähdekoodiin tehtävillä muutoksilla. Luvussa 8 tarkastellaan muutamaa Korppi-järjestelmän toiminnan tehostamisesimerkkiä. Kyseisen luvun lopussa luvussa 8.7 esitellään toimiviksi ja toimimattomiksi havaitut toteutusratkaisut ja suositukset.

Luku 9 esittelee lyhyesti muutamia tutkielman aihepiiriin liittyviä jatkotutkimukseen soveltuvia aiheita. Luvussa 10 esitellään tutkielman yhteenveto.

2 Taustaa, tavoitteet ja tutkimusasetelma

Luvussa kuvataan tutkielman tavoitteiden ja tutkimusasetelman lisäksi Korppi-järjestelmän historiaa, sovelluksia ja osioita. Luku perustuu lähteisiin [6, s. 554–558], [8], [10] ja [11, s. 260–275].

2.1 Tietokanta tietojärjestelmän osana

Elmasrin [6, s. 530] mukaan isossa organisaatiossa tietokantajärjestelmä (engl. *database system*) on yleensä osa **tietojärjestelmää** (engl. *information system*), johon kuuluvat kaikki organisaation tietojen keräykseen, hallintaan, käyttöön ja levittämiseen liittyvät resurssit. Näihin resursseihin Elmasri laskee kuuluviksi datan, tietokannanhallintajärjestelmän, tietojärjestelmän laitteiston ja tallennusmedian, dataa hyödyntävän ja hallinnoivan henkilöstön (mm. tietokannan valvojat ja loppukäyttäjät), dataa hyödyntävät ja muokkaavat ohjelmistot sekä ohjelmoijat, jotka kyseisiä ohjelmistoja kehittävät.

Mullinsin [11, s. 260] mukaan tietokantaa hyödyntävä järjestelmä käyttää palvelinlaitteiston resursseja, kuten käyttöjärjestelmää (engl. *operating system*), verkko-ohjelmistoa (engl. *networking software*) ja viestinvälitysjärjestelmiä (engl. *message queuing systems*). Mullins määrittelee **järjestelmän optimointiin** kuuluviksi asennuksen (engl. *installation issues*), asetusten hallinnan (engl. *configuration issues*) sekä järjestelmän osien integrointiin liittyvät kysymykset (engl. *integration issues*).

Mullinsin [11, s. 260] mukaan mikään tietojärjestelmään kuuluva **sovellus tai tietokanta ei hyödy** sovellukseen tai tietokantaan kohdistuvasta **optimoinnista, mikäli palvelinlaitteiston resurssit eivät ole riittävät** tai resursseja on asennettu ja hallittu puutteellisesti. Mullinsin [11, s. 268–271 ja 274–275] mukaan käyttöjärjestelmän hyödynnettävissä olevan keskusmuistin määrän tulee olla riittävä, ja lisäksi sen tulee olla tasapainossa tietokannanhallintajärjestelmän käyttöön annettavan muistin määrän kanssa. Lisäksi levymuistien tyyppien tulee soveltua tietojärjestelmään kuuluvien sovellusten ja tietokantojen tehokkaaseen käyttöön. Levymuistin kohdentamisen muun muassa tietokannanhallintajärjestelmän ja tietokantojen käyt-

töön tulee olla järkevää. Samaten palvelinlaitteiston laskentatehon tulee olla optimaalinen kokonaisuuteen nähden.

Elmasrin [6, s. 554] mukaan **resurssien optimoinnilla pyritään siihen, että tietokantaa hyödyntävät sovellukset suoriutuvat tehtävistään nopeammin, tietokannan dataan kohdistettujen kyselyiden vasteajat ovat lyhyempiä ja operaatioiden kokonaissuoritusteho lisääntyy**. Tavoitteisiin päästään Elmasrin [6, s. 555–558] mukaan analysoimalla tietokantaan tallennetusta datasta ja tietokannan käytöstä kerättyjä tietokannan sisäisiä tilastoja sekä optimoimalla indeksien käyttöä, tietokannan rakennetta ja tietokantaan kohdistuvia hakuja.

Mullinsin [11, s. 270–271] mukaan yksi suurimmista pullonkauloista tietokantojen suorituskyvyssä johtuu **levyoperaatioiden runsaasta määrästä**. Hänen mukaansa levyoperaatioiden määrän väheneminen tai suoritusaikojen lyheneminen saattaa vaikuttaa suotuisasti tietokannan suorituskyvyn paranemiseen. Mullinsin [11, s. 298] mukaan hakuja tehostavien indeksien luominen tietokannan tauluihin (ks. luku 5.6) on tehokkaimpia keinoja, joilla tietokannan suorituskykyä voidaan tehostaa.

2.2 Tutkielman tavoitteet

Korppi-opintotietojärjestelmää on kehitetty usean opiskelijaprojektiryhmän ja muutaman palkatun kehittäjän voimin (ks. luku 2.5). Kehitystyö on tapahtunut järjestelmän alkuaikoina pääosin ns. kädestä suuhun -periaatteella. Kehittäjien kohtuullisen vähäinen optimointiosaaminen ja pidemmän aikavälin suuntaviivojen puutteellinen kohdentaminen ovat johtaneet siihen, että kehitystyössä kiireisellä aikataululla toteutetut ratkaisut eivät aina ole aikaa myöten pysyneet tehokkaina. Korppi-järjestelmän tietokantaan tallennetun datan määrä on käyttäjien ja järjestelmän laajentuneen hyödyntämisen myötä lisääntynyt valtavasti vuosien varrella (ks. liite A), joten järjestelmän toteutusratkaisuja on ajan saatossa jouduttu optimoimaan jouhevan käytön turvaamiseksi.

Tutkielman tavoitteena on kartoittaa keinoja tietokantojen tehokkaampaan hyödyntämiseen keskittyen erityisesti tietokantaan luotaviin indekseihin sekä tietokantakyselyiden ja sovelluksen lähdekoodin muokkaukseen. Tutkielmassa keskitytään Jyväskylän yliopistossa kehitettyyn ja käytössä olevaan Korppi-opintotietojärjestelmään, mutta suurin osa havainnoista ja suosituksista yleistyy käytettäväksi mihin tahansa tietokannanhallintajärjestelmään ja tietokantaan.

Korppi-järjestelmän palvelinsovellusten kirjoittamien lokitiedostojen sekä järjestelmän tietokannan käyttötapoja kuvaavien tilastojen avulla voidaan selvittää mahdolliset pullonkaulat (ks. luku 6.5). Kun lokitiedostojen ja käyttötilastojen avulla on selvitetty tehottomia ja aikaavieviä SQL-kyselyitä, voidaan näitä analysoida tarkemmin luvussa 6.3 kuvatuin keinoin. Tarkemman analysoinnin tuloksena saatetaan päätyä vaikkapa siihen, että tietokannan tauluihin tulisi luoda tietojen hakua nopeuttavia indeksejä (ks. luku 5.6). Jos pelkkä indeksien luonti ei riitä nopeuttamaan hakua riittävästi, toisinaan joudutaan muokkaamaan SQL-kyselyä tehokkaammaksi (ks. luku 7.1). Tämä onnistuu vaikkapa yhdistämällä alikysely osaksi pääkyselyä taikka jakamalla iso kysely yksinkertaisempiin ja selkeämpiin osiin alikyselyiden avulla.

Jos SQL-kyselyn optimointi ei nopeuta toiminnon suorittamista tarpeeksi, on mahdollista muokata sovelluksen lähdekoodin algoritmeja ja tietorakenteita tehokkaammiksi. Tällöin vaikkapa iso ja monimutkainen SQL-kysely voidaan yleensä jakaa suoritettavaksi pienemmissä palasissa, joiden tulokset kootaan varsinaiseksi lopputulokseksi sovelluksen lähdekoodin keinoin (ks. luku 7.2). Myös tehokkaampia tietorakenteita ja tietokantakyselyiden välimuistiratkaisuja voidaan hyödyntää lähdekoodia muokkaamalla (ks. luvut 7.3 ja 7.4).

Tutkielmaan on dokumentoitu Korppi-järjestelmän kehittämisen aikana olennaisia toimiviksi ja toimimattomiksi havaittuja menetelmiä. Tietokantojen hyödyntämiseen liittyvän teorian, luvun 8 tehostamisesimerkkien sekä luvun 8.7 havaintojen ja suositusten toivotaan auttavan muun muassa Korppi-järjestelmän jatkokehittäjiä tehostamaan tietokannan hyödyntämistä.

2.3 Tutkimusasetelma

Tutkielman alkuperäisenä aiheena syksyllä 2003 oli selvittää keinoja ja periaatteita Korppi-opintotietojärjestelmän tietokannan hajauttamiseksi. Tutkielman ohjaajan kanssa joulukuussa 2003 käydyn keskustelun jälkeen aihetta muutettiin siten, että tutkielmaan dokumentoidaan Korppi-järjestelmän kehityksen aikana hyödynnettyjä tehostamismenetelmiä. Lisäksi tutkielmaan kirjataan joitakin olennaisia virheellisiksi tai muuten toimimattomiksi havaittuja toteutusratkaisuja, jotta näitä voidaan välttää Korppi-järjestelmän jatkokehityksessä.

Tietokantojen hyödyntämiseen liittyvän teorian pohjalta on luvussa 8 käytännössä tarkasteltu muutamaa Korppi-järjestelmän toiminnan tehostamisesimerkkiä.

Kuhunkin esimerkkiin on kuvattu yrityksen ja erehdyksen sekä runsaiden erilaisia lähestymistapoja hyödyntävien kokeilujen kautta löydetty kyseisen esimerkin lähtökohtana olevan suorituskykyongelman ratkaiseva tehostamistoimenpide.

Korppi-järjestelmän tuotantokäytössä ilmenneitä suorituskykyongelmia on havaittu sekä järjestelmän dynaamisten WWW-sivujen hitaana suorituksena että jälkikäteen lokitiedostojen analysoinnin yhteydessä. Suorituskykyongelmia on ensin pyritty korjaamaan tietokannan rakenteisiin kohdistuvin muutoksin, erityisesti tauluihin luotavien indeksien (ks. luku 5.6) avulla. Jos pelkästään indeksien avulla ei suorituskykyä ole saatu parannettua riittävästi, on toisinaan jouduttu muokkaamaan järjestelmän suorittamia SQL-kyselyitä tai Java-lähdekoodia tehokkaammaksi. Suorituskykyä lisääviä ratkaisuja on selvitetty lähdemateriaalin avulla, kuten PostgreSQL-tietokannanhallintajärjestelmän dokumentaatiosta [13] ja postituslistoilta [14].

Tutkimusasetelma koostuu siten suurimmaksi osaksi Korppi-järjestelmän tuotantokäytön kautta esiintulleista suorituskykyongelmista, niiden analysoinnista sekä järjestelmän tietokantaan ja lähdekoodiin toteutetuista suorituskykyä parantavista muutoksista.

SQL-kyselyiden analysointiin on tuotantoympäristön lisäksi käytettävissä testausympäristö, johon on asennettu PostgreSQL-tietokannanhallintajärjestelmän versiot 7.1, 7.3 ja 7.4. Kuhunkin näistä oli asennettuina samat tietorakenteet ja saman datan käsittävä tietokanta, jolloin voitiin tarkastella tietokannanhallintajärjestelmän eri versioiden vaikutusta yksittäisten SQL-kyselyiden tehokkuuteen ja suorittamistapaan.

2.4 Korppi-opintotietojärjestelmän sovellukset ja osiot

Korppi-järjestelmä koostuu PostgreSQL-tietokannanhallintajärjestelmästä ja sen avulla toteutetun tietokannan sisältämästä datasta, Tomcat-nimisestä servletti- ja JSP-moottorista sekä Apache-nimisestä WWW-palvelinohjelmistosta. Java-kielen avulla on ohjelmoitu Korppi-järjestelmän sovellukset, jotka ovat toisiinsa nähden enemmän tai vähemmän erillään.

Kurssihallinnan avulla laitosten ja tiedekuntien opettajat ja hallintohenkilökunta voivat hallita kurssien kuvauksia, opetustapahtumien aikatauluja ja salivarauksia, ilmoittautumisia ja suorituksia. Opiskelijat voivat ilmoittautua kursseille ja niiden opetusryhmiin sekä tentteihin. Lisäksi opiskelijat voivat seurata opintojensa

edistymistä kurssien merkintöjen ja opintosuoritusotteen avulla. Erillisen Kaakkuri-sovelluksen avulla voidaan laitoksen **opetuksen sijoittelu tilavarauksineen** suunnitella visuaalisesti graafisen liittymän avulla vaikkapa koko lukuvuodeksi. Suunnitellut kurssien aikataulut voidaan tallentaa Korppi-järjestelmän tietokantaan. **Kalenterista** jokainen voi tarkistaa päivä-, viikko-, kuukausi- ja vuosilukujärjestyksensä sekä merkitä uusia henkilökohtaisia tai henkilöryhmien tapahtumia. Synkronointitoiminnon avulla tapahtumat on helppo tahdistaa (engl. *synchronize*) esimerkiksi matkapuhelimen kalenterin ja Korppi-järjestelmän välillä.

Salivaraukset sekä salivarauspyynnöt ja niiden vahvistaminen kytkeytyvät niin opetuksen järjestämiseen, opiskelijoiden lukujärjestysten ylläpitämiseen kuin muuhunkin yliopiston yksiköiden toimintaan. **Karttaosion** avulla voidaan tarkastella yliopiston rakennusten karttoja sekä salien, huoneiden ja muiden tilojen paikkatietoja.

Opinnäytteiden toteutus- ja ohjausprosessin läpivientiä ja niiden tehtäviä voidaan hallita tarkoitusta varten toteutetulla osiolla. **eHOPS-osion** avulla opiskelijat voivat suunnitella, aikatauluttaa ja hyväksyttää henkilökohtaisen opintosuunnitelmansa laitoksen määrittämien opetussuunnitelmien pohjalta. Lisäksi osion tarjoaman ylläpitokäyttöliittymän avulla voivat laitoksen opintorakenteista vastaavat sihteerit muokata opetussuunnitelmia ja malliopintosuunnitelmia.

Kyselyosion avulla voidaan luoda kyselyitä vaikkapa kurssien palautteita, opinnäytteiden tiedonkeräystä tai henkilökohtaisten opintosuunnitelmien pohdiskeleviin kysymyksiin vastaamista varten. Kyselyiden analysointitoiminnon avulla voidaan luoda yksinkertaisia raportteja kyselyiden vastauksista. Kyselyn vastaukset voidaan myös siirtää jatkokäsiteltäväksi vaikkapa SPSS-sovellukseen sekä taulukkolaskentaohjelmiin.

Organisaatioyksikön henkilöstön ja opiskelijoiden tietoja voidaan hallita **henkilöhallinnan** avulla. Henkilöitä voidaan koota **ryhmäksi**, jolla on jokin yhteinen tavoite tai ominaisuus. Esimerkiksi tiettynä vuonna tietyssä pääaineessa aloittaneista opiskelijoista voidaan muodostaa ryhmä, jolle merkitään yhteisiä tapaamisia kalenteriin. **Ylläpito-osiossa** voidaan mm. muokata opintojaksojen ja organisaatioiden tietoja sekä jakaa laitoksen henkilökuntaan kuuluville henkilöille oikeuksia.

Sähköpostilistojen avulla voidaan hoitaa tiedotus. Postilista voidaan luoda kurssille tai mille tahansa henkilöryhmälle, jolloin kurssin tai ryhmän jäsenistön päivittyessä myös postilistan jäsenistö päivittyy automaattisesti. Postilistalle lähetetyt viestit voidaan määrätä tallentumaan myös arkistoon. Julkista postilista-arkistoa

voidaan selata täysin vapaasti, kun taas salaiseen arkistoon pääsevät käsiksi vain postilistalle kuuluvat jäsenet. **Ilmoitustaulun** avulla voidaan hoitaa tiedotusta.

Tietokanta- ja Java-luokkatasolla Korppi-järjestelmä koostuu seuraavista osittain päällekkäisistä kokonaisuuksista:

- opintokokonaisuudet, opintojaksot, kurssit, opetusryhmät ja tentit sekä yksittäiset opetustapahtumat,
- opinnäyteaiheet ja opinnäytteet,
- opetussuunnitelmat ja opintosuunnitelmat (OPS, JOPS ja HOPS),
- henkilöiden tiedot ja henkilöryhmät,
- tapahtumien kalenteri aikataulu- ja paikkatietoineen,
- tapahtumien synkronointi,
- kurssien ja henkilöryhmien sähköpostilistat arkistoineen,
- ilmoitustaulu ja muu järjestelmän sisäinen tiedotus,
- kyselyt, kyselypankki ja kyselyjen analysointi,
- salit ja salivaraukset sekä rakennusten kerroskartat,
- organisaatiot,
- rajapinnat sekä
- oikeushallinta.

Osa kokonaisuuksista on sidoksissa hyvin vahvasti muihin kokonaisuuksiin. Esimerkiksi **henkilöt** liittyvät olennaisesti kursseihin ja tentteihin, joissa **henkilöiden rooleina** voivat olla muiden muassa opiskelija, opettaja, tenttijä, tentaattori, tenttien tarkastaja ja tenttitilaisuuden valvoja. Toisaalta henkilöt liittyvät olennaisesti myös opinnäytteisiin, joissa rooleina voivat olla muiden muassa tekijä, ohjaaja, tarkastaja, hyväksyjä ja opintoneuvoja. **Henkilöt ovat ryhmien jäseniä**, ja kun ryhmille luodaan yhteisiä tapahtumia, jäsenet voivat halutessaan varmistaa osallistumisensa tai osallistumattomuutensa tapahtumiin.

Kurssit ja niiden **opetusryhmät** liittyvät kurssihallinnan ohella henkilöiden **kalentereihin** luentoajankohtina, tentteinä sekä muina opetustilaisuuksina. Kurssien opetusryhmien tapahtumille useimmiten tehdään myös **salivaraus** johonkin Jyväskylän yliopiston viralliseen saliin, jota kautta kurssit liittyvät olennaisesti myös yliopiston tilapalveluiden prosesseihin. Saleittain tai saliryhmittäin nimetyt henkilöt vahvistavat järjestelmän kautta tehdyt salivarauspyynnöt.

Kyselyillä on aina joku yksittäinen laatija sekä mahdollisesti useita vastaamaan oikeutettuja henkilöryhmiä. Kyselyiden vastauksista tuotettuja raportteja voivat tar-

kastella ne henkilöryhmät, joille on annettu vastausten tai raporttien tarkasteluoikeus.

Henkilötietojen ja henkilöryhmien ohella myös organisaatiotiedot, opintojaksoja kurssitiedot sekä kalenteritiedot liittyvät jollakin tavalla lähes jokaiseen Korppi-järjestelmän osioon.

2.5 Korppi-opintotietojärjestelmän kehityshistoriaa

Jyväskylän yliopistossa toteutettiin keväällä 1998 WWW-pohjainen **Kurki-kurssikirjanpitojärjestelmä** tietotekniikan opiskelijaprojektina. Projektissa käytettiin tietokannanhallintajärjestelmänä Microsoft Accessia sekä kehitys- ja käyttöympäristönä Borland IntraBuilderia. Tutkielman kirjoittaja oli yhtenä kolmesta opiskelijasta toteuttamassa kyseistä kurssikirjanpitojärjestelmää. Kirjoittaja jatkoi Kurki-järjestelmän jatkokehittämistä tietotekniikan harjoitteluna ja erikoistyönä sekä palkattuna työntekijänä. Tiedot perustuvat Kurki- ja Korppi-järjestelmien historiaa esittelevään WWW-sivuun [8].

Tarpeiden lisääntyessä sekä Kurki-järjestelmässä esiintyneiden käytettävyyspuutteiden ja ongelmia aiheuttaneen toteutustekniikan takia järjestelmää ei kannattanut enää laajentaa. Syksyllä 2000 päätettiin toteuttaa alusta lähtien uudelleensuunniteltu järjestelmä, joka olisi laajennettava ja käytettävä. Uuden järjestelmän toteutustekniikan ja -alustan tulisi olla standardeja ja yleisesti käytettyjä. Kirjoittaja on ollut alusta saakka mukana kyseisen Kurki-järjestelmän seuraajan suunnittelussa, kehityksessä ja tukitoiminnoissa.

Syksyllä 2000 Kotka-opiskelijaprojekti keskittyi **järjestelmän tietokannan rakenteen suunnitteluun ja toteutukseen**. Projektiryhmä myös kartoitti ja testasi sopivia toteutustekniikoita, tietokanta- ja WWW-palvelimia sekä ohjelmointikieliä. Tietokannanhallintajärjestelmäksi valittiin PostgreSQL, palvelinkäyttöjärjestelmäksi Linux, ohjelmointikieleksi Java sekä palvelinohjelmistoiksi Apache ja Tomcat. Projektiryhmä toteutti järjestelmään **henkilötietomoduulin prototyypin**.

Korppi-opiskelijaprojekti jatkoi Kotka-projektin työtä keväällä 2001 kehittämällä järjestelmään **kurssikirjanpito-osion**. Tietokannan rakenteen oli Kotka-projekti määritellyt kohtuullisen toimivaksi, mutta rakennetta jouduttiin muokkaamaan jonkin verran. Korppi-projekti määritteli, suunnitteli ja toteutti WWW-käyttöliittymät opiskelijoille, opettajille ja ylläpitäjälle. Käyttöliittymät toteutettiin prototyyppiasteelle sovellusarkkitehtuurin ja perustekniikoiden osalta. Aikaansaa-

tu Korppi-sovellus korvasi vielä tuolloin käytössä olleen Kurki-järjestelmän täysin. Uusi järjestelmä tunnettiin aluksi Kotka-nimisenä, mutta kevään 2001 Korppi-sovellusprojektin myötä vähitellen laajeneva käyttäjäkunta omaksui järjestelmälle Korppi-nimen.

Kesällä ja syksyllä 2001 Korppi-järjestelmää jatkokehitettiin havaittujen puutteiden ja saadun palautteen pohjalta. Kesän 2001 palkatut kehittäjät viimeistelivät ulkoasua ja kehittivät järjestelmän rajatun käyttäjäkunnan käyttöön. Syksyllä järjestelmän otti pilottikäyttöön muutama tietotekniikan ja tietojenkäsittelytieteen opettaja luennoimillaan kursseilla.

Tietotekniikan opiskelijaprojekteina järjestelmän kehittämistä jatkoivat Kolibri-projekti syksyllä 2001 lisäten **päivyrisovelluksen**, Koppelo-projekti keväällä 2002 toteuttaen **opinnäyteosion** sekä Kiuru-projekti syksyllä 2002 lisäten **salivaraussovelluksen**.

Keväällä 2003 Kottarainen-opiskelijaprojekti toteutti järjestelmään **kyselyjen laatimis- ja hallintasovelluksen** prototyypin. Syksyllä 2003 Koskikara-opiskelijaprojekti jatkoi kyselyosion kehittämistä liittämällä siihen **kyselypankin ja vastausten analysointityökalun**.

Syksyllä 2003 Käki-opiskelijaprojekti suunnitteli henkilökohtaisten **opintosuunnitelmien** (eHOPS) laadinta-, seuranta- ja raportointisovelluksen. Projekti toteutti sovelluksesta sihteerin käyttöliittymän, jolla voidaan koostaa Korppi-järjestelmässä olevista kursseista opintokokonaisuuksia ja opintokokonaisuuksista tutkintovaatimuksia opiskelijoiden opiskelusuunnitelmien pohjaksi. Kuikka-sovellusprojekti jatkoi keväällä 2004 Käki-projektin aloittamaa ohjausosiota toteuttaen käyttöliittymän opiskelijan eHOPSien laadintaan.

Kyyhky-sovellusprojekti lisäsi keväällä 2004 järjestelmään **kalenterin synkronointirajapinnan**. Kyseisen rajapinnan avulla pystytään synkronoimaan kalenteritapahtumia Korppi-järjestelmän kalenteriosion sekä erilaisten mikrojen, matkapuhelimien, taskumikrojen ja muiden kämmenlaitteiden kalenterien kesken.

Kaakkuri-opiskelijaprojekti toteutti keväällä 2005 prototyypin erillisestä työpöytäsovelluksesta, jonka avulla voidaan laatia laitoksen kaikkien kurssien **lukuvuoden luentolukujärjestys**.

Kuovi-opiskelijaprojekti suunnitteli ja toteutti syksyllä 2005 Korppi-järjestelmän opiskelijan keskeisimpien toimintojen (kurssi- ja tentti-ilmoittautumiset sekä kalenteri) muuntamisen **matkapuhelimissa paremmin toimiviksi**. Keväällä 2006 Kiiruna-opiskelijaprojekti toteutti **karttarajapinnan** Korppi-järjestelmän päälle. Ky-

seisen rajapinnan avulla pystytään selvittämään mm. tilojen sijainti ja käytettävyys. Syksyllä 2007 Koskelo-sovellusprojekti suunnitteli ja toteutti Korppi-järjestelmään **kurssien opetusryhmien ja opetustapahtumien hallintaan** nykyistä sovellusta **käytettävämmän käyttöliittymän**.

Sovellusprojektien lisäksi järjestelmää on vuosien saatossa kehittänyt lukuisa joukko palkattuja kehittäjiä, joista muutama on työskennellyt järjestelmän kehityksen ja tuen parissa yli puoli vuosikymmentä. Muutama arvokas kehittäjä- ja tukiresurssikin on menetetty heidän siirryttyään muualle uusien haasteiden pariin.

2.6 Korppi-opintotietojärjestelmän palvelinhistoriaa

Tiedot perustuvat Kurki- ja Korppi-järjestelmien palvelintekniikkaa esittelevään WWW-sivuun [10]. Kurki-järjestelmä toimi elinkaarensa lopussa vuonna 2001 Windows NT -palvelimella, joka pystyi palvelemaan noin 40 yhtäaikaista käyttäjää.

Korppi-järjestelmän alkuaikoina syksystä 2000 aina tammikuuhun 2004 asti kaikki käyttäjien WWW-selainten kautta järjestelmään kohdistuneet pyynnöt on käsitelty **yhden sovelluspalvelimen avulla**, ja myös tietokanta on fyysisesti sijainnut kyseisellä palvelimella. Käyttäjämäärän ja järjestelmän käytön lisääntyessä tämä järjestely havaittiin riskialttiiksi sekä satunnaisesti tehottomaksi ja ongelmia aiheuttavaksi.

Tammikuussa 2004 tuotantokäyttöön asennettiin **palvelinpari**, joka oli ensimmäinen askel kohti hajautettua järjestelmää. Tietokantapalvelimen ollessa erillään sovelluspalvelimesta kumpikin palvelinlaitteisto voitiin optimoida omaan tehtäväänsä. Lisäksi kumpikin palvelinmikro oli täsmälleen identtinen, jolloin toisen rikkoutuessa olisi ollut mahdollista väliaikaisesti siirtää toisen palvelimen toiminnot jäljelle jääneeseen palvelimeen. Tuotantokäyttöön otettiin tuolloin PostgreSQL-tietokannanhallintajärjestelmän versio 7.3 aiemmin käytössä olleen version 7.1 tilalle. Korppi-järjestelmä on pystynyt palvelemaan vuonna 2004 käyttöönotetun palvelinparin myötä useampia satoja yhtäaikaisia käyttäjiä.

Tammikuussa 2006 käyttöön otettiin **tehokkaampi tietokantapalvelin**, jossa oli huomattavasti **enemmän muistia** kuin aiemmin käytössä olleessa laitteistossa. Prosessoritehoa ei sinällään juurikaan tullut lisää, mutta lisääntyneen muistimäärän ansiosta yhtäaikaisia pyyntöjä saatiin suoritettua tehokkaammin. Käytännössä koko tietokanta pysyi tällä laitteistolla kaiken aikaa käyttöjärjestelmän muistissa, ja vain tietojen muutokset tallennettiin fyysiselle levymuistille. Samalla uudessa pal-

velinlaitteistossa otettiin käyttöön PostgreSQL-tietokannanhallintajärjestelmän versio 7.4, joka myös toi omalta osaltaan lisää tehokkuutta järjestelmän käyttöön.

Kyseinen tietokantapalvelin on käytössä keväällä 2008, mutta se on tarkoitus korvata kesällä 2008 uudella palvelinlaitteella. Samalla käyttöön otetaan uudempi ja tehokkaampi PostgreSQL:n versio 8.1.

3 Termejä

Luvussa esitellään tutkielmassa käytettäviä termejä. Luku perustuu pääosin lähteisiin [32] ja [7].

3.1 Palvelinohjelmistoihin ja tekniikkaan liittyviä termejä

Korppi-järjestelmässä käytössä oleviin palvelinohjelmistoihin ja tekniikkaan liittyviä termejä ovat seuraavat:

Apache on avoimen lähdekoodin WWW-palvelinohjelmisto, joka palauttaa asiakkaan HTTP-protokollalla pyytämän HTML-sivun.

HTML (Hypertext Markup Language) on WWW-sivujen rakennetta kuvaava kieli.

HTTP (HyperText Transfer Protocol) on WWW-tekniikassa käytettävä tiedonsiirtoprotokolla, jolla asiakkaana toimiva selain pyytää haluttuja sivuja WWW-palvelimelta.

HTTPS (Hypertext Transfer Protocol over Secure Socket Layer) on HTTP-protokollan salattu versio.

Java on Sun Microsystemin kehittämä laitteistoriippumaton oliopohjainen ohjelmointikieli.

JDBC (Java Database Connectivity) on Java-teknologian käyttämä tietosilta erilaisiin tietokantoihin.

JSP (Java Server Pages) on skriptaustyylinen ohjelmointikieli, jossa HTML-koodin sekaan on mahdollista lisätä Java-kielellä kirjoitettua koodia.

PostgreSQL on avoimeen lähdekoodiin perustuva relaatiotietokannanhallintajärjestelmä.

Selain (WWW-selain) on tietokoneohjelma, joka on tarkoitettu Internetin WWW-sivujen selaamiseen. Selain hakee WWW-sivun HTTP-protokollan avulla WWW-palvelimelta ja tulkkaa sen sisältämät HTML-elementit esitettävään muotoon.

Servletti (engl. *servlet*) eli sovelma on pieni ohjelma, joka on toteutettu ohjelmointikielellä laajentamaan palvelimen toiminnallisuutta. Servletit ajetaan palvelimella.

Tomcat on ohjelmisto, jota käytetään servletti- ja JSP-moottorina Apache-palvelimella. Se välittää asiakaspyynnön servletille ja toimittaa sen tuottaman vastauksen takaisin pyytäjälle.

WWW (World Wide Web) on Internetissä toimiva hypertekstijärjestelmä. Hypertekstiä selataan selaimella, joka hakee sivuiksi kutsuttuja dokumentteja WWW-palvelimilta ja esittää niitä käyttäjälle.

3.2 Tietokannanhallintajärjestelmiin ja tietokantoihin liittyviä termejä

Tietokannanhallintajärjestelmiin tai tietokantoihin liittyvät seuraavat termit:

Alikysely eli sisäkysely (engl. *subquery, subselect*) on kyselylause, joka on kirjoitettu toisen kyselylauseen sisään.

Alkeisarvo (engl. *atomic value*) on arvo, jota ei voida jakaa pienempiin osiin. Alkeisarvo ei siten ole lista arvoista eikä arvojen yhdiste (engl. *composite value*). (Tarkalleen ottaen alkeisarvon alkuperäinen määritelmä on monitulkintainen, koska vaikkapa liukuluku voidaan jakaa kokonaisosaan ja desimaaliosaan.)

Arvo (engl. *value*) on alkio, joka on esitettävissä tietokannassa.

Avainehdokas (engl. *candidate key*) on taulun yksittäinen kenttä tai minimaalinen kenttien joukko, jonka eri riveillä olevat arvot tai arvoyhdistelmät eroavat toisistaan. Taulun perusavaimeksi valitaan yksi avainehdokkaista.

Data (engl. *data*) on tallennettavissa olevia tunnettuja tosiasioita, tekstiä, grafiikkaa, kuvia, ääntä tai videoita, joilla on merkitystä käyttäjien ympäristössä.

Eheys (engl. *integrity*) on tietokannan tila, jossa kaikki tietokantaan liittyvät rajoitukset ja säännöt ovat voimassa.

Indeksi eli hakemisto (engl. *index*) on sarakkeen tai sarakeyhdistelmän perusteella muodostettu aputietorakenne, jonka tarkoituksena on nopeuttaa tiedonsaantia.

Hakuehto (engl. *search condition*) on kriteeri, jonka perusteella tietoja etsitään.

Johdettu taulu (engl. *derived table*) on taulu, joka saadaan suoraan tai epäsuorasti yhdestä tai useammasta muusta taulusta.

Korrelaatioviite (engl. *correlated reference*) eli kytkös on alikyselystä tehty viittaus ulommassa kyselyssä esiteltyyn tauluun. Korrelaatioviitteitä käytetään mm. `EXISTS`-rakennetta käyttävissä alikyselyissä (ks. luku 5.4).

Kysely (engl. *query*) on relaatiotietokannan käsittelykielellä esitetty tulostaulun määrittely muiden taulujen avulla.

Liipaisin eli herätin (engl. *trigger*) on mekanismi, jolla voidaan automaattisesti käynnistää haluttuja tarkistus- ja päivitystoimintoja muiden tietokantatoimintojen vaikutuksesta tai tiettynä ajankohtana.

Liitos (engl. *join*) on relaatioalgebran operaatio, jossa tulostaulu muodostetaan ottamalla kahden lähtötaulun ristitulosta ne rivit, jotka täyttävät operaatiossa määritellyn ehdon.

Liitosehto (engl. *join condition*) on liitoksissa käytetty vastinsarakkeiden vertailuehto.

Kuvaustieto (engl. *metadata*) on tietokannassa olevaa dataa, jonka avulla kuvataan varsinaisen datan ominaisuudet ja tuotetaan asiayhteys, johon data liittyy.

Näkymä eli näkymätaulu (engl. *view*) on johdettu taulu, jolle on annettu nimi.

Optimoija (engl. *optimizer*) on tietokannanhallintajärjestelmän osa, joka valitsee edullisimman saantipolun tietokantaoperaation suorittamiseen.

Perusavain (engl. *primary key*) on yhden tai useamman taulussa olevan kentän yhdistelmä, jonka arvojen avulla rivi yksilöidään muihin taulun riveihin nähden.

Puuttuva arvo eli tyhjä arvo (engl. *NULL value*) on merkintä siitä, että sarakearvo ei ole tiedossa tai sitä ei ole olemassa.

Rajoite (engl. *constraint*) on tietokannan sisältöä rajoittava ehto. Esimerkiksi eheysrajoitteen (engl. *integrity constraint*) avulla määritellään tietokannan perustauluissa yhtäaikaa sallitut rivit eli tietokannan sallitut tilat.

Relaatiotietokanta (engl. *relational database*) on relaatiomalliin perustuvan tietokannanhallintajärjestelmän avulla toteutettu tietokanta.

Ristitulo (engl. *cartesian product*, *cross join*, *quadratic join*) on operaatio, jossa tulostaulu muodostetaan kahdesta lähtötaulusta ottamalla mukaan kummankin lähtötaulun kaikki sarakkeet ja muodostamalla rivit siten, että ensimmäisen lähtötaulun kukin rivi kombinoidaan toisen lähtötaulun kunkin rivin kanssa.

Rivi (engl. *row*) eli tietue (engl. *record*) sisältää tietokannan tauluun tallennettujen arvojen listan, jossa esiintyy yksi tietyn arvoalueen arvo kutakin rivin määrittelyssä osoitettua arvoaluetta kohti näiden järjestyksessä.

Saantipolku (engl. *access path*) on optimoijan SQL-kyselyille valitsema hakutapa, jossa hyödynnetään mm. indeksejä sekä erilaisia taulujärjestyksiä, lajitteluita ja liitostekniikoita.

Samarakenteisuus eli yhdistettävyys (engl. *union compatibility*) on joihinkin tietokantaoperaatioihin liittyvä vaatimus siitä, että operaation läh-

tötaulut ovat rakenteeltaan samanlaisia ts. lähtötauluissa on sama määrä sarakkeita ja vastinsarakkeilla on samat arvojoukot.

Sarake (engl. *column*) eli kenttä (engl. *field*) sisältää tietokannan taulussa eri riveillä samaan arvoalueen määrittelyyn liittyvien arvojen listan.

Selaus eli läpiluku (engl. *scan*) on haku, jossa käydään läpi taulun kaikki rivit.

SQL (Structured Query Language) on tietokantojen hallintaan kehitetty standardoitu kieli.

Suoritussuunnitelma (engl. *query plan*) on suunnitelma, joka sisältää kuvauksen tietokantaoperaation suorittamiseen käytetyistä saantipoluista.

Taulu (engl. *table*) on samaan rivin määrittelyyn liittyvien rivien monijoukko. Taulu esitetään usein käyttäjälle siten, että rivit ovat allekkain vaakasuunnassa ja sarakkeet pystysuunnassa.

Tieto (engl. *information*) on dataa, jonka käsittely lisää dataa käyttävän henkilön tietämystä.

Tietokannanhallintajärjestelmä (engl. *database management system*, DBMS) on ohjelmisto, jonka avulla hallitaan tietokantoja.

Tietokannan hoitaja tai valvoja (engl. *database administrator*, DBA) on henkilö tai yksikkö, jonka tehtävänä on vastata tietokannanhallintajärjestelmän ja tietokantojen toimivuudesta.

Tietokannan tehostaminen (engl. *database tuning*) jatkuu niin kauan kuin tietokantaa käyttävässä sovelluksessa ilmenee suorituskykyongelmia tai tietokantaan kohdistuvat vaatimukset muuttuvat.

Tietokanta (engl. *database*) on järjestetty kokoelma toisiinsa liittyvää dataa, joka on sijoitettu tietokantatauluihin.

Toistuva rivi (engl. *duplicate row*) on tilanne, jossa taulussa on kaksi tai useampia ainakin näennäisesti identtistä riviä.

Tuntematon arvo (engl. *unknown*) on loogisen lausekkeen arvo, jota ei voida laskea.

Ulkoliitos (engl. *outer join*) on taulujen yhdistämisoperaatio, jossa liitoksen tulostauluun lisätään liitoksessa parittomiksi jääneet rivit puuttuvien tietojen osalta tyhjin arvoin täydennettynä. Ulkoliitos voi olla toispuoleinen tai kaksipuolinen.

Viiteavain (engl. *foreign key*) on kenttä tai kenttien yhdistelmä, jonka avulla voidaan osoittaa jokin tietokannan (saman tai jonkin toisen taulun) yksittäinen rivi käyttäen tämän perusavainta.

Viite-eheys (engl. *referential integrity*) on tietokannan eheysehto, joka on voimassa, kun kunkin viittaavan taulun riveillä viiteavainten arvot ovat viitatussa taulussa olevia perusavainten arvoja tai puuttuvia arvoja.

Viritys (engl. *performance tuning*) on tietokantaoperaatioiden suoritusajan parantamista tietokannan loogista tai fyysistä rakennetta muuttamalla.

Yksilöivä hakemisto eli erilaishakemisto (engl. *unique index*) on indeksi eli hakemisto, jossa kukin avainarvo voi esiintyä vain kerran.

Yleisrasite (engl. *overhead*) on johonkin toimintoon liittyvä välttämätön resurssien käyttö tai muu kustannus, joka on yleensä riippumaton toiminnon kestosta tai laajuudesta. Esimerkiksi SQL-kyselyn suorittamiseksi tulee tietokannanhallintajärjestelmän ensin jäsentää (engl. *parse*) suoritettavaksi annettu kysely, ja vasta jäsentämisen jälkeen kysely voidaan suorittaa.

4 Tietokantojen perusteita

Nykyään suuri osa tietokoneiden avulla tapahtuvasta toiminnasta hyödyntää tietokannoissa olevaa dataa välillisesti tai suoraan jonkin käyttöliittymän kautta. Muun muassa yritysten asiakasrekisterit, tilausjärjestelmät, varastohallintajärjestelmät ja WWW-sivujen kautta näkyvät tuote-esittelysivustot hyödyntävät tietokantoja, joihin käsiteltävät tiedot ja usein myös tietoihin kohdistuvat toimenpiteet on tallennettu. Hoffer mainitsee kirjassaan [7, s. 3–4], että tallennetuista tiedoista voidaan muodostaa yhteenvetoja, ja näin saada lisäarvoa organisaatiolle muun muassa kustannusten ja tietämyksen seurantaa varten sekä markkinoinnin avuksi. Luku perustuu pääosin lähteisiin [6, s. 4–10, 25–27, 199–200, 468–475 ja 546–551] sekä [7, s. 5–7, 24–26 ja 166–194].

4.1 Tietokanta, data ja tieto

Hoffer määrittelee kirjassaan [7, s. 5] **tietokannan** (engl. *database*) järjestetyksi kokoelmaksi toisiinsa liittyvää dataa (engl. *organized collection of logically related data*). **Data** (engl. *data*) on tallennettavissa olevia tunnettuja tosiasioita, tekstiä, grafiikkaa, kuvia, ääntä tai videoita (engl. *facts, text, graphics, images, sound and video segments*), joilla on merkitystä käyttäjien ympäristössä. Hofferin [7, s. 5] mukaan **tieto** (engl. *information*) on dataa, jonka käsittely lisää dataa käyttävän henkilön tietämystä. Hoffer mainitsee, että varsin usein dataa ja tietoa käytetään samassa merkityksessä.

Edellä esitetyssä määritelmässä **järjestetyllä** Hoffer [7, s. 5] tarkoittaa sitä, että data on rakenteeltaan helposti tallennettavaa, käsiteltävää ja käyttöön noudettavaa. **Toisiinsa liittyvä** data kuvaa Hofferin [7, s. 5] määritelmän mukaan jollekin käyttäjäryhmälle jonkin kiinnostavan vaikutusalan ominaisuuksia, ja tämän datan avulla käyttäjät voivat vastata kyseistä vaikutusalaa koskeviin kysymyksiin.

Edellä esitetty tietokannan määritelmä on varsin yleinen, joten vaikkapa tietyn päivän sanomalehtikin voidaan väljästi ajateltuna katsoa olevan eräänlainen tietokanta. Elmasri tarkentaa kirjassaan [6, s. 4] yleensä **termiä tietokanta** käytettävän kuitenkin vain silloin, kun vähintään seuraavat ehdot toteutuvat:

- Tietokanta kuvaa joitain todellisen maailman piirteitä. Näiden piirteiden muuttuessa myös tietokantaan tallennetut piirteiden kuvaukset ja ominaisuudet muuttuvat.
- Tietokanta on loogisesti yhtenäinen ja järjestetty kokoelma tallennettavissa olevia tietoja, joilla on jokin luontainen merkitys. Minkä tahansa satunnaisen tai mielivaltaisen tietojen kokoelman ei voida katsoa olevan tietokanta.
- Tietokanta suunnitellaan ja toteutetaan tiettyjä tavoitteita varten. Tietty käyttäjien joukko käyttää tietokannan dataa ja palveluja muodostaakseen uutta tietoa, joka tarvittaessa tallennetaan uudelleenkäytettäväksi joko samaan tietokantaan tai jonnekin muualle.

Elmasrin määritelmän [6, s. 4] mukaan tietokantaan liittyvät oleellisesti **tietolähteet**, joista tiedot ovat peräisin, **toiminnot ja palvelut**, joilla tietokantaan tallennettuja tietoja käsitellään, sekä **toimijat**, jotka tietoja käsittelevät ja jotka tiedoista ovat jollakin tavalla kiinnostuneita. **Tietolähteinä** voivat toimia vaikkapa toiset tietokannat, jotkin todellisen maailman ominaisuuksia mittaavat anturit tai WWW-sovelluksen avulla tietoja syöttävät ihmiset. Tietokannan tarjoamiin **palveluihin** voi kuulua erittäin monenlaisia toimintoja, kuten vaikkapa käteisen rahan nostaminen pankkiautomaatista tai lentolipun tilaaminen puhelinautomaatin avulla. **Toimijoita** ovat mm. tietokannan tietosisällöstä kiinnostuneet ohjelmistot ja ihmiset, kuten yrityksen tuotteista kiinnostuneen asiakkaan puhelinsoittoon tai sähköpostiviestiin vastaava asiakaspalvelija.

4.2 Tietokannan taulut, tietueet ja kentät sekä kuvaustiedot

Tietokannoissa kaikki tiedot on Hofferin [7, s. 166] mukaan tallennettu **taulujen** (engl. *table* tai *relation*) sisältämiin **tietueisiin** (engl. *record*), jotka koostuvat **kentistä** (engl. *field*). Kukin tietue muodostaa taulussa yhden **rivin** (engl. *row*), ja kukin tietueen kenttä on sijoitettu **sarakkeeksi** (engl. *column*).

Hofferin [7, s. 6–7] mukaan **kuvaustiedon** (engl. *metadata*) avulla kuvataan varsinaisen datan ominaisuudet ja tuotetaan asiayhteys (engl. *context*), johon data liittyy. Ominaisuuksiin kuuluvat muun muassa datan määritelmät (engl. *data definitions*), tietorakenteet (engl. *data structures*), sekä säännöt ja rajoitteet (engl. *rules and constraints*). Kuvaustiedon avulla tietokannan suunnittelijat ja käyttäjät ymmärtävät, mitä dataa on olemassa, mitä data tarkoittaa ja mitä eroja on toisiinsa nähden

samanlaisissa data-alkioissa. Kuvaustiedon hallinta on Hofferin mukaan vähintään yhtä tärkeää kuin varsinaisen datankin hallinta, sillä ilman selkeää merkitystä voi data olla hämmentävää, sitä voidaan tulkita väärin tai suorastaan käyttää virheellisesti.

Jokaisessa tietokannan taulussa tulee Hofferin [7, s. 167] mukaan olla kuvaustietona nimi sekä **perusavaimeksi** (engl. *primary key*) määritelty yhden tai useamman kentän yhdistelmä. Taulun nimen avulla taulu yksilöidään muihin tauluihin nähden. Taulun perusavaimen arvojen avulla rivi yksilöidään muihin taulun riveihin nähden. Jos taulun sisältämistä kentistä ei luonnollisesti löydy yksilöivää perusavainta, voidaan tauluun Hofferin [7, s. 183] mukaan lisätä yksilöintiä varten erityinen **tunnistekenttä** (engl. *identification*, `ID` tai *surrogate key*). Perusavaimen arvoihin voidaan viitata muista tauluista **viite-eheyksien** (engl. *referential integrity*) varmistamiseksi. Hofferin [7, s. 167] mukaan viittaavia kenttiä kutsutaan **viiteavaimiksi** (engl. *foreign key*).

Jos rivien yksilöintiä varten jouduttiin luomaan tauluun kenttä perusavainta varten, voi luotu tunnistekenttä ja siten samalla myös taulun perusavain olla Mullinsin [11, s. 125] mukaan **automaattisesti numeroituva**, joka voi tietokannanhallintajärjestelmästä riippuen olla vaikkapa `autonumber`-, `serial`- tai `counter`-tyyppinen. Yksilöintiä varten luodun tunnistekentän arvoista voidaan myös huolehtia sovelluksen lähdekoodin avulla tietojen tallennusvaiheessa. Tämä tapa on edelleen laajasti käytössä Korppi-järjestelmässä, mutta erityisesti automaattisina ajoina suoritettavissa tietojen tietokantaan latauksissa (engl. *data import*) on siirrytty automaattiseen numerointiin.

Kullakin tietueen kentällä on välttämättöminä kuvaustietoina nimi, tietotyyppi ja tietotyypistä suoraan johtuva tai tietokannan suunnittelijan määrittämä suurin mahdollinen koko. Lisäksi kentillä voi olla Hofferin [7, s. 171] mukaan muita rajoitteita (engl. *constraint*), kuten oletusarvo ja tieto siitä, että kentän arvoa ei saa jättää tyhjäksi tietoja lisättäessä tai päivitettäessä (SQL-kielessä `NULL` tai `NOT NULL`, ks. luku 5.4). Tietueiden kenttien kuvaustietoihin voidaan määritellä lisäksi tarkistusrajoite (engl. *check constraint*) sitä varten, että tauluun päivitettävän tai lisättävän rivin kenttien arvot täyttävät tietyt ennalta asetetut ehdot.

Kunkin tietokannan sisältämä data ja kuvaustiedot on tallennettu Hofferin [7, s. 261] mukaan **skeemaan** (engl. *schema*). Tietokantojen skeemat sekä määritystiedot sisältävä taulukokoelma eli ns. tietoskeema (engl. *information schema*) on tallennettu tietokannanhallintajärjestelmän tarjoamaan **systeemitaulustoon** (engl. *catalog*).

Tutkielmassa käytetään taulujen, relaatioiden, rivien ja kenttien määritelminä teknisiä tietokantayhteyksissä käytettyjä termejä, eikä niiden formaaleja matemaattisia määritelmiä. Elmasrin [6, s. 199–200] mukaan tietokannan taulu ja formaalissa mielessä relaatio (engl. *relation*) eroavat hieman toisistaan, sillä tietokannoissa taulut ovat tarkkaanottaen joukkoja (engl. *set*), joissa sallitaan toistuvat rivit (engl. *duplicate rows*). Formaaleissa relaatioissa datariveillä (engl. *tuple*) ei ole keskinäistä järjestystä, koska relaatioissa toimitaan abstraktilla tai loogisella tasolla, jossa ei ole mielekästä asettaa mitään järjestystä etusijalle. Samaten relaatioiden attribuuttien (engl. *attributes*) järjestys on merkityksetön, kunhan säilytetään vastaavuus kentän ja sen arvon välillä. Sen sijaan tietokannan taulujen rivit ja rivien kentät ovat tallennusmedialle tallennettuina aina jossakin järjestyksessä.

4.3 Tietokannanhallintajärjestelmä

Tietokannan rakennetta ja dataa voidaan ylläpitää käsin tai tietokoneiden avulla. Koneellista tietojen ylläpitoa varten voidaan rakentaa erityisiä sovelluksia, tai tietojen tallentaminen ja käsittely voidaan antaa jonkin tietokannanhallintajärjestelmän huoleksi. Jos tietojen hyödyntämistä varten toteutetaan erityinen sovellus, menetetään Elmasrin [6, s. 9] mukaan yksi oleellinen tietokannanhallintajärjestelmien etu. Tiedot nimittäin tallennetaan sovelluksen tietorakenteiden ehdoilla, ja jos tietorakenteisiin halutaan tehdä muutoksia, joudutaan muokkaamaan kaikkia niitä sovelluksia, jotka kyseisiä tietoja käyttävät.

Tietokannanhallintajärjestelmillä (engl. *database management system, DBMS*) hyödynnetään tietokannan tietoja lukujen 4.1 ja 4.2 mukaisesti. Järjestelmällä voidaan tietokannan rakenne ja ominaisuudet suunnitella ja toteuttaa sekä hallita ja hyödyntää muiden sovellusten avulla tietokantaan tallennettuja tietoja. Hofferin [7, s. 24 ja 26] mukaan tietokannanhallintajärjestelmän avulla kullekin tietokannan käyttäjälle ja tietokantaa hyödyntävälle sovellukselle voidaan antaa vain tarvittavat ja riittävät oikeudet tietokannan tietojen raportointiin ja päivittämiseen.

Vuonna 1970 Edgar F. Codd esitteli artikkelissaan [3] ensimmäisen kerran relaatiotietokantateknologioiden perusteita. IBM ryhtyi kehittämään System R -nimistä järjestelmää esitelläkseen sitä, kuinka relaatiomalli voitaisiin toteuttaa tietokannanhallintajärjestelmissä. Projektissa käytettiin SEQUEL-nimistä kieltä (engl. *Structured English Query Language*), jonka IBM oli kehittänyt hieman aiemmin. Hofferin [7,

s. 258] mukaan SEQUEL nimettiin SQL:ksi projektin kuluessa 1970-luvun jälkipuoliskolla. SQL-kieli esitellään luvussa 5.

Koska IBM:n kehittämä System R otettiin Hofferin [7, s. 258] mukaan asiakaskunnassa vastaan tyytyväisyydellä, alkoivat muutkin toimijat kehittää SQL-kieltä hyödyntäviä relaatiomalliin perustuvia tuotteita. Nykyisin Oracle-nimellä tunnettu Relational Software esitteli kehittämänsä tietokannanhallintajärjestelmän vuonna 1979. Ensimmäinen IBM:n kaupallinen relaatiotietokannanhallintajärjestelmä nimeltään SQL/DS esiteltiin vuonna 1981. Kaksi vuotta myöhemmin esiteltiin kyseisen järjestelmän MVS-käyttöjärjestelmälle tarkoitettu versio, jolle annettiin nimeksi DB2. 1970–1980-lukujen vaihteessa moni muukin tuote näki päivänvalon, kuten Kalifornian Berkeleyn yliopiston vapaassa jakelussa oleva Ingres-niminen tietokannanhallintajärjestelmä. PostgreSQL:n tekijöiden [28] mukaan Postgres- ja PostgreSQL-tietokannanhallintajärjestelmät on toteutettu Ingresin pohjalta (”Post Ingres”).

Kaupallisesti ja vapaasti on nykyään saatavilla useita erilaisia ja eri tarpeisiin suunnattuja tietokannanhallintajärjestelmiä. Tutkielmassa tarkastellaan esimerkkinä PostgreSQL-tietokannanhallintajärjestelmää ja sen tarjoamia ominaisuuksia tietokantaa hyödyntävän Korppi-järjestelmän näkökulmasta.

4.4 Operaatiot ja datan abstraktius

Elmasrin [6, s. 9] mukaan tietokannoissa varsinaiset tiedot sijaitsevat erillään tietorakenteista, eikä tarpeiden muuttuessa välttämättä tarvita suuria muutoksia tietoja hyödyntäviin sovelluksiin (engl. *program-data independence*). Elmasrin mukaan yleensä riittää muokata tietokannan skeemaa (ks. luku 4.2).

Tietokannoissa voi olla varsinaisen tiedon lisäksi tallennettuina myös **operaatioita**, joilla tallennettuja tietoja käsitellään. Elmasrin [6, s. 9–10] mukaan näin on erityisesti oliotietokannoissa (engl. *object-oriented databases*) ja oliorelaatiotietokannoissa (engl. *object relational databases*). Tiedot ja operaatiot on eriytetty toisistaan (engl. *program-operation independence*), ja operaatioista esitetään vain rajapinnat (engl. *interfaces*), joiden avulla tietoja voidaan käsitellä. Näin operaatioiden sisäisellä toimintatavalla ei ole Elmasrin mukaan käyttäjälle merkitystä.

Mainituilla riippumattomuuksilla saavutetaan Elmasrin [6, s. 10] mukaan **datan abstraktius** (engl. *data abstraction*). Käsitteelliset esitystavat ja tietomallit (engl. *data*

model) piilottavat tietojen tallennuksen yksityiskohdat, jotka eivät käyttäjiä kiinnosta.

4.5 Tietokannan kehitysvaiheet

Tietokannan **määrittely** (engl. *requirements collection and analysis*) lähtee sitä hyödyntävien käyttäjien ja sovellusten tarpeiden määrittelystä. Ainakin seuraaviin Elmasrin [6, s. 535] ja Hofferin [7, s. 41] teoksiin pohjautuviin kysymyksiin tulisi miettiä vastaus ennen kuin minkäänlaisia taulurakenteen hahmotelmia toteutetaan:

- Mitä tarpeita varten tietoja halutaan tallentaa?
- Millaisia raportteja ja tulosteita tallennetuista tiedoista halutaan tuottaa?
- Mitä vaatimuksia tallennettaville tiedoille tulee asettaa?
- Millaisia tietoja ja missä muodossa tietoja halutaan tallentaa, eli mitkä ovat tietojen tyypit, rajoitteet ja yhteydet muihin tietoihin?
- Millaisia tietorakenteita halutaan käyttää tietojen tallentamiseen?
- Mitä tietokannanhallintajärjestelmää halutaan käyttää?
- Voidaanko suoraan tai rinnalla hyödyntää jotain olemassa olevaa tietokantaa, jota laajentamalla ja kehittämällä saadaan aikaan halutut uudet toiminnot?

Elmasrin [6, s. 536] mukaan tietokannan määrittelyä seuraa **käsitteellisen mallintamisen vaihe** (engl. *conceptual database design*). Tällöin tallennettavista tiedoista etsitään Elmasrin määritelmän [6, s. 25] mukaan kohteet (engl. *entity*), näiden ominaisuudet (engl. *attributes*) sekä kohteiden väliset riippuvuudet ja suhteet (engl. *relationships*). Tietokohteet ominaisuuksineen ja suhteineen mallinnetaan Hofferin [7, s. 76 ja 523] mukaan nykyään useimmiten relaatiomallin (engl. *relational data model*) tai oliomallin (engl. *object data model*) avulla. Tutkielmassa keskitytään pelkästään relaatiomalliin.

Käsitteellisen mallintamisen vaiheessa **tietomallin kohteet** kuvataan jollakin **tiedonmäärityskielellä** (engl. *data definition language*, `DDL`) esimerkiksi ER-malliksi (engl. *Entity-Relationship model*). Se ei ota vielä kantaa siihen, miten ja millaiseen tietokannanhallintajärjestelmään tietoja tullaan tallentamaan. Elmasrin [6, s. 27] mukaan malli piilottaa tietojen fyysisen tallennuksen yksityiskohdat, joten mallinta-

misessa keskitytään kohteisiin, niiden ominaisuuksiin ja tietotyyppeihin, kohteiden välisiin suhteisiin, käyttäjien operaatioihin sekä rajoitteisiin eri kohteiden välillä.

Käsitteellisen mallintamisen vaihetta seuraa Elmasrin [6, s. 546] mukaan **tietokannanhallintajärjestelmän valinta** (engl. *choice of a DBMS*). Valinnan tueksi tulee kartoittaa ainakin seuraavat **kustannukset**:

- tietokannanhallintajärjestelmän, siihen liittyvien kehitysympäristöjen ja sitä käyttävien sovellusten hankintakulut,
- tietokannanhallintajärjestelmän ylläpito- ja päivityskulut,
- olemassa olevan laitteiston laajentamiskulut tai uuden laitteiston hankintakulut,
- tietokannan luonti- ja konvertointikulut,
- henkilöstökulut,
- koulutuskulut sekä
- tietokannanhallintajärjestelmän jatkuvaan toimintaan liittyvät käyttö- ja lisenssikulut.

Lisäksi Elmasrin [6, s. 548] mukaan tietokannanhallintajärjestelmän valintaan vaikuttavat seuraavat **taloudelliset ja organisatoriset tekijät**:

- organisaatiolaajuinen tiettyjen toimintatapojen noudattaminen,
- henkilöstön perehtyneisyys johonkin tietokannanhallintajärjestelmään,
- sovelluksen toimittajan palveluiden saatavuus sekä
- tietokannan siirrettävyys eri alustojen välillä (engl. *portability among platforms*).

Elmasrin [6, s. 549] mukaan tietokannanhallintajärjestelmän valintaa seuraa tietokannan **loogisen mallintamisen vaihe** (engl. *logical database design*). Tällöin Mullinsin [11, s. 106–108] mukaan loogiseen tietomalliin (engl. *logical data model*) merkitään kaikki kohteet, näiden kaikki ominaisuudet sekä kohteiden väliset suhteet. Kunkin kohteen tulee olla normalisoitu halutulle tasolle [11, s. 115] (ks. luku 4.6). Kunkin ominaisuuden tietotyyppi tulee olla määritelty. Lisäksi kunkin kohteen perusavain sekä viite-eheydet tulee olla selvitettyinä. Viite-eheyksiin liittyvät tarkemmat tiedot, kuten lukumäärärajoitteet (engl. *cardinality constraint*) ja viite-eheyden nimi, tulee olla merkittynä loogiseen tietomalliin. Loogisen tietomallin tulee olla täydellinen dokumentti, jonka avulla fyysinen tietokanta voidaan seuraavissa vaiheissa toteuttaa.

Elmasrin [6, s. 549–550] mukaan loogista mallintamisvaihetta seuraa tietokannan **fyysinen mallintaminen** (engl. *physical database design*). Fyysisen mallintamisen aikana rajoitetaan toteutusvaihtoehdot valitun tietokannanhallintajärjestelmän tar-

joamiin ominaisuuksiin ja tietorakenteisiin, kuten erityyppisiin indeksointitekniikoihin ja yhteenkuuluvien tietueiden liittämiseen toisiinsa osoittimien avulla (engl. *linking related records via pointers*). Täten vaiheen aikana aikaansaatua fyysistä tietomallia (engl. *physical data model*) käytetään muunnettaessa looginen tietomalli fyysiseksi tietokannaksi.

Elmasrin [6, s. 550] mukaan edellä kuvattuja mallintamisvaiheita seuraa tietokannan **toteutusvaihe** (engl. *database system implementation*). Tällöin edellisissä vaiheissa suunnitellut tietorakenteet luodaan valittuun tietokannanhallintajärjestelmään tietokannaksi kannankuvauskielellä, yleensä luvussa 5 esiteltävällä SQL-kielellä. Lisäksi mahdollisiin aiemmin käytössä olleisiin tietokantoihin tallennetut tiedot tallennetaan uusiin tietorakenteisiin joko suoraan tai muuntaen entisestä tallennusmuodosta.

Toimintavaiheen (engl. *operational phase*) aikana Elmasrin [6, s. 550] mukaan seurataan tietokannan ja tietokannanhallintajärjestelmän toimintaa muun muassa tietokantaan tallennettujen käyttötilastojen avulla. Kyselyitä ja tietokantaan tallennettuja operaatioita joudutaan ehkä uudelleenkirjoittamaan, jotta ne suoriutuvat tehokkaammin. Hofferin [7, s. 42] mukaan toimintavaiheen aikana korjataan tietokannasta ja tietokantaa käyttävistä sovelluksista löydetyt virheet sekä muokataan tietorakenteita ja dataa vastaamaan todellisen maailman muutoksia.

Elmasrin [6, s. 551] mukaan **tietokannan tehostaminen** (engl. *database tuning*) jatkuu niin kauan kuin suorituskykyongelmia ilmenee, tietokantaan kohdistuvat vaatimukset muuttuvat tai tietokantaa yleensäkään hyödynnetään.

4.6 Normalisoinnin tavoitteet

Elmasrin [6, s. 468] mukaan tietokannan kaikkien taulujen tulisi olla helposti ymmärrettäviä ja sisältää tietoja vain yhdestä **kohteesta** (engl. *entity*). Tällä vältetään saman tiedon tallentaminen useampaan eri kenttään tai tauluun, joista osa jäisi päivittämättä tietojen päivitysten yhteydessä. Celko [2, s. 21] kiteyttää tämän ytimekkäästi vertaukseen miehestä, jolla on kaksi kelloa. Mies ei voi koskaan tietää, mikä on täsmällinen aika.

Hoffer [7, s. 174] mainitsee, että samaa tietoa ei saa tallentaa myöskään saman taulun useammalle riville. Tietokannan taulujen rakenne muodostuu näin Elmasrin [6, s. 472] mukaan selkeäksi ja kuvaavaksi, eikä **ongelmia** (engl. *anomalies*) ilmene tietojen lisäysten (engl. *insert*), päivitysten (engl. *update*) tai poistojen (engl. *delete*)

yhteydessä. Edellä mainitut tavoitteet saavutetaan normalisoinnin avulla.

Normalisoinniksi kutsutaan Elmasrin [6, s. 484] mukaan kaikkia niitä toimenpiteitä, joilla tietokannan rakenteesta poistetaan samojen tietojen päällekkäisyydet (engl. *redundancy*) ja toistuvuudet, järjestellään tietojen tallennustapaa tehokkaammaksi sekä vähennetään tietoihin kohdistuvien operaatioiden aiheuttamia mahdollisia ongelmia ja poikkeamia (engl. *anomaly*). Tietojen normalisointi saattaa lisätä tietojen yhtenäisyyttä (engl. *consistency*), ja se saattaa myös helpottaa tietokantarakenteen laajentamista jatkossa. Elmasrin [6, s. 484] mukaan ei yleensä riitä tarkastella normalisointiprosessin onnistumista pelkästään yksittäisten taulujen tasolla, vaan myös kokonaisuutena.

Edgar F. Codd esitteli vuonna 1972 raportissaan [4] kolme tietokannan normaalimuotoa (*1NF*, *2NF* ja *3NF*). Niiden testeillä ja korjaustoimenpiteillä pyritään tietokannan rakenne muokkaamaan sellaiseksi, että edellä mainittuja epäkohtia ei ilmenisi. Ollakseen korkeamman normaalimuodon mukainen taulu, on taulun Hofferin [7, s. 192–193] mukaan toteutettava myös kaikkien alempien normaalimuotojen ehdot.

Hofferin [7, s. 190–191] mukaan kenttä on **funktionaalisesti riippuva** (engl. *functionally dependent*), jos kyseisen kentän arvo on yksikäsitteisesti pääteltävissä saman taulun jonkin toisen kentän tai useamman kentän yhdistelmän arvojen perusteella. Esimerkiksi henkilön nimi ja kotiosoite määräytyvät henkilötunnuksen perusteella, mutta ei toisinpäin. Täten henkilön nimi ja kotiosoite ovat funktionaalisesti riippuvia henkilötunnuksesta. Koska samannimisiä henkilöitä voi olla useita, nimestä ei suoraan voida päätellä henkilötunnusta, eikä siten henkilötunnus ole funktionaalisesti riippuva henkilön nimestä.

Hofferin [7, s. 191] mukaan **avainehdokas** (engl. *candidate key*) on taulun yksittäinen kenttä tai minimaalinen kenttien joukko, jonka eri riveillä olevat arvot tai arvoyhdistelmät eroavat toisistaan. Lisäksi yhtäkään avainehdokkaan kenttää ei voida poistaa niin, että mainittu yksikäsitteisyyssääntö vielä pätisi. Avainehdokasta voidaan siten käyttää yksikäsitteisesti rivien tunnisteena, ja taulun perusavaimeksi valitaankin yksi tällaisista avainehdokkaista.

Normalisointiprosessin onnistunut läpivienti vaatii Elmasrin [6, s. 484] mukaan ehdottomasti sen, että kun taulu jaetaan pienemmiksi osatauluiksi, mitään tietoa ei saa prosessissa kadota (engl. *lossless decomposition*). Tämän tulee päteä myös toisinpäin, eli kun alkuperäinen taulu koostetaan osatauluista, ei ylimääräisiä tietoja saa ilmestyä (engl. *nonadditive join property*).

Vaatimuksena normalisoinnille asetetaan Elmasrin [6, s. 484] mukaan yleensä myös riippumattomuuden säilyminen (engl. *dependency preservation property*). Sen mukaan jokainen alkuperäisen taulun funktionaalinen riippumattomuus esiintyy joissakin normalisoinnin seurauksena syntyvissä uusissa osatauluissa. Tämä ei ole Elmasrin mukaan yhtä ehdoton vaatimus kuin edellä esitetyt vaatimukset tietojen katoamattomuudelle ja ilmestymättömyydelle.

4.7 Ensimmäinen normaalimuoto

Taulu on Hofferin [7, s. 192] mukaan **ensimmäisen normaalimuodon** (engl. *First Normal Form*, `1NF`) mukainen, mikäli kaikissa taulun kentissä saa esiintyä vain **alkeisarvoja** (engl. *atomic values*). Jos arvojen kokoelmia tai listoja esiintyy, jaetaan taulu Elmasrin ohjeen [6, s. 486] perusteella useampaan osaan, sekä tallennetaan kokoelmissa ja listoissa olevat arvot uuteen tauluun yksittäisinä alkeisarvoina omille riveilleen. Kukin rivi viittaa tällöin alkuperäisen taulun perusavaimeen sekä mahdollisesti muiden taulujen perusavaimiin viite-eheyksien avulla.

Esimerkkinä taulukossa 4.1 esitetään henkilöiden perustiedot tallentava `henkilö`-taulu, joka ei ole `omaisuus`-kentän listojen vuoksi ensimmäisen normaalimuodon mukainen. Taulun perusavain muodostuu asteriskilla $\star$ merkitystä `henkilöID`-kentästä.

henkilöID *	**sukunimi**	**etunimi**	**valtiokoodi**	**valtio**	**omaisuus**
1	Meikäläinen	Matti	fi	Suomi	tietokone, asunto
3	Kolmonen	Jaakko	fi	Suomi	2 wokkipannua
5	Palme	Olof	sv	Ruotsi	NULL
6	Doe	John	NULL	NULL	NULL
7	Presley	Elvis	us	Yhdysvallat	4 kitaraa, 3 Cadillacia, asunto Gracelandissa

Taulukko 4.1: `Henkilö`-taulu ei ole ensimmäisen normaalimuodon mukainen `omaisuus`-kentän listojen vuoksi.

Jotta taulukon 4.1 taulu saataisiin ensimmäisen normaalimuodon mukaiseksi, tulee taulun `omaisuus`-kentän listat jakaa alkeisarvoiksi uuteen tauluun. Uusi taulurakenne esitetään taulukoissa 4.2 ja 4.3.

Taulukon 4.3 mukaista `henkilön_omaisuus`-taulua kannattaa vielä muokata siirtämällä hyödykkeen tiedot omaksi `hyödyke`-taulukseen, jolloin taulukon 4.3

henkilöID *	**sukunimi**	**etunimi**	**valtiokoodi**	**valtio**
1	Meikäläinen	Matti	fi	Suomi
3	Kolmonen	Jaakko	fi	Suomi
5	Palme	Olof	sv	Ruotsi
6	Doe	John	NULL	NULL
7	Presley	Elvis	us	Yhdysvallat

Taulukko 4.2: Poistamalla taulukon 4.1 taulusta `omaisuus`-kenttä saadaan ensimmäisen normaalimuodon mukainen `henkilö`-taulu.

henkilöID *	**hyödyke** *	**lukumäärä**
1	tietokone	1
1	asunto	1
3	wokkipannu	2
7	kitara	4
7	Cadillac	3
7	asunto Gracelandissa	1

Taulukko 4.3: Ensimmäisen normaalimuodon mukainen taulu `henkilön_omaisuus`.

mukaiseen tauluun tulisi `hyödyke`-kentän sijasta viite-eheys kyseisen `hyödyke`-taulun `hyödykeID`-kenttään. Näin vaikkapa `asunto`-tyyppisten hyödykkeiden toistuvat arvot saadaan tallennettua kertaalleen omaan tauluunsa.

4.8 Toinen normaalimuoto

Taulu on Hofferin [7, s. 193] mukaan **toisessa normaalimuodossa** (engl. *Second Normal Form*, `2NF`), mikäli jokainen taulun muu kuin jonkin avainehdokkaan kenttä (engl. *nonkey attribute*) on funktionaalisesti riippuva koko perusavaimesta. Ehto on triviaalisti totta, mikäli perusavain muodostuu ainoastaan yhdestä kentästä tai taulussa ei ole muita kuin perusavaimeen kuuluvia kenttiä. Jos mainittu toisen normaalimuodon ehto ei toteudu, taulu tulee jakaa osiin siten, että ehto pitää paikkansa jokaiselle jaon seurauksena muodostetulle taululle.

Koska taulukon 4.2 esimerkkitaulun perusavain muodostuu vain yhdestä kentästä, on kyseinen taulu toisen normaalimuodon mukainen. Myös omaisuuden tallentamiseen käytetty taulukon 4.3 taulu on tällaisenaan toisen normaalimuo-

don mukainen. Jos kuitenkin kyseiseen `henkilön_omaisuus`-tauluun lisättäisiin vaikkapa `valmistusmaa`-kenttä kuvaamaan hyödykkeen valmistusmaata, ei taulu enää olisikaan toisessa normaalimuodossa. Tällöin nimittäin hyödykkeen valmistusmaa riippuisi ainoastaan hyödykelajista, eikä enää nimenomaan tietyn henkilön omistamista yksittäisistä hyödykkeistä. Valmistusmaa riippuisi siis ainoastaan osasta perusavaimen arvoa, ei koko perusavaimen arvosta.

4.9 Kolmas normaalimuoto

Hofferin [7, s. 193] mukaan **kolmannen normaalimuodon** (engl. *Third Normal Form*, `3NF`) mukaisessa taulussa ei saa olla keskinäisiä funktionaalisia riippuvuuksia niiden kenttien kesken, jotka eivät kuulu perusavaimeen (engl. *transitive dependency*). Kaikkien perusavaimeen kuulumattomien kenttien täytyy siis olla itsenäisiä ja riippumattomia muista perusavaimeen kuulumattomista kentistä.

Jos tällaisia keskinäisiä riippuvuuksia on, jaetaan taulu Hofferin [7, s. 193–194] mukaan osiin yleensä siten, että riippuvuussuhteen määräävä kenttä (engl. *determinant*) viedään perusavaimeksi uuteen tauluun, tai uuden taulun perusavaimeksi luodaan erillinen tunnistekenttä. Tähän tauluun siirretään myös riippuvuussuhteen perusteella määräytyvät kentät. Alkuperäisessä taulussa määräävä kenttä muunnetaan viiteavaimeksi, joka siis viittaa uuden taulun perusavaimeen.

Kolmas normaalimuoto on Elmasrin [6, s. 493] mukaan toisen normaalimuodon erikoistapaus, joten aina **riittää tarkistaa taulukohtaisesti vain ensimmäisen ja kolmannen normaalimuodon paikkansapitävyys** sekä tehdä tarvittavat korjaukset.

Luvun 4.7 taulukossa 4.2 esimerkkitaulun `valtio`-kentän arvo on funktionaalisesti riippuva valtiokoodista. Valtiokoodit ja valtioiden nimet tulee siis siirtää uuteen `valtio`-tauluun. Näin `henkilö`-taulusta poistetaan `valtio`-kenttä ja vaihdetaan `valtiokoodi`-kenttä viittaamaan `valtio`-taulun perusavaimeen. Nämä muokkaukset esitetään taulukoissa 4.4 ja 4.5. Perusavain on kummassakin taulussa merkitty asteriskilla `*`. Perusavainten muodon ja yhtenäisyyden vuoksi `valtio`-taulun perusavain voisi koostua `valtiokoodi`-kentän sijasta myös uudesta tauluun lisättävästä `valtioID`-tunnistekentästä.

Taulukon 4.5 `henkilö`-taulun `valtiokoodi`-kentässä on viite-eheys taulukon 4.4 `valtio`-taulun `valtiokoodi`-kenttään.

valtiokoodi *	valtio
fi	Suomi
sv	Ruotsi
us	Yhdysvallat
ru	Venäjä

Taulukko 4.4: Kolmannen normaalimuodon mukainen `valtio`-taulu.

henkilöID *	sukunimi	etunimi	valtiokoodi
1	Meikäläinen	Matti	fi
3	Kolmonen	Jaakko	fi
5	Palme	Olof	sv
6	Doe	John	NULL
7	Presley	Elvis	us

Taulukko 4.5: Kolmannen normaalimuodon mukainen `henkilö`-taulu.

4.10 Muut normaalimuodot

Hofferin [7, s. 589] ja Mullinsin [11, s. 114] mukaan **kolmannen normaalimuodon katsotaan olevan riittävä taso** useimmissa käytännön tietokannoissa. Kolmas normaalimuoto ei kuitenkaan Celkon [2, s. 26] mukaan takaa sitä, että tauluissa ei enää olisi jäljellä mitään luvussa 4.9 mainituista kolmannen normaalimuodon estävistä epäkohdista. Tästä syystä Codd esitteli yhdessä Boycen kanssa vuonna 1974 tiukemman määritelmän kolmannesta normaalimuodosta, joka tunnetaan Boycen-Coddin normaalimuotona (engl. *Boyce-Codd Normal Form*, BCNF). Näitä seurasi Özsun [35, s. 29] mukaan vielä neljännen ja viidennen normaalimuodon määritelmien esittely 1970-luvun loppupuolella.

Boycen-Coddin normaalimuodossa kolmannen normaalimuodon sääntöä on Hofferin [7, s. 590] mukaan tiukennettu siten, että riippuvuussuhteen määräävän kentän tai kenttien joukon tulee olla avainehdokas. Tällöin Celkon [2, s. 27] mukaan poistuvat kaikki keskinäiset funktionaaliset riippuvuudet, jonka seurauksena taasen poistuvat kaikki tärkeimmät poikkeavuudet (engl. *anomalies*) tietojen päivitysoperaatioissa.

Näitä myöhemmin esiteltyjä normaalimuotoja ei tarkastella tutkielmassa. Alan kirjallisuudesta löytyy runsaasti pohdintoja aiheesta, kuten vaikkapa Hofferin kirjoittamana kirjasta [7, Appendix B].

5 SQL-kieli

SQL-kielen (Structured Query Language) avulla voidaan määritellä relaatiotietokannan rakenne, tallentaa käsiteltävät tiedot tauluihin sekä muodostaa kyselyitä tietokantaan tallennettuihin rakenteisiin ja dataan. Luku keskittyy pääasiassa kannankäsittelykieleen ja erityisesti SQL-kyselyihin tutkielman tavoitteiden mukaisesti. Luku perustuu pääosin lähteisiin [2, s. 174–176], [6, s. 155–156] sekä [7, s. 209–212, 258–276 ja 312–315].

5.1 Standardin SQL-kielen historiaa ja etuja

Ashenfelterin [1] mukaan 1980-luvun alkupuoliskolla muutama valmistaja näki relaatiotietokantojen hyödyt, joten ne julkistivat omia SEQUELin periaatteita noudattavia kyselykieliä sisältäviä tietokannanhallintajärjestelmiään. Jotta eri tietokannanhallintajärjestelmien valmistajien toteuttamat SQL-murteet saataisiin yhtenäisemmiksi, kansainväliset ANSI- ja ISO-organisaatiot määrittelivät Hofferin [7, s. 258] mukaan SQL-relaatiokyselykielen standardin vuonna 1986. Ensimmäinen versio standardista tunnetaan nimellä SQL-86. Tämän jälkeen Ashenfelterin [1] mukaan päivänvalon ovat nähneet standardit SQL-89, SQL-92 ja SQL-99, joista viimeisin tunnetaan myös nimellä SQL3.

Standardilla SQL-kielellä on Hofferin [7, s. 260] mukaan monia etuja. Kun käyttäjät ja ohjelmoijat osaavat SQL-kielen perusteet, kohtuullisen pienellä kouluttautumisella ja siten vähillä kustannuksilla pystytään hyödyntämään mitä tahansa SQL-kieltä tukevaa relaatiotietokannanhallintajärjestelmää.

IT-ammattilaisen **tuottavuus** paranee, koska hän voi keskittyä ongelmien ratkomiseen, eikä erilaisten työkalujen ja ympäristöjen opetteluun. Organisaatioihin voidaan hankkia ammattilaisten avuksi **työkaluohjelmistoja**, jotka toimivat standardin SQL:n päällä ilman, että on pelkoa investointien menemisestä hukkaan pitkälläkään aikavälillä. Olemassaolevien ohjelmien muokkaaminen onnistuu nopeammin ja vähemmällä tutustumisella, kun tietokannan käyttämiseen tarvittava kieli on jo tuttu.

Ohjelmistot ovat **alustariippumattomia** ainakin tietojen tallennuksen osalta, kun

käytettävissä ovat standardit rajapinnat ja työkalut tietokantojen käsittelyyn. Ohjelmistoja voidaan siten siirtää laitteelta toiselle hyvin helposti. Eri tietojärjestelmien välinen kommunikointikin onnistuu helpommin yhteisellä datankäsittelykielellä.

Riippuvuus yhdestä tietokannanhallintajärjestelmän **toimittajasta vähenee**, kun tarjolla on useita kilpailevia järjestelmiä. Tästä johtuen hinnat saattavat alentua ja palvelu parantua. Lisäksi koulutustarpeiden täyttämiseen on olemassa useita konsultointi- ja kouluttajaorganisaatioita sekä monenlaista kirjallisuutta, koska aina voidaan opettaa saman standardin mukaista SQL-kieltä.

5.2 SQL-kielen komennot

Hofferin [7, s. 262–275 ja 312–315] mukaan SQL-kieli koostuu tiedonmäärityskielestä, jonka avulla määritellään tietokannan tietorakenteet, kannankäsittelykielestä, jonka avulla käsitellään tauluissa olevaa dataa, ja kannanvalvontakielestä, jolla määritellään tietokantaoikeudet, operaatiot ja liipaisimet.

Hofferin [7, s. 262–273] mukaan SQL-kielen **tiedonmäärityskielellä** (engl. *data definition language*, DDL) luodaan ja poistetaan tauluja ja muita objekteja sekä muokataan taulujen rakenteita. `CREATE TABLE` -komennolla luodaan taulu. Vastaavasti `DROP TABLE` -komennolla taulu voidaan tuhota. `ALTER TABLE` -komennolla muokataan taulun rakennetta lisäämällä tai poistamalla kenttiä, rajoitteita tai viiteeheyksiä.

`CREATE INDEX` -komennolla luodaan tauluun hakuja nopeuttava indeksi, ja vastaavasti `DROP INDEX` -komennolla voidaan indeksejä poistaa. Indeksejä kuvataan enemmän luvussa 5.6. Mainitut indeksienkäsittelykomennot eivät ole mukana standardissa SQL-92, koska Hofferin [7, s. 276] mukaan indeksoinnilla pyritään tehostamaan tietojen hakua, ei määrittelemään tietokannan objektien rakenteita. ISO-standardit yleisesti eivät Hofferin [7, s. 276] mukaan muutenkaan ota kantaa suorituskykyongelmiin. Useimmat tietokannanhallintajärjestelmät kuitenkin tukevat indeksointia `CREATE INDEX` -komennon avulla.

`CREATE VIEW` -komennolla luodaan SQL-näkymä jonkin `SELECT`-lauseen avulla, ja `DROP VIEW` -komennolla poistetaan SQL-näkymä. Mullinsin [11, s. 154] mukaan näkymien avulla voidaan piilottaa monimutkaisten hakujen yksityiskohtia. Näkymillä voidaan myös turvata tietokannan tietoja taulujen rivien ja sarakkeiden asiattomalta käytöltä, kun taulujen tietojen tarkastelemiseksi annetaan vain välttämättömät oikeudet, ja näkymien avulla tietojen tarkasteluun annetaan laajem-

mat oikeudet. Myös taulujen ja kenttien uudelleennimeäminen onnistuu näkymien avulla.

Hofferin [7, s. 273–275] mukaan SQL-kielen **kannankäsittelykieleen** (engl. *data manipulation language*, DML) kuuluvilla `INSERT`-, `UPDATE`-, `DELETE`- ja `SELECT`-komennoilla lisätään, muutetaan, poistetaan ja haetaan dataa taulujen riveiltä. `SELECT`-lauseen rakenne esitellään tarkemmin luvussa 5.3.

Tietokannan oikeuksien sekä operaatioiden ja tapahtumien hallintaan on Hofferin [7, s. 312–315] mukaan olemassa SQL-kielen **kannanvalvontakielen** (engl. *data control language*, DCL) komennot. Näihin kuuluvilla komennoilla `GRANT` ja `REVOKE` annetaan ja poistetaan tietokantaoikeuksia. Komennoilla `CREATE FUNCTION`, `CREATE PROCEDURE` ja `CREATE TRIGGER` tietokantaan luodaan käyttäjän määrittelemiä operaatioita (engl. *function* ja *procedure*) ja liipaisimia (engl. *trigger*). Vastaavilla `DROP`-komennoilla voidaan luodut operaatiot ja liipaisimet poistaa. Hofferin [7, s. 312] mukaan käyttäjän määrittelemät operaatiot ja liipaisimet tulivat mukaan vasta standardiin SQL-99.

5.3 SQL-kyselyn rakenne ja suoritus

SQL-kielen mukainen kyselylause koostuu Celkon [2, s. 174] mukaan alla mainitussa järjestyksessä seuraavista lausekkeista:

`SELECT` -lausekkeessa luetellaan kentät, laskutoimitukset ja vakioarvot, jotka palautetaan kyselylauseen tuloksena.

`FROM` -lausekkeen avulla luetellaan taulut tai näkymät, joista tietoja haetaan tai joiden kentissä olevia tietoja vertaillaan.

`WHERE` -lausekkeeseen on koottu taulujen tai näkymien väliset viiteeheysehdot tai vakioarvoihin verrattavat ehdot.

`GROUP BY` -lausekkeen avulla annetaan hakutuloksen ryhmittelyehto.

`HAVING` -lausekkeella kerrotaan ryhmittelyssä käytettävä suodatusehto.

`ORDER BY` -lausekkeessa luetellaan kyselylauseen tuloksen järjestysehto.

Celkon [2, s. 174] mukaan ainoastaan `SELECT`- ja `FROM`-lausekkeet ovat pakollisia jokaisessa kyselylauseessa, mutta käytännössä lauseissa on lähes aina mu-

kana myös hakutulosta rajoittavia `WHERE`-ehtoja. Pilkulla erotettuna voidaan antaa useampia kenttien, taulujen tai näkymien nimiä lausekkeissa `SELECT`, `FROM`, `GROUP BY` ja `ORDER BY`. Sen sijaan `WHERE`- ja `HAVING`-lausekkeissa useampi ehto erotellaan `AND`-, `OR`- ja `NOT`-operaattoreilla.

Celko [2, s. 174–176] kuvaa **SQL-kyselylauseen suorituksen** yksityiskohtaisesti. Ensimmäisenä koostetaan **`FROM`-lausekkeessa** olevista taulu- tai näkymäviittauksista ristitulo (engl. *cross join*), josta aiemmin käytettiin myös termiä karteesinen tulo (engl. *Cartesian product*). Jokaisen tarvittavan taulun tai näkymän jokainen rivi kombinoidaan jokaisen muun taulun tai näkymän rivin kanssa, jolloin lopputuloksena saadaan kaikkien rivien kaikki kombinaatiot. Käytännössä Celkon mukaan mikään tietokannanhallintajärjestelmä ei tällaista ristituloa muodosta, koska siitä tulisi hyvin nopeasti varsin iso. Jo kahden tuhat riviä sisältävän taulun ristitulossa on miljoona riviä, joten kombinaation tallentamiseen ja käsittelyyn kuluisi huomattavan paljon muisti- ja tehoresursseja.

Seuraavassa vaiheessa mahdollisessa **`WHERE`-lausekkeessa** olevia ehtoja verrataan edellisen vaiheen ristitulolla saatuihin riveihin. Tarkastelun seurauksena vain arvon tosi saavat rivit annetaan kyselylauseen käsittelyn seuraavaan vaiheeseen. Celkon [2, s. 175] mukaan `WHERE`-lausekkeen ehdot voivat olla varsin monimutkaisia, ja niissä voi olla myös alikyselyitä. Alikyselyt ovat luvussa kuvatuin lausekkein käsiteltäviä kyselylauseita, jotka on sijoitettu pääkyselylauseen sisään suluilla erotettuna.

Jos kyselylauseessa on käytetty **`GROUP BY` -lauseketta**, muodostetaan edellisessä vaiheessa saadusta välituloksesta uusi välitulos siten, että jokaisessa ryhmässä on kyselylauseen ryhmittelyehdossa annetuilla kentillä sama arvo. Celkon mukaan puuttuvia arvoja eli `NULL`-arvoja (ks. luku 5.4) käsitellään kuin ne olisivat samanarvoisia toistensa kanssa. Ryhmittelyn seurauksena saadut ryhmät supistetaan yksittäisiksi riveiksi uuteen välitulokseen.

Jos kyselylauseessa on käytetty ryhmittelyn lisäksi myös **`HAVING`-lauseketta**, verrataan sen ehtoja ryhmittelyn tuloksena saatuihin riveihin. Kuten `WHERE`-lausekkeen ehtojen tarkastelussa, tässäkin välitulokseen otetaan mukaan vain ne ryhmät, jotka saavat tarkastelussa arvon tosi. Celkon [2, s. 176] mukaan ehdottomana edellytyksenä on se, että `HAVING`-lausekkeen ehdot käsittelevät ryhmittelyn seurauksena saatua välitulosta tai ryhmän ominaisuuksia, eivätkä yksittäisiä rivejä, joista alunperin ryhmä muodostettiin `GROUP BY` -lausekkeen käsittelyn aikana.

Lopuksi kyselylauseen tuloksena edellisten operaatioiden aikana saadusta viimeisimmästä välituloksesta palautetaan omina sarakkeineen **SELECT-lausekkeen** mukaiset kentät, laskutoimitukset ja vakioarvot. Laskutoimituksia ovat vaikkapa merkkijonossa olevan osamerkkijonon paikan selvittäminen `POSITION`-funktiolla tai ryhmittelyn seurauksena saadun ryhmän alkuperäisten rivien lukumäärän selvittäminen `COUNT`-funktiolla. Jos kyselylauseessa on käytetty ryhmittelyä, on Celkon [2, s. 176] mukaan `SELECT`-lausekkeessa oltava mukana kaikki ryhmittelyehdot.

Elmasri painottaa kirjassaan [6, s. 475], että taulujen tietoja tulee yhdistellä SQL-kyselyissä vain perusavainten (engl. *primary key*) ja viiteavainten (engl. *foreign key*) avulla. Muussa tapauksessa SQL-kyselyn tulosjoukkoon voi ilmestyä epäkelpoja ja virheellisiä rivejä (engl. *invalid data* tai *spurious data*) ja siten suorastaan vääriä tietoja.

5.4 SQL-operaattoreita

Elmasrin [6, s. 267] mukaan `SELECT`-lausekkeen **kenttiä ja laskutoimitusten tuloksia** sekä `FROM`-lausekkeen **tauluja voidaan nimetä** `AS`-määritteen avulla. Esimerkkejä taulujen lyhyempään asuun nimeämisestä on luvussa 5.8.

Celkon [2, s. 143] mukaan operaattoreiden `BETWEEN` ja `OVERLAPS` avulla voidaan selvittää, onko verrattava arvo kahden annetun päätepisteen välissä. Päätepisteet lasketaan mukaan arvoalueeseen. Celkon [2, s. 143] mukaan `BETWEEN`-operaattoria voidaan käyttää minkä tahansa vertailtavien tietotyyppien yhteydessä, kun taas `OVERLAPS`-operaattori toimii ainoastaan aikaleimatyypeillä. Esimerkiksi SQL-lause

```
SELECT henkilöID, sukunimi, etunimi
FROM henkilö
WHERE sukunimi BETWEEN 'Aalto' AND 'Kulmanen'
ORDER BY sukunimi, etunimi;
```

palauttaa luvun 4.9 taulukon 4.5 `henkilö`-taulusta haettuna kaksi ensimmäistä riviä.

Celkon [2, s. 130–131] mukaan tietokannanhallintajärjestelmä suorittaa tarvittaessa automaattisesti arvojen **tietotyypin muuntamisen toiseen tietotyyppiin** vaikkapa vertailuja varten. Celkon mukaan `CAST`-funktiolla voidaan pakotettuna muuntaa tietotyyppejä toisiksi. Esimerkiksi merkkijonona esitetystä syntymä-

vuodesta voidaan laskea PostgreSQL-tietokannanhallintajärjestelmällä henkilön ikä vuonna 2008 seuraavasti käyttäen apuna `CAST`-funktiota:

```
SELECT 2008 - CAST('1956' AS INTEGER) AS ikä;
```

Em. kyselyn lopputuloksena saadaan `INTEGER`-tyyppisenä kokonaislukuna arvo 52.

Jos `SELECT`-lausekkeessa (ks. luku 5.3) on `DISTINCT`-määrite, palautetaan Elmasrin [6, s. 257] mukaan kyselylauseen lopputuloksena vain ne rivit, joiden arvot eroavat toisistaan yhden tai useamman kentän osalta. Tällöin siis lopputuloksessa ei ole mukana **toistuvia rivejä** (engl. *duplicate rows*).

Elmasrin [6, s. 473] mukaan jostakin kentästä voi puuttua arvo, mikäli kentän avulla kuvattu ominaisuus ei koske tallennettua tietuetta tai kenttään tallennettavaa arvoa ei tiedetä. Kenttään tarkoitettu arvo voi hänen mukaansa toisinaan olla tiedossa, mutta sitä ei ole vielä tallennettu kenttään. Tällaisia **puuttuvia arvoja** merkitään Elmasrin [6, s. 200] mukaan SQL-kielessä termillä `NULL`.

Elmasrin [6, s. 267] mukaan kentän arvoa tulee verrata puuttuvaan arvoon operaattorin `IS` tai `IS NOT` avulla, eikä tavanomaisilla vertailuoperaattoreilla, kuten = ja <>. Tämä johtuu Elmasrin mukaan siitä, että **puuttuva arvo ei ole samanarvoinen edes minkään toisen puuttuvan arvon kanssa**. Celkon [2, s. 101] mukaan puuttuvien arvojen tulkinta johtaa tavanomaisen Boolen kaksiarvoisen (tosi tai epätosi) logiikan sijasta kolmiarvoiseen logiikkaan, jossa arvoina toimivat siis tosi (engl. *true*), epätosi (engl. *false*) ja tuntematon (engl. *unknown*). Celko [2, s. 96] kehottaa välttämään puuttuvia arvoja, mutta käyttämään niitä tarvittaessa oikein.

Celkon [2, s. 139–142] mukaan `LIKE`-operaattorilla voidaan etsiä **täsmääviä merkkijonoja** käyttäen hakuehdossa prosenttimerkkiä `%`, joka vastaa mitä tahansa merkkiä nollan tai useamman kerran. Etsinnässä voidaan käyttää myös alleviivamerkkiä _, joka vastaa aina tasan yhtä merkkiä. Luvussa 5.8 on esitetty esimerkki `LIKE`-operaattorin käytöstä.

Alikysely (engl. *subquery* tai *nested query*) on Elmasrin [6, s. 261] mukaan täydellinen SQL-kysely, joka on sijoitettu pääkyselyksi kutsutun kyselyn `WHERE`-lausekkeeseen suluilla eroteltuna. Kun alikyselyn `WHERE`-lausekkeessa hyödynnetään jotain pääkyselyssä esiteltyä taulua, sanotaan alikyselyn ja pääkyselyn olevan Elmasrin [6, s. 263] mukaan **kytköksessä** tai korrelaatiosuhteessa (engl. *correlated reference*). Kytköksessä oleva alikysely suoritetaan Elmasrin mukaan tällöin käytännössä kertaalleen jokaista pääkyselyn tietuetta kohti.

Alikyselyitä voidaan hyödyntää mm. `IN`-predikaatin avulla. Pääkyselyssä olevan kyseistä predikaattia hyödyntävän operaation tuloksena saadaan Elmasrin [6, s. 261] mukaan arvo tosi, mikäli verrattava arvo on jokin arvojoukossa eli alikyselyn tuloksessa esiintyvistä arvoista. `EXISTS`-rakennetta hyödynnetään Elmasrin [6, s. 264] mukaan selvitettäessä, onko kytköksessä käytetyn alikyselyn tuloksessa yhtään riviä. Celkon [2, s. 164] mukaan `EXISTS`-rakennetta käyttävä alikysely suoriutuu yleensä nopeammin kuin `IN`-rakennetta käyttävä alikysely, mutta paras lähestymistapa riippuu hänen mukaansa mm. pääkyselyn ja alikyselyn rivimääristä ja kenttien arvojen jakaumasta. Luvuissa 5.8 ja 8 on esimerkkejä alikyselyistä.

Celkon [2, s. 312] mukaan `UNION`-lausekkeella voidaan **liittää kaksi hakutulosta yhteen**. Hakutulosten täytyy olla samarakenteisia (engl. *union compatible*), jolloin niissä on sama määrä sarakkeita ja niiden vastinsarakkeilla on samat arvojoukot. Jos arvojoukot eivät ole täsmälleen samat, tulee kenttien eri arvojoukkojen olla automaattisesti muunnettavissa samaan yhteiseen arvojoukkoon (ks. edellä mainittu `CAST`-funktio).

5.5 Taulujen kenttien tietotyypit

Tietokannan rakennetta suunniteltaessa tulee taulujen kentissä käytettävät tietotyypit harkita tarkkaan tietokannan käyttötarkoituksen mukaan. Hofferin [7, s. 212] mukaan **tallennettavat tiedot ja näiden jakautuminen eri arvoihin** tulee ensin määritellä tarkasti, jonka jälkeen järjestelmän tarjoamista vaihtoehdoista parhaan tietotyypin löytäminen kuhunkin käyttökohteeseen on helpompaa. Valitun tietotyypin avulla tulee olla mahdollista tallentaa kaikki käyttökohteen eri arvot sekä estää kelvottomien arvojen tallentaminen. Lisäksi Hofferin [7, s. 209] mukaan valitun tietotyypin tulisi minimoida tietojen tallennukseen tarvittava tila samalla, kun tiedon eheys (engl. *data integrity*) maksimoidaan. Oikean tietotyypin tulisi tukea vaadittavaa tietojenkäsittelyä, kuten aritmeettisia toimenpiteitä numeerisilla tyypeillä.

Tietokannanhallintajärjestelmissä on Celkon [2, s. 47, 61 ja 81] mukaan tarjolla useita standardeja tietotyyppejä merkkien, merkkijonojen, numeeristen arvojen ja aikaleimojen esittämiseen, puhumattakaan tietokannanhallintajärjestelmien valmistajien epästandardeista lisäyksistä tyyppivalikoimaan. Liitteen B taulukoissa on esitelty PostgreSQL-tietokannanhallintajärjestelmän tukemat [27] sekä SQL-92 -standardin mukaiset tietotyypit [2, s. 47–94].

Tietokantoja hyödyntävissä sovelluksissa tulisi pyrkiä käyttämään standardeja

tietotyyppejä, jotta tietokannan muuntaminen toiseen tietokannanhallintajärjestelmään myöhemmin onnistuisi vaivattomammin. Samankin tietokannanhallintajärjestelmän myöhemmissä versioissa epästandardeja ominaisuuksia on saatettu karsia tai niiden toimintaa on muutettu, jonka takia tietokannan rakenteeseen ja myös sovelluksen lähdekoodiin joudutaan tekemään muutoksia. Muutokset eivät välttämättä ole helppoja toteuttaa.

5.6 Tietojen hakua tehostava hakemisto eli indeksi

Tietokannan taulun yhteen tai useampaan kenttään voidaan Elmasrin [6, s. 155–156] mukaan määritellä **hakemisto eli indeksi** (engl. *index*, monikko *indexes* tai *indices*), joka tehostaa tietojen hakemista taulusta. Periaate on sama kuin kirjoissa, joissa teoksen lopussa saattaa olla asiasanalista sivunumeroineen. Kyseisen listan avulla tietyn asiasanan etsiminen on nopeampaa kuin käydä läpi koko kirja sivu sivulta. Kirjoissa ovat apuna otsikot ja sisällysluettelo, mutta silti asiasanalista on ainoa tehokas keino etsittävän sanan löytymiseksi tai sen selvittämiseksi, ettei asiasanaa kirjasta löydy lainkaan.

Indeksi kannattaa luoda **yksilöivänä hakemistona** (engl. *unique index*), jos indeksissä määriteltyyn kenttään tai kenttäkokoelmaan ei haluta useita samoja arvoja (engl. *duplicate values*). Esimerkiksi henkilöiden perustiedot tallentavassa taulussa henkilötunnuksen sisältävässä kentässä voisi olla indeksi yksilöivänä hakemistona. Sen sijaan henkilön sukunimi- ja etunimihakuja tehostavan indeksin ei haluttane olevan yksilöivä, koska samannimisiä henkilöitä on olemassa. Luvussa 5.8 on esimerkkejä indeksien luomisesta.

5.7 Nyrkkisääntöjä indeksien hyödyntämisestä

Indeksien oikeanlainen hyödyntäminen on Mullinsin [11, s. 298] mukaan paras tietokannan valvojien suorittamista tietokannan tehokkuutta lisäävistä tekniikoista. Indeksien analysoinnin tavoitteena on Mullinsin [11, s. 299] mukaan se, että kyselyiden suorittamisen aikana haetaan **vähemmän dataa** (engl. *less I/O*) levymuistissa sijaitsevasta tietokannasta. Hänen mukaansa tehokkaan indeksointistrategian ansiosta **levyoperaatioiden määrä vähenee oleellisesti**, kun samalla **indeksien ajan tasalla pitämiseksi joudutaan suorittamaan hyväksyttävä määrä lisäoperaatioita** (ks. luku 6.1). Indeksejä ei Hofferin [7, s. 276] mukaan luonnollisestikaan kannata

luoda jokaiseen taulun kenttään.

Hoffer luettelee kirjassaan [7, s. 233] seuraavat **nyrkkisäännöt** indeksien tarpeellisuuden pohdinnan tueksi:

- Indeksit ovat käyttökelpoisimpia isoissa tauluissa.
- Taulun perusavaimeen tulee määritellä yksilöivä hakemisto (engl. *unique index*). Muun muassa PostgreSQL luo aina automaattisesti taulujen perusavaimiin yksilöivän hakemiston [20].
- Indeksit ovat hyödyllisimpiä kentissä, joita käytetään `WHERE`-lausekkeen (ks. luku 5.3) ehdoissa joko haluttujen rivien löytämiseksi vakioarvojen avulla tai johonkin toiseen tauluun viittaavan viite-eheyden avulla.
- SQL:n järjestys- ja ryhmittelylausekkeissa (`ORDER BY` ja `GROUP BY`, ks. luku 5.3) usein käytettäviin kenttiin kannattaa luoda indeksi, koska tämä saattaa nopeuttaa kyselyn lopputuloksen järjestämistä tai ryhmittelyä. Hofferin [7, s. 233] mukaan on kuitenkin tutkittava erityisen tarkkaan se, käyttääkö tietokannanhallintajärjestelmä tällaisissa tapauksissa todellakin indeksejä vai ei.
- Jos kentän arvot vaihtelevat suuresti, indeksistä on todennäköisesti hyötyä. Hoffer ehdottaa kirjassaan [7, s. 233] Oracle-yhtiön suosituksen mukaisesti, että indeksi on selkeästi hyödyllinen vasta, jos kentässä on vähintään sata eri arvoa.
- Hofferin mukaan indeksistä on apua vain silloin, kun kyselyn lopputuloksen tuottamiseen on käytetty enintään 20 prosenttia kaikista taulun riveistä. Mullins asettaa kirjassaan [11, s. 130] rajaksi 25 prosenttia.
- Joissakin tietokannanhallintajärjestelmissä on rajoitettu indeksien lukumäärää taulua kohden. Tauluun tulisi siten jättää vain ne indeksit, joista on todennäköisesti eniten hyötyä tehokkuuden saavuttamisessa.
- Puuttuvan arvon eli `NULL`-arvon sallivien kenttien indeksoinnissa tulee olla tarkkana. Useilla tietokannanhallintajärjestelmillä `NULL`-arvoja sisältäviä rivejä ei oteta lainkaan mukaan indeksiin, joten tällaiset rivit etsitään hitaammalla koko taulun läpikäyvällä tavalla.

Mullins mainitsee kirjassaan [11, s. 130] lisäksi, että viittaaviin kenttiin luodut indeksit saattavat tehostaa viite-eheyksien voimaansaattamista. Mullins [11, s. 300]

kertoo, että indeksiin kannattaa joskus sijoittaa enemmän kenttiä kuin normaalisti olisi tarpeellista. Tämä johtuu hänen mukaansa siitä, että jos kaikki SQL-kyselyn tarvitsemien kenttien tiedot sijaitsevat jo indeksissä, voidaan mahdollisesti välttää lukeminen itse tietokantataulusta. Tämä taasen saattaa tehostaa tietojen hakua kokonaisuutena.

Elmasri neuvoo kirjassaan [6, s. 550 ja 553], että **ensisijaisesti kannattaa luoda vain todella hyödylliset ja hakuja nopeuttavat indeksit**. Tarvittaessa voidaan lisätä indeksejä, kun niille on selkeästi tarvetta. Kyselyiden muuttuessa toisenlaisiksi on myös mahdollista, ettei joitakin vanhoja indeksejä enää käytetä, jolloin niitä voidaan poistaa turhina. Mullins ehdottaa kirjassaan [11, s. 131], että säännöllisesti muuttuviin kenttiin ei luotaisi indeksejä lainkaan.

Mullinsin [11, s. 131] mukaan jonkin tietyn SQL-kyselyn nopeuttamiseen luotu **indeksi saattaa epäsuotuisasti hidastaa muiden SQL-kyselyiden suoritusta**. Indeksin lisääminen ei siten aina tuo nopeutushyötyä kaikissa niissä kyselyissä, joissa nopeutta oletetaan tulevan lisää. Koska indeksi on taulukohtainen eikä kyselykohtainen, on lisättyä indeksiä mahdollista käyttää kaikissa kyselyissä, eikä ainoastaan tehottomuudesta kärsineissä kyselyissä.

Monessa kyselyssä hyödynnetään samoja kenttiä, jolloin **indeksin luominen johonkin usein käytettyyn taulun kenttään saattaa nopeuttaa useita kyselyitä**. Koska em. nyrkkisääntöjen kolmannen kohdan mukaan kyselyissä vertaillaan yleensä tauluissa olevia viiteavaimia jonkin toisen taulun perusavaimeen, indeksi kannattaa mahdollisesti luoda jonain **viiteavainten yhdistelmänä**. Silloin samaa indeksiä voidaan käyttää muunkinlaisissa kyselylauseissa, joissa välttämättä ei edes tarvita kaikkia indeksin määrittelyssä mukana olevia kenttiä. Elmasrin [6, s. 553] mukaan tiettyä indeksiä voidaan hyödyntää vain silloin, kun sen määrittely sisältää samassa järjestyksessä alusta lukien kyselyn `WHERE`-ehdoissa (ks. luku 5.3) tarvittuja kenttiä.

PostgreSQL:n versio 7.4 dokumentaationsa [16] mukaan toimii **useampikenttäisten indeksien** (engl. *multicolumn indexes*) suhteen mainitulla tavalla, jossa oleellista on indeksin määrittelyssä käytettyjen kenttien järjestys. Kyseisellä PostgreSQL:n versiolla 7.4 SQL-kyselyssä indeksistä hyötyvän ehdon tulee olla liitettynä muihin kyselyn ehtoihin `AND`-operaattorilla, jonka lisäksi kyselyssä käytettävän tiettyä kenttää koskevan vertailuehdon tulee olla yhteensopiva indeksin tyypin kanssa. PostgreSQL:ssa indeksin tyyppi on oletuksena yleisimmin B-puu (engl. *B Tree*), joka sallii tavanomaiset yhtäsuuruuden, pienemmyyden ja suuremmuuden vertailut operaattoreilla `<`, `<=`, `=`, `>=` ja `>`. B-puu on relaatiotietokantasanaston

[32] mukaan ajonaikaisesti tasapainoisena pysyvä puumainen tietorakenne, jota käytetään lähinnä indeksien toteutukseen.

PostgreSQL alkaen versiosta 8.1 osaa dokumentaationsa [17] mukaan hyödyntää indeksiä myös silloin, kun kyselyn hakuehdoissa on mukana mikä tahansa osajoukko indeksin määrittelevistä kentistä. Näillä uusillakin PostgreSQL:n versioilla indeksi on dokumentaationsa [17] mukaan silti tehokkaimmillaan vain silloin, kun kyselyn hakuehdoissa vertaillaan indeksiin sisältyviä kenttiä määrittelyjärjestyksessään alusta lukien.

PostgreSQL:n dokumentaatiossa [17] suositellaan, että **useampikenttäisiä indeksejä tulisi käyttää säästeliäästi**. PostgreSQL osaa versiosta 8.1 alkaen yhdistää kyselyn aikana useita indeksejä siten, että myös `OR`-operaattorilla liitetyissä ehdoissa voidaan tehokkaasti hyödyntää indeksejä. Silti edelleen tulee tarkasti harkita se, luodaanko tarpeen mukaan useampikenttäisiä indeksejä vai luotetaanko mainittuun PostgreSQL:n versiosta 8.1 lähtien osaamaan indeksien yhdistämistekniikkaan.

5.8 Esimerkkejä SQL-kyselyistä ja indekseistä

Tarkastellaan luvuissa 4.7 ja 4.9 esiteltyjen taulukoiden 4.2, 4.3, 4.4 ja 4.5 avulla SQL-kyselyitä normalisoidussa tietokannassa. Liitteessä C on esitetty taulurakenteen `CREATE TABLE` -luontilauseet, tietojen tallennukseen käytetyt `INSERT`-lauseet sekä taulujen poistamiseen käytetyt `DROP TABLE` -lauseet.

Henkilölistaus valtiotietoineen saadaan seuraavalla kyselyllä:

```
SELECT h.henkilöID, h.sukunimi, h.etunimi, v.valtiokoodi, v.valtio
FROM henkilö AS h, valtio AS v
WHERE h.valtiokoodi = v.valtiokoodi
ORDER BY LOWER(h.sukunimi), LOWER(h.etunimi);
```

Kyselyssä taulut on nimetty `AS`-määritteen avulla lyhyempään asuun. Kyselyn hakutulos on sama kuin taulukossa 4.2 esitetty. Hakutulos on järjestetty henkilön sukunimen ja etunimen mukaan aakkoselliseen järjestykseen, eikä nimien kirjainkoolla ole merkitystä järjestysehdossa käytettyjen `LOWER`-funktioiden takia.

Koska liitteen C mukaisesti henkilön valtiokoodi ei ole pakollinen tieto eli kyseiselle kentälle `henkilö`-taulussa on sallittu `NULL`-arvo, jäivät hakutuloksesta pois sellaiset henkilöt, joille ei ole valtiokoodia määritelty. Kaikkien henkilöiden perus-

tietojen listaamiseksi joudutaan käyttämään **SQL:n ulkoliitosta** (`OUTER JOIN`) seuraavasti:

```
SELECT h.henkilöID, h.sukunimi, h.etunimi, v.valtiokoodi, v.valtio
FROM henkilö AS h LEFT OUTER JOIN valtio AS v
ON (h.valtiokoodi = v.valtiokoodi)
ORDER BY LOWER(h.sukunimi), LOWER(h.etunimi);
```

Muutoksessa `WHERE`-lausekkeen ainoa ehto siirrettiin ulkoliitokseen kuuluvaksi `ON`-ehdoksi, ja lisähakuehtoja voitaisiin tavanomaiseen tapaan sisällyttää `WHERE`-lausekkeeseen. Ulkoliitoksen `LEFT`-määrite tarkoittaa sitä, että erityisesti mukaan halutaan kaikki henkilöt, jolloin `henkilö`-taulu esitetään lausekkeessa `OUTER JOIN` -määritteen vasemmalla puolella.

Kun halutaan selvittää, monenko eri hyödykelajin hyödykkeitä henkilöt omistavat, voidaan käyttää seuraavanlaista kyselyä:

```
SELECT h.henkilöID, h.sukunimi, h.etunimi, COUNT(*) AS lajien_lkm
FROM henkilö AS h, henkilön_omaisuus AS o
WHERE h.henkilöID = o.henkilöID
GROUP BY h.henkilöID, h.sukunimi, h.etunimi
ORDER BY LOWER(h.sukunimi), LOWER(h.etunimi);
```

`SELECT`-lausekkeessa voidaan käyttää `AS`-määritettä palautettavien kenttien nimien uudelleennimeämisessä (ks. luku 5.4). Kyselylauseen hakutulos on esitetty taulukossa 5.1. Jos henkilö ei omista mitään, ei hän tule mukaan tähän hakutulokseen.

henkilöID	**sukunimi**	**etunimi**	**lajien_lkm**
3	Kolmonen	Jaakko	1
1	Meikäläinen	Matti	2
7	Presley	Elvis	3

Taulukko 5.1: Esimerkkitaulurakenteen henkilöiden omistamien hyödykelajien lukumäärät.

Kun edellisen kyselyn hakutulokseen halutaan mukaan vain asunnon omistavat henkilöt, voidaan lisätä `WHERE`-lausekkeeseen vaikkapa `EXISTS`-alikysely seuraavasti:

```
SELECT h.henkilöID, h.sukunimi, h.etunimi, COUNT(*) AS lajien_lkm
FROM henkilö AS h, henkilön_omaisuus AS o
WHERE h.henkilöID = o.henkilöID
AND EXISTS (
  SELECT o2.henkilöID FROM henkilön_omaisuus AS o2
  WHERE h.henkilöID = o2.henkilöID
  AND LOWER(o2.hyödyke) LIKE 'asunto%'
)
GROUP BY h.henkilöID, h.sukunimi, h.etunimi
ORDER BY LOWER(h.sukunimi), LOWER(h.etunimi);
```

Kyselylauseen hakutuloksena saadaan taulukon 5.1 hakutulos ensimmäistä riviä lukuunottamatta. Alikyselyssä etsitään `LIKE`-operaattorilla (ks. luku 5.4) täsmääviä hyödykelajeja käyttäen prosenttimerkkiä `%`, joka vastaa mitä tahansa merkkiä nollan tai useamman kerran. Prosenttimerkillä varustettua hakuehtoa käytettäessä hyödykkeen kuvauksen alussa tulee olla merkkijono `asunto`, jotta se täsmäisi etsittyyn merkkijonoon.

`EXISTS`-rakennetta käyttävä alikysely voidaan Celkon [2, s. 156] mukaan yleensä helposti muuntaa `IN`-rakenteeksi tai toisinpäin. Edellistä kyselyä vastaava `IN`-alikyselyä hyödyntävä kyselylause on seuraavanlainen:

```
SELECT h.henkilöID, h.sukunimi, h.etunimi, COUNT(*) AS lajien_lkm
FROM henkilö AS h, henkilön_omaisuus AS o
WHERE h.henkilöID = o.henkilöID
AND h.henkilöID IN (
  SELECT o2.henkilöID FROM henkilön_omaisuus AS o2
  WHERE LOWER(o2.hyödyke) LIKE 'asunto%'
)
GROUP BY h.henkilöID, h.sukunimi, h.etunimi
ORDER BY LOWER(h.sukunimi), LOWER(h.etunimi);
```

Em. `IN`-rakennetta käyttävän kyselyn hakutulos on luonnollisesti sama kuin vastaavan `EXISTS`-rakennetta käyttävän kyselyn hakutulos.

Edellä mainittujen henkilölistauksien ja hyödykelajien lukumäärien hakemiseen käytettyjen SQL-lauseiden nopeuttamiseksi voidaan luoda tietokantaan vaikkapa seuraavat indeksit:

```
CREATE INDEX henkilö_valtiokoodi ON henkilö(valtiokoodi);
CREATE INDEX henkilön_omaisuus_hyödyke ON henkilön_omaisuus(hyödyke);
```

Mikäli esimerkkitaulurakenteessa olisi enemmän dataa, mainitut kyselyesimerkit todennäköisesti hyötyisivät em. indekseistä. Tietokannanhallintajärjestelmä

päättelee luvun 5.7 nyrkkisääntöjen mukaisesti vähäisellä datamäärällä, että indeksejä ei kannata käyttää. Taulun sisällön läpikäynti kokonaisuudessaankin on nimittäin tarpeeksi nopeata, koska todennäköisesti se sopii yhteen tai korkeintaan muutamaan levylohkoon. Tällöin kyseiset levylohkot joudutaan joka tapauksessa hakemaan levyltä, eikä niiden lisäksi kannata ensin hakea levyltä indeksin sisällön tallentamiseen käytettyä levylohkoa.

5.9 SQL-standardien noudattaminen Korppi-järjestelmässä

SQL-standardeja SQL-92 ja SQL-99 noudattamalla saadaan aikaan suhteellisen nopea ja siirrettävä järjestelmä. Korppi-järjestelmä on pyritty toteuttamaan SQL-standardeja noudattavaksi, ja PostgreSQL-tietokannanhallintajärjestelmän omien laajennuksien käyttö on dokumentoitu erityisen huolella. On kuitenkin huomattu Korppi-järjestelmän kehityksen aikana käytetyn myös sellaisia ominaisuuksia, joiden on virheellisesti oletettu olevan SQL:n määrittelemiä standardeja. Luvussa kuvataan muutamia tällaisten ominaisuuksien käytöstä aiheutuneita ongelmia ja niiden ratkaisuja.

Eräs Korppi-järjestelmän epästandardeista aiemmin hyödynnetyistä ominaisuuksista on `TIME`-funktio, joka palauttaa päivämäärän ja kellonajan sisältävästä `TIMESTAMP`-tyyppisestä kentästä kellonajan. Esimerkiksi funktiokutsun `TIME('2004-03-14 15:23:23.000')` tulos on `15:23:23.000`. Muun muassa `TIME`-funktio poistettiin käytöstä PostgreSQL:n versiossa 7.2 [29].

Kun Korppi-järjestelmän PostgreSQL-tietokannanhallintajärjestelmä vaihdettiin versiosta 7.1 versioon 7.3 vuodenvaihteessa 2003–2004, jouduttiin lähdekoodissa olleet `TIME`-funktiokutsut muuttamaan pikaisesti toisenlaisiksi. Tässä tapauksessa onneksi korjaus oli helppo, koska aikaleiman esitystapa tiedetään. Viittaukset `TIME`-funktioon korvattiin viittauksella `SUBSTRING`-funktioon, joka hakee merkkijonona esitettävästä aikaleimasta osamerkkijonon. SQL-lauseissa käytetyt funktiokutsut `TIME(kenttänimi)` vaihdettiin muotoon `SUBSTRING(kenttänimi FROM 12) AS time`, koska aikaleimassa esitetään kellonaika 12. merkistä alkaen. Lisäksi tulos palautetaan `time`-nimisessä kentässä, koska tämännimisessä kentässä funktiokutsujen `TIME(kenttänimi)` tulokset on myös oletuksena palautettu. Tällöin Java-lähdekoodin muokkaus kyseisen kentän hyödyntämisen osalta oli tarpeetonta.

Tietokantaan olisi mainitun ongelman ratkaisemiseksi voitu myös luoda oma `TIME`-funktio, joka palauttaisi osamerkkijonon aikaleimasta suoraan tietokannan-

hallintajärjestelmän avulla. Tämä olisi kuitenkin luultavasti ollut tulevaisuutta ja tulevia tietokannanhallintajärjestelmän versioita ajatellen huonompi ratkaisu, koska varsin helposti kellonaika voidaan aikaleimasta selvittää ilman kyseistä funktiotakin. Järjestelmän ylläpidettävyyden kannalta kaikista ylimääräisistä epästandardeista ratkaisuista tulisi luopua.

SQL:n määrittelemällä standarditavalla aikatietoja tulee hakea käyttäen `EXTRACT`-funktiota. Esimerkiksi tuntiosa saadaan selvitettyä `TIMESTAMP`-tyyppisestä aikaleimasta funktiokutsulla `EXTRACT(HOUR FROM kenttänimi)`.

Myös toinen Korppi-järjestelmän epästandardien ominaisuuksien käyttö liittyy aikaleimoihin. Syksyllä 2001 kalenteriosion toteuttanut Kolibri-projektiryhmä käytti PostgreSQL:n omia epästandardeja `TIMESTAMP_GE`- ja `TIMESTAMP_LE`-funktioita aikaleimojen järjestyksen vertailuun. Loppuvuodesta 2003 näiden funktioiden toimintaa ja aikaleimojen järjestettävyyttä tutkittiin tarkemmin, jolloin huomattiin kyseiset funktiot voitavan korvata suoraan aikaleimojen vertailulla muun muassa operaattoreiden < ja > avulla. Tällainen suora vertailu on suoritettujen testien mukaan paitsi yhtä tehokasta myös erityisesti siirrettävämpää kuin PostgreSQL:n tarjoamien epästandardien vertailufunktioiden käyttö.

SQL:n määrittelemään vaihtuvamittaisten merkkijonojen tallentamiseen tarkoitettuun `CHARACTER VARYING` -tyyppiin liittyy Korppi-järjestelmässä eräs valitettava PostgreSQL:n epästandardin piirteen hyödyntäminen. Korppi-järjestelmän tietokannan tauluihin mainituntyyppisiä merkkijonokenttiä luotaessa ei nimittäin ole useimmiten asetettu kokorajoitusta tallennettavalle tiedolle, joten näihin kenttiin voidaan käytännössä tallentaa kuinka pitkiä merkkijonoja tahansa, tietenkin levymuistien tallennuskapasiteetin ja PostgreSQL:n rajoissa. Kokorajoitusta vailla olevia kyseisentyyppisiä kenttiä on noin puolessa Korppi-järjestelmän tietokannan tauluista, joten ongelma on melko laaja.

Ongelma ratkeaisi luomalla tietokanta uudestaan siten, että taulujen luontilauseissa kyseistä `CHARACTER VARYING` -tyyppiä koskevat rivit muokattaisiin käyttämään jotain tarpeeksi isoa kokorajoitusta. Käytettävä kokorajoitus tulisi valita tauluihin tallennetun datan perusteella. Lisäksi lähdekoodissa tulisi tarkistaa tilanne, jossa kenttään yritetään tallentaa maksimipituutta pitempiä merkkijonoja. Tällöin Korppi-järjestelmän käyttäjälle tulisi palauttaa selvä virheilmoitus liian pitkästä tallennettavasta arvosta.

Kokorajoitus voisi olla määritelty kenttäkohtaisesti. Tyyppitaulujen kentille kyseinen kokorajoitus voisi olla suhteellisen pieni, koska tällaisiin tauluihin tallenne-

taan staattista harvoin muuttuvaa tietoa. Mainitunlaisissa tyyppitauluissa on tyypin nimen ohella tarvittaessa erikseen kuvauskenttä, jonka kokorajoitus tulisi olla isompi. Mainitulla tavalla aikaansaatu keinotekoinen tallennettavan tiedon maksimipituus tuskin vaikuttaa kovinkaan paljoa tietokannan rakenteeseen, tiedon lisäämiseen tauluihin tai hakujen nopeuteen.

PostgreSQL osaa dokumentaationsa [30] mukaan versiosta 7.4 alkaen aiemmista versioista poiketen ryhmitellä (SQL:ssa `GROUP BY`, ks. luku 5.3) hakutuloksen ilman järjestämistä (SQL:ssa `ORDER BY`, ks. luku 5.3). Korppi-järjestelmässä hyödynnettiin kyseistä rivien automaattista järjestymistä muutamien sellaisten ryhmittelyiden yhteydessä, joissa ei erikseen annettu järjestämisehtoa. Näissä siis Java-lähdekoodissa oletettiin, että hakutulos palautetaan ryhmittelyehtojen mukaisessa järjestyksessä. Kun PostgreSQL:n versio 7.4 otettiin Korppi-järjestelmässä testikäyttöön vuoden 2005 lopussa, osa kyseistä järjestysoletusta käyttävästä sovelluslogiikasta toimi virheellisesti. Kaikki ryhmittelyehdot tarkistettiin ja puuttuvat järjestelyehdot lisättiin.

6 Tietokannan suorituskyvyn valvonta

Luvussa kuvataan tietokannan suorituskykyyn vaikuttavia tekijöitä, tietokannanhallintajärjestelmän keräämien käyttötilastojen hyödyntämistä sekä kyselyiden suorittamistavan analysointityökalujen hyödyntämistä. Lisäksi luvussa tarkastellaan sovellusten kirjoittamista lokitiedostoista saatavaa hyötyä. Luku perustuu pääosin lähteisiin [11, s. 249–252 ja 322–327], [15] sekä [22].

6.1 Tietokannan suorituskykyyn vaikuttavat tekijät

Mullinsin [11, s. 249] mukaan ei ole yllättävää, että tietokantojen suorituskyvyn valvonta ja tehostaminen (engl. *performance monitoring and tuning*) ovat yleensä ensimmäisinä mieleentulevia asioita, joita tietokannan valvojat (engl. *database administrator*, DBA) tekevät. Mullins [11, s. 250–251] jaottelee **tietokannan suorituskykyyn** vaikuttaviksi tekijöiksi kuormituksen, suoritustehon, resurssit, optimoinnin ja kilpailun.

Kuormituksella (engl. *workload*) tarkoitetaan kaikkia niitä kyselyitä ja komentoja, jotka kuormittavat järjestelmää kunakin ajanhetkenä. Kuormitus voi vaihdella huomattavasti riippuen ajanhetkestä, sillä vaikkapa öisin järjestelmää ja sen tietokantaa ei välttämättä käytetä yhtä paljon kuin päivällä. Kokonaiskuormituksella on Mullinsin mukaan erittäin suuri merkitys tietokannan suorituskyvylle.

Suoritusteho (engl. *throughput*) määräytyy palvelinlaitteiston tiedonkäsittelyn kokonaiskyvystä, johon luetaan tiedonsiirtoväylien ja prosessorien nopeus, palvelinlaitteiston rinnakkaiskäsittelyominaisuudet sekä käyttöjärjestelmän ja järjestelmäohjelmistojen tehokkuus. **Resurssit** (engl. *resources*) koostuvat järjestelmän sisältämistä ohjelmistoista ja laitteistosta, johon luetaan mm. levytallennuslaitteet, keskusmuistipiirit ja välimuistiohjaimet.

Optimointi (engl. *optimization*) voi olla minkä tahansa järjestelmän muokkaamista siten, että se toimii paremmin ja tehokkaammin. Tietokantaoptimointi käsittää sekä SQL-lauseiden uudelleenmuotoilut että tietokannan asetusten muokkaamisen. Tietokannanhallintajärjestelmä optimoi sisäisesti suoritusta siten, että kyselyn suorittamiseksi valittu saantipolku (engl. *access path*) on mahdollisimman tehokas.

Kilpailu (engl. *contention*) on tilanne, jolloin vähintään kaksi komponenttia yrittää käyttää yksittäistä resurssia ristiriitaisella tavalla. Kilpailun lisääntyessä suoritusteho laskee.

Em. viiden tekijän avulla Mullins määrittelee kirjassaan [11, s. 251] tietokannan suorituskyvyn vapaasti suomennettuna seuraavasti:

> *Tietokannan suorituskyky on resurssien käytön optimoimista suoritustehon kasvattamiseksi ja kilpailun vähentämiseksi, jotta suurin mahdollinen kuormitus voidaan käsitellä.*

Mullins mainitsee kuitenkin, että tietokantaa hyödyntävät sovellukset kommunikoivat myös muiden tietojärjestelmien ja komponenttien kanssa. Ne tulee myös ottaa huomioon, kun organisaatiossa suunnitellaan kokonaissuorituskykyyn liittyviä tekijöitä.

6.2 Tietokannan käyttötilastojen hyödyntäminen

Elmasrin [6, s. 550] ja Hofferin [7, s. 308] mukaan jokaisessa varteenotettavassa tietokannanhallintajärjestelmässä on jonkinlaisia työkaluja tietokannan historiatiedon ja käyttötilastojen esillesaamiseksi. Resurssien käytön muutosten ja suorituskykytilastojen koonti sekä analysointi ovat Mullinsin [11, s. 257] mukaan yksi **suorituskyvyn arviointiin liittyvistä tehtävistä**, jotka tietokannanhallintajärjestelmien valvojien tulee suorittaa.

PostgreSQL:n avulla hallittavassa tietokannassa tilastoja suoritetuista kyselyistä sekä niissä käytetyistä indekseistä ja tauluista saadaan tietyistä tietokannanhallintajärjestelmän tarjoamista **SQL-näkymistä** [15]. PostgreSQL kerää tilastoja vain silloin, kun sen asetuksista on tilastointi otettu erikseen käyttöön. Lisäksi asetuksista voidaan määrätä se, tyhjennetäänkö tilastot aina kun tietokannanhallintajärjestelmä käynnistetään, vai tuotetaanko tilastoja aiemman historian jatkoksi tietokannanhallintajärjestelmän käynnistyksistä riippumatta.

PostgreSQL:ssa esimerkiksi `pg_statio_user_indexes`-näkymä kertoo, paljonko tietokannassa olevia indeksejä on hyödynnetty. Vastaavasti `pg_statio_user_tables`-näkymä ilmoittaa tietokannan tauluihin kohdistuneiden lukuoperaatioiden määrät. Kyseisten näkymien avulla on helppoa listata vaikkapa indeksit, joita ei käytetä juuri lainkaan ja joista ei siten ole juurikaan hyötyä. Samaten mainittujen näkymien avulla voidaan tarkastella eniten hyödyn-

nettyjä tauluja vaikkapa nopeammalle fyysiselle levymuistille siirtämistä silmällä pitäen.

PostgreSQL:ssa `pg_class`-taulu luettelee kaikki järjestelmän ns. **systeemitaulujen ja -näkymien ominaisuudet sekä käyttäjän omassa tietokannassa olevien taulujen, indeksien ja näkymien ominaisuudet** [23]. Tauluja, indeksejä ja näkymiä kutsutaan PostgreSQL:ssa yhteisesti objekteiksi. PostgreSQL-tietokannassa olevien objektien nimet, kussakin objektissa olevan datan tallentamiseen käytetty levytila kilotavuina, objektissa olevien rivien lukumäärä sekä objektin tyyppi voidaan selvittää seuraavanlaisella kyselyllä:

```
SELECT relname AS name, relpages*8 AS size_KB,
CAST(reltuples AS integer) AS rows, relkind AS type
FROM pg_class
ORDER BY size_KB DESC;
```

Taulukossa 6.1 on esitetty muutama ensimmäinen rivi hakutuloksesta, kun em. kysely suoritetaan Korppi-järjestelmän tietokannassa. Datamääriltään suurimmat Korppi-järjestelmän tietokannan taulut ovat siis kooltaan useita satoja megatavuja, joten niiden hyödyntämisessä tulee olla tarkkana.

Tietokannan objekti	Koko (KB)	Rivejä (kpl)	Tyyppi
`syslog`	433128	2395583	r
`jore_suoritukset`	294216	1321770	r
`coursedescription`	283888	988212	r
`questionresult`	172872	1928637	r
`coursedescription_pkey`	127344	988212	i
`groupparticipant`	113456	1505940	r
`result`	105200	1482830	r
`event`	102184	908802	r
`courseparticipant`	83280	1009689	r
`questionfield_htmlparamete_pkey`	82880	473297	i
`syslog_pk`	78216	2395583	i
`questionresult_pkey`	75080	1928637	i

Taulukko 6.1: Korppi-järjestelmän tietokannassa olevia objekteja datan määrän mukaan laskevassa järjestyksessä.

Taulukossa 6.1 objektin tyyppi `type`-kentässä on joko `r` eli taulu (engl. *relation*), `i` eli indeksi (engl. *index*) tai `v` eli näkymä (engl. *view*). Lohkojen lukumäärä eli

`relpages`-kentän arvo on kerrottu levylohkon koolla, jotta varsinainen kulutettu levytila saadaan esiin. PostgreSQL:ssa levylohkon koko on yleisesti 8 kilotavua [24].

6.3 Kyselyiden suorittamistavan analysointityökalut

Elmasrin [6, s. 556] mukaan tietokannanhallintajärjestelmissä on työkaluja **kyselyiden suorittamistavan** selvittämiseksi. PostgreSQL:ssä [22] lisäämällä `EXPLAIN`-komento `SELECT`-lauseen eteen järjestelmä esittää, millä tavalla **kyselyoptimoija** (engl. *optimizer* tai *query planner*) arvelee SQL-lauseen tehokkaimmin suoritettavan. Käyttämällä `EXPLAIN ANALYZE` -muotoa annettu SQL-lause myös suoritetaan. Kyseistä arviota tehokkaimmasta kyselyn kokonaissuorittamistavasta kutsutaan suoritussuunnitelmaksi (engl. *execution plan*), joka koostuu useista yksittäisten tietokantaoperaatioiden saantipoluista (engl. *access path*). Liitteessä D tarkastellaan muutamia `EXPLAIN ANALYZE` -muodossa suoritettuja kyselyitä.

Kyselyn muodossa `EXPLAIN` tai `EXPLAIN ANALYZE` **suorittamisen jälkeen PostgreSQL esittää tehokkaimmaksi arvioidun ja käytetyn suorittamistavan**, kunkin yksittäisen operaation suorittamiseen kuluneen tarkan suoritusajan ja suorituksessa hyödynnettyjen levyoperaatioiden määrät sekä suorituksen aikana hyödynnetyt SQL-lauseen vertailuehdot. Tulosteessa ovat mukana lisäksi kyselyn suorittamisen aikana hyödynnetyt indeksit ja muut käytettävissä olevat rajoitteet. Liitteessä D kuvataan analysointitulosteen sisältö.

Edellä mainituilla työkaluilla voidaan selvittää mahdollisia pullonkauloja tietokannan hyödyntämisessä ja toiminnassa. Haittana on se, että **tarkka analysointi tulee tehdä aina yksittäinen SQL-lause kerrallaan**. Lisäksi suoritettu analysointi edustaa PostgreSQL:n dokumentaation [18] mukaan aina vain kyseistä tietokantaa ja sen sisältämää dataa. Kyselyn analysoinnin tuloksia siis ei välttämättä voida yleistää mihinkään toiseen samanlaiseen tietokantaan tai erilaiseen dataan. Mullinsin [11, s. 323] mukaan kehitettäessä tietokantaa hyödyntävää järjestelmää tulisi pyrkiä siihen, että myös **testauskäyttöön tuodaan dataa tuotantotietokannasta**. Muussa tapauksessa hänen mukaansa saattaa eri tavalla suoritettujen kyselyiden takia ilmetä suorituskykyongelmia siinä vaiheessa, kun järjestelmä otetaan kehitysvaiheen jälkeen tuotantokäyttöön.

PostgreSQL:n dokumentaatio [22] ja Mullins [11, s. 322] korostavat sitä, että kyselyoptimoijalla tulee olla käytössään **ajantasainen tilasto** kyselyssä tarvittavien taulujen datan jakautumisesta (engl. *statistics about the distribution of data within the*

table). Muussa tapauksessa kyselyn suorittamistavan optimointi perustuu puutteellisiin tietoihin, ja `EXPLAIN`-komennon esittämä suorittamistapa voi olla täysin virheellinen ja tehoton. PostgreSQL:ssa taulujen datasta muodostetaan ajantasaiset tilastot antamalla tietokannanhallintajärjestelmälle `ANALYZE`-komento. Taulujen datan analysointi voidaan suorittaa myös tietokannassa olevien turhien sisäisten tietojen poistamisen yhteydessä `VACUUM ANALYZE` -komennolla.

Kyseiset analysointityökalut saattavat tarjota vihjeitä siihen, että tiettyjä indeksejä mahdollisesti kannattaa luoda tietokantaan. Esimerkiksi `EXPLAIN`-komennon tulosteessa esiintyvä termi `seq scan` eli **peräkkäishaku** (engl. *sequential scan*) tarkoittaa sitä, että taulun sisältö käydään läpi kokonaisuudessaan alusta loppuun. Jokainen taulun rivi siis joudutaan lukemaan yksi kerrallaan ja vertailemaan rivin sarakkeiden arvoja suoritettavan kyselyn `WHERE`-ehtoihin. Tämä on yleensä varsin hidasta, koska mitään nopeuttavia keinoja tarkasteluun ei ole tarjolla. Puhelinluetteloa voidaan tietyllä tavalla verrata edellä olevaan ongelmaan. Tietyn nimen etsiminen luettelosta on hidasta, mikäli aina joudutaan selaamaan luettelo alusta lähtien läpi.

`EXPLAIN`-tulosteessa oleva `index scan` -viittaus tarkoittaa sitä, että jotain tiettyä tietokannan tauluun luotua **indeksiä on voitu hyödyntää** annetun kyselyn suorituksessa. Jos `EXPLAIN`-tuloste näyttää kyselylle jonkin `seq scan` -rivin, on mahdollista, että luomalla yksi tai useampia indeksejä kyselyn käyttämiin tauluihin autetaan tietokantaa poistamaan peräkkäishaut tästä nimenomaisesta kyselystä ja näinollen tehostamaan kyselyn suorittamista. Tarvittavien ja kyselyiden suorittamista tehostavien indeksien löytäminen onnistuu kuitenkin valitettavasti yleensä vain yleisten nyrkkisääntöjen (ks. luku 5.7) ja tietokantoihin liittyvän kokemuksen tuoman tietämyksen avulla. Yleensä tarvitaan kyselyn monimutkaisuudesta riippuen myös runsaasti erilaisia ratkaisuja hyödyntäviä versioita ja kokeiluja, jotta paras versio kyselystä ja suorittamistavasta on löydettävissä.

Toisinaan Mullinsin [11, s. 327] mukaan tietokannanhallintajärjestelmän kyselyoptimoija päättelee, että olemassa olevia indeksejä ei kannata käyttää, koska niiden tuoma hyöty olisi olematon tai indeksin käyttö aiheuttaisi jopa ylimääräisiä levyoperaatioita. **Indeksien luominen tauluihin ei nimittäin pakota tietokannanhallintajärjestelmää käyttämään indeksejä.** Indeksit ovat vain tarjolle asetettuja nopeutusapuja, joita tietokannanhallintajärjestelmä käyttää tarvittaessa. Esimerkiksi PostgreSQL:n konfiguraatiotiedostossa [25] voidaan vaikuttaa varsin vapaasti siihen, milloin indeksejä tulisi hyödyntää ja milloin ei.

6.4 Keskittyminen eniten suoritettaviin kyselyihin

Elmasri toteaa kirjassaan [6, s. 552], että tietokantojenkin käytön yhteydessä pätee yleensä ns. 80/20-sääntö. Kyseistä sääntöä kutsutaan Mullinsin [11, s. 252] mukaan joskus myös Pareton periaatteeksi. Elmasrin mukaan sääntö tarkoittaa sitä, että suuressa määrässä tietokantaoperaatioita viidesosa sovelluksen sisältämistä kyselyistä ja päivityksistä tuottaa 80 prosenttia kaikista tietokantaan kohdistuvista operaatioista. Täten ei tarvita perusteellista analyysiä kaikista tietokantaan kohdistuvista operaatioista, vaan **riittää seurata suoritetuista kyselyistä yleisimpiä**. Mullins väittää kirjassaan [11, s. 252], että todennäköisimmin suurin osa suorituskykyongelmista johtuu tehottomasta SQL-kielen ja lähdekoodin hyödyntämisestä.

Esimerkiksi Korppi-järjestelmässä erään vuorokauden aikana lukumääräisesti 20 eniten käytettyä kyselyä suoritettiin yhteensä runsaat miljoona kertaa, kun yhteensä kyseisen vuorokauden aikana suoritettiin noin 2.5 miljoonaa kyselyä. Tietokantaan kohdistuva kuormitus aiheutui siten 40-prosenttisesti näistä useimmin suoritetuista rakenteeltaan samoista kyselyistä. Toki on huomattava se, että lähes aina tietokantakyselyissä tietyt käyttäjästä tai ympäristöstä johtuvat kyselyparametrit vaihtuvat, joten kyselyt eivät ole täsmälleen samoja. Suorituskyvyn seurannan tulee ottaa tämä huomioon, jotta tulokset olisivat järkeviä ja vertailukelpoisia. Kuitenkin esimerkiksi taulujen kentissä olevien indeksien kannalta ei ole mitään väliä sillä, mitkä ovat kyselyiden muuttuvien parametrien arvot.

6.5 Lokitiedostojen tarkastelu Korppi-järjestelmässä

Tietojärjestelmien sovellusten ja palvelinohjelmistojen kirjoittamat erityyppiset lokitiedostot ovat oiva apu tutkittaessa ongelmia, mahdollisia pullonkauloja ja tehottomuutta tietokantoihin pohjautuvien järjestelmien toiminnassa. Myös Korppi-järjestelmä kirjoittaa omaa lokiaan. Jokaisesta Korppi-järjestelmän suorittamasta sivupyynnöstä järjestelmä kirjaa lokiin tiettyjä toimenpiteitä ja niihin liittyviä tietoja, jolloin järjestelmän ja käyttäjien virheiden etsintä ja myös mahdollisten väärinkäytösten selvittäminen jälkikäteen on mahdollista. Lokiin kirjoitetaan muun muassa Korppi-järjestelmän JSP-sivujen avaukset ja sivujen suoritusten valmistumiset sekä suoritetut tietokantakyselyt ja tietokannan tauluissa olevien kenttien arvojen päivitykset. SQL-kyselylauseista ja -päivityslauseista tallennetaan ainoastaan suoritettu lause sekä toiminnon suoritusaika, ei saatua tulosjoukkoa.

Korppi-järjestelmässä hyödynnetyt Apache- ja Tomcat-sovellukset kirjoittavat omia lokejaan. PostgreSQL-tietokannanhallintajärjestelmäkin voidaan käskeä kirjoittamaan kaikki suoritetut SQL-lauseet ja -päivitykset omaan lokiinsa tai Unix-järjestelmien yleisesti käyttämällä `syslog`-tekniikalla lokipalvelimelle [25]. Kyseisiä PostgreSQL:n tarjoamia tapoja ei ole käytetty Korppi-järjestelmässä, koska lokitettavasta tiedosta katoaisi silloin oleellista informaatiota, kuten toiminnon suorittavan käyttäjän tiedot, Korppi-istunnon tunnus ja avatun Korppi-sivun tunnus. Tällöin nykyisen informaation esillesaamiseksi tulisi hyödyntää useiden eri lokitiedostojen tietoja.

Korppi-järjestelmän lokin hyödyntäminen ja sen tarjoamat mahdollisuudet käyttäjien toimenpiteiden seurantaan eivät tietenkään tarkoita sitä, että vaikkapa tietoturvallisuus tulisi unohtaa. Loki tarjoaa sellaisenaan vain mahdollisuuden huomata virheet ja tarkasteltavat asiat jälkikäteen, jolloin samojen virheiden esiintyminen myöhemmin voidaan estää lähdekoodia korjaamalla.

6.6 Esimerkki Korppi-järjestelmän lokitiedostosta

Jokaisesta Korppi-järjestelmän suorittamasta tietokantakyselystä tallennetaan käytetty SQL-lause, kyselyn aloitusaika ja kyselyn tietokantapalvelimella viemä aika sekä viittaus käyttäjään, jonka suorittama toiminto Korppi-järjestelmässä tietokantakyselyn aiheutti. Samaten tietokannan päivityksistä tallentuvat SQL-päivityslause, päivityksen aloitushetki ja päivityksen kestoaika. Näiden tietojen avulla yksittäisten SQL-lauseiden viemiä aikoja on helppo seurata myöhemmin.

Seuraavassa Korppi-järjestelmän tuottamasta lokitiedostosta poimitussa esimerkissä avataan tietokantayhteys, suoritetaan yksi kysely (ks. luku 5.3) ja yksi päivityslause, sekä lopuksi suljetaan tietokantayhteys:

```
04.02.02 12:00:01.542 user 0 'dummy' opened page
 /kotka/portal/showLogout.jsp?, IP: 130.234.xx.xx
04.02.02 12:00:01.550 nr: 1 0 id: 1343526 0
 * new User.getOption(String, String) db *
04.02.02 12:00:01.550 nr: 1 1 id: 1343526 0 * CONN * JDBC pooling
04.02.02 12:00:01.558 nr: 2 0 id: 1343526 0 * CS ** diff: 8
04.02.02 12:00:01.558 nr: 2 0 id: 1343526 1 QUERY START:
 SELECT s.name, ps.value FROM personsetup AS ps, setup AS s
 WHERE ps.personid=0 AND ps.setupid=s.setupid
 AND ps.deleted=FALSE AND s.deleted=FALSE;
04.02.02 12:00:01.561 nr: 2 0 id: 1343526 1 QUERY END qtime: 3
```

```
04.02.02 12:00:01.562 nr: 2 0 id: 1343526 2 UPDATE:
 UPDATE personsetup SET modifiedon=current_timestamp, value='5'
 WHERE personid=0 AND setupid=23;
04.02.02 12:00:01.563 nr: 1 0 id: 1343526 2 UPDATE END,
 updated 1 rows, utime: 1
04.02.02 12:00:01.563 nr: 1 0 id: 1343526 2 * D *, ctime: 3
```

Em. esimerkissä pitkät rivit on tulostusta varten katkaistu ja sisennetty useammalle riville. Lisäksi jokaisen rivin lopussa olevat tietojen yhdistämistä helpottavat merkinnät `th: Thread-9| p149464 39: s1203128468 u0` on poistettu. Kyseiset tiedot kertovat JSP-sivun suorittaneen käyttöjärjestelmän säikeen eli Tomcat-sovelluspalvelimen prosessin tunnisteen (`Thread-9|`), sivun järjestysnumeron Tomcat-sovelluspalvelimen käynnistyksestä lähtien (`p149464`), käyttäjän istunnon (engl. *session*) tunnisteen (`s1203128468`) sekä käyttäjän yksilöivän tunnistenumeron eli `personid`-arvon (`u0`).

Jokaisen lokitiedostorivin tai -rivikokoelman alussa on aikaleima muodossa `vv.kk.pp hh:mm:ss.sss`. Tietokantaan kohdistuvissa operaatioissa käytetään seuraavissa kappaleissa kuvattuja merkintöjä.

`nr`-merkinnän perässä oleva ensimmäinen luku ilmoittaa **auki olevien tietokantayhteyksien määrän** kyseisenä lokirivin ajanhetkenä, sekä jälkimmäinen luku ilmaisee aukeamista odottavien tietokantayhteyksien määrän.

`id`-merkinnän perässä olevan ensimmäisen luvun avulla esitetään **tietokantayhteyden järjestysnumero** Korppi-järjestelmän Tomcat-sovelluspalvelimen käynnistymisestä lähtien. Järjestysnumeroa seuraava luku on tietokantayhteyden avulla **suoritetun operaation sisäinen järjestysnumero**. Jokainen SQL-kysely ja -päivityslause saman tietokantayhteyden sisällä kasvattaa jälkimmäistä lukua yhdellä, jotta eri tietokantaoperaatiot voidaan erottaa toisistaan yksikäsitteisesti samankin tietokantayhteyden sisällä.

Asteriskien eli `*`-merkkien ympäröimänä esitetään **tietokantayhteyden avaamiseen ja sulkemiseen liittyvät seuraavat toimenpiteet**:

`new db` ilmaisee, että uusi Korppi-järjestelmän tietokantaolio eli `DB`-luokan (engl. *database*) ilmentymä on luotu Javassa. Tietokantaoliolle on mahdollisesti annettu tunniste, jonka avulla lähdekoodin oikea kohta on helpommin löydettävissä.

`CONN` tarkoittaa, että edellä luotu `DB`-olio pyytää tietokantayhteyttä (engl. *connection*). `JDBC pooling` -merkintä ilmaisee, että käytössä on

JDBC-ajurin (engl. *Java Database Connectivity Driver*) tarjoama standardi tapa käyttää SQL-tietokantoja. Luku 7.3 kuvaa tarkemmin JDBC:n käyttöä.

CS tarkoittaa sitä, että tietokantayhteys on saatu avattua (engl. *connection success*). Rivillä esiintyvä `diff`-merkintä (engl. *difference*) ilmaisee, kauanko aikaa millisekunteina kului yhteydenluomisen aloittamisesta kyseiseen lokirivin ajanhetkeen eli siihen, kun tietokantayhteys on valmiina käyttöön.

QUERY START -merkinnällä tietokantakyselyn suorittaminen on aloitettu.

QUERY END -merkintään mennessä tietokantapalvelin on saanut suoritettua kyselyn sekä palauttanut tuloksen kysyjälle. Tällöin tietokantakyselyn viemä ajallinen kesto on tulostettavissa lokitiedostoon, jolloin lokeja analysoimalla voidaan nopeasti selvittää hitaat kyselyt.

UPDATE -merkinnällä aloitetaan päivityslauseen suorittaminen.

UPDATE END -merkintä päättää päivityslauseen suorittamisen. Tällöin tietokannan sisältämää dataa on jollakin tavalla muutettu, ja muutettujen rivien lukumäärä ilmaistaan lokirivillä. Lisäksi lokirivin lopussa ilmaistaan päivityksen kestoaika millisekunteina.

D -merkintä kirjataan lokiin siinä vaiheessa, kun tietokantayhteys suljetaan (engl. *disconnect*). Jos yhteyttä ei suljeta, se suljetaan pakotetusti silloin, kun `DB`-olio Javan roskienkeräyksen (engl. *garbage collection*) johdosta kuolee. Kustakin sulkemattomasta yhteydestä tulee virhemerkintä lokiin, jolloin ongelmat voidaan korjata lähdekoodiin. Mikäli tietokantayhteyttä yritetään sulkea useaan kertaan, kirjaa järjestelmä lokiin virheilmoituksen.

7 SQL-kyselylauseiden nopeuttaminen

Luvussa tarkastellaan niitä keinoja, joilla SQL-kyselyihin ja muihin sovelluksen lähdekoodiin tehtävillä muutoksilla saadaan tehostettua sovelluksen käyttöä. Luku perustuu pääosin lähteisiin [7, s. 246–247] ja [11, s. 344–349].

Korppi-opintotietojärjestelmän kehittäjien kohtuullisen vähäinen optimointiosaaminen ja pidemmän aikavälin suuntaviivojen puutteellinen kohdentaminen ovat johtaneet siihen, että kehitystyössä kiireisellä aikataululla toteutetut ratkaisut eivät aina ole aikaa myöten pysyneet tehokkaina. Korppi-järjestelmän tietokantaan tallennetun datan määrä on käyttäjien ja järjestelmän laajentuneen hyödyntämisen myötä lisääntynyt valtavasti vuosien varrella, joten järjestelmän toteutusratkaisuja on ajan saatossa jouduttu optimoimaan jouhevan käytön turvaamiseksi.

7.1 Nyrkkisääntöjä SQL-kyselylauseiden nopeuttamiseen

Mullins [11, s. 344–349] ja Hoffer [7, s. 246–247] mainitsevat nyrkkisääntöjä, joiden avulla ohjelmoijan on yleensä mahdollista tehostaa tietokantahakuja ja SQL:n käyttöä. Jokainen tietokantojen suorituskykyä koskeva kysymys Mullinsin [11, s. 344] mukaan **riippuu muista asioista** (engl. *it depends*). Tietokannanhallintajärjestelmän hoitajan tulee tietää, mistä suorituskykyyn liittyvä kysymys tai ongelma riippuu. Mullins kehottaa kirjassaan [11, s. 344] **varomaan** sellaisia ohjeiden ja dokumenttien tehostamisvinkkejä, joissa käytetään **määritteitä ”aina” tai ”ei koskaan”** (engl. *”always” and ”never”*). Hänen mukaansa lähes kaikki asiat riippuvat toisista asioista.

Mullins mainitsee kirjassaan [11, s. 344], että **kyselyn alkioiden järjestyksellä saattaa olla merkitystä** kyselyn suorituskyvyn kannalta. Hän kehottaa sijoittamaan kaikkein rajoittavimman ehdon kyselyssä siten, että kyselyoptimoija voi hyödyntää sitä ensin. Tällöin jo ensimmäisen kyselyehdon hyödyntämisen jälkeen on haun välitulos mahdollisimman pieni, eikä edelleen vaikkapa koko taulun sisältö tai useiden taulujen sisältöjen karteesinen tulo (ks. luku 5.3).

Mullinsin [11, s. 344] mukaan kaikenlaisten IT-toimintojen perusnyrkkisääntö on noudattaa KISS-periaatetta (engl. *Keep it simple, stupid*). Hänen mukaansa **yksinkertaisina pidettävät SQL-kyselyt** auttavat pitämään kehitys- ja ylläpitotehtävätkin

yksinkertaisina. Yksinkertaisia SQL-lauseita ja niiden toimintaa on helpompi tulkita ja muokata. Myös Hoffer [7, s. 246] on samaa mieltä yksinkertaisista SQL-kyselyistä. Hän kehottaa jopa **jakamaan isoja ja monimutkaisia kyselyitä pienemmiksi palasiksi**, joista kukin voi hyödyntää paremmin indeksejä. Kyselyn lopputulos voidaan koostaa yhteen vaikkapa `UNION`-operaation avulla (ks. luku 5.4).

Toisaalta kuitenkin Mullinsin [11, s. 345] mukaan **monipuolisempi SQL-kielen hyödyntäminen** voi olla huomattavasti tehokkaampaa suorituskyvyn kannalta. Mitä enemmän vastuuta annetaan tietokannanhallintajärjestelmälle ja kyselyoptimoijalle, sitä parempaa suorituskykyä voidaan odottaa. Mullins kertoo esimerkkinä, että jotkut ohjelmoijat välttävät SQL-lauseissa taulujen välisten liitosten (engl. *joins*) käyttämistä. Tällöin kaikki vaadittavat tiedot haetaan yksittäisten lyhyiden SQL-kyselyiden avulla, ja tiedot yhdistellään lopuksi lähdekoodin avulla. SQL-kielen hyödyntäminen on tällöin yksinkertaisempaa, koska ohjelmoijan ei tarvitse tietää, kuinka taulujen tietoja yhdistellään SQL:n avulla. Tietokannanhallintajärjestelmä saattaa kuitenkin olla huomattavasti tehokkaampi tietojen yhdistelyssä, koska vähemmän dataa palautetaan sovelluksen käsiteltäväksi. Lisäksi datan määrän tai tietokannan rakenteen muutosten myötä kyselyoptimoija vaihtaa kyselyn suorittamistapaa automaattisesti, kun taas lähdekoodin toimintaa joutuu ohjelmoija muokkaamaan käsin.

SQL-kyselyn tuloksena tulisi Mullinsin [11, s. 345] mukaan aina palauttaa **pienin mahdollinen joukko välttämättömiä kenttiä**. Tietokantakyselyssä ei siis koskaan tulisi käyttää muotoa `SELECT * FROM henkilö` (ks. luku 5.3), joka hakee kohdetaulusta kaikki kentät. Mitä enemmän kenttiä tietokannanhallintajärjestelmä joutuu käsittelemään ja palauttamaan kyselyn `SELECT`-osion kenttälistan takia, sitä enemmän järjestelmän tulee tehdä töitä tämän toimenpiteen suorittamiseksi. Lisäksi Mullinsin mukaan **palautettavien rivien lukumäärä tulisi minimoida** liittämällä SQL-kyselyyn mukaan kaikki tarvittavat liitosehdot eri taulujen välille. Mitä enemmän palautettavia tietoja voidaan vähentää jo kyselyn suorituksen aikana, sitä tehokkaampi kysely on, koska vähemmän dataa lopulta palautetaan kyselyn suorittajalle. Mullinsin [11, s. 345] mukaan joillakin ohjelmoijilla on tapana jättää kyselyn `WHERE`-lausekkeen ehtoja pois, jotta kysely yksinkertaistuisi. Kuitenkin mitä enemmän tietoa kyselyoptimoijalla on taulujen sisällön suhteen, sitä tehokkaammin kysely voidaan suorittaa.

Mullins kehottaa kirjassaan [11, s. 346] **välttämään karteesisten tulojen käyttämistä** (ks. luku 5.3). Tämä onnistuu siten, että jokaista SQL-kyselyssä hyödynnettyä

taulua kohti on kyselyn hakuehdoissa mukana myös kyseisen taulun liitokset muihin tauluihin. Jos näin ei toimita, saattaa tuloksena olla hyvin tehoton kysely ja jopa vääriä tietoja. Tällöin hakutulokseen saatetaan ottaa mukaan vaikkapa yhdistelmä kaikkien kyselyssä mukana olevien taulujen riveistä. Jos rivien tiedot eivät täsmää (engl. *nonmatching rows*), niitä ei voida hakutuloksesta poistaa, koska kunnollisia ehtoja täsmäävyydelle ei ole annettu.

Mullins ehdottaa kirjassaan [11, s. 346], että SQL-lauseissa kannattaa **`OR`-ehtoja** mahdollisuuksien mukaan kokeilla vaihtaa **`IN`-listojen avulla käytettäviksi ehdoiksi** (ks. luku 5.8). Hänen mukaansa tällöin on todennäköistä, että kyselyn suorituskyky paranee.

Mullinsin [11, s. 347] mukaan **`LIKE`-operaattorin tehottomalla käytöllä** (ks. luku 5.4) saadaan helposti aikaan suorituskykyongelmia. Jos hakuehdossa ei käytetä lainkaan SQL-jokerimerkkejä, ei `LIKE`-operaattoriakaan kannata käyttää, koska tämä saattaa tarpeettomasti hidastaa vertailuja ja siten kyselyn suorittamista. PostgreSQL:n dokumentaation [19] ja Mullinsin [11, s. 347] mukaan indeksejä ei voida myöskään hyödyntää, jos haettava merkkijono ei sijaitse hakuehdon alussa. Esimerkiksi luvun 4.9 taulukkoon 4.5 kohdistuvan SQL-kyselyn

```
SELECT sukunimi, etunimi
FROM henkilö
WHERE sukunimi LIKE '%mone%';
```

suorittaakseen tietokannanhallintajärjestelmä joutuu käymään läpi koko `henkilö`-taulun `sukunimi`-kentän datan, eikä `sukunimi`-kenttään luotua indeksiä voida hyödyntää.

SQL-kieli on joustava, joten se sallii saman hakutuloksen palauttavan kyselyn ohjelmoinnin monella eri tavalla. Mullinsin [11, s. 347] mukaan yleensä kuitenkin **yksi tapa on tehokkaampi kuin muut**. Hän kehottaakin tietokannan valvojaa ja tietokantaohjelmoijaa ylläpitämään tietoutta siitä, mikä tapa on paras mihinkin käyttötarkoitukseen. Lisäksi sama tietous tulisi saattaa myös muiden ohjelmoijien saataville. Hoffer mainitsee kirjassaan [7, s. 247] opiskelun, kyselyiden analysoinnin ja optimointiin liittyvän tietouden lisäämisen yhdeksi tehokeinoksi tehokkaampien kyselyiden tuottamiseksi.

Mullins kehottaa kirjassaan [11, s. 348] tietokantaohjelmoijia kirjoittamaan sovelluksen lähdekoodin siten, että tietokantatapahtumien (engl. *transactions*) aiheuttamat muutokset vahvistetaan (engl. *commit*) mahdollisimman usein, eli että **muutokset kirjoitetaan tietokantaan mahdollisimman aikaisessa vaiheessa**. Tämä mah-

dollistaa Mullinsin mukaan sen, että tietokantaobjekteihin kohdistuvien lukkojen määrä saadaan minimoitua, eikä muutoksia tarvitse säilyttää kovin kauaa isokokoisissa väliaikaisissa peruutuslohkoissa (engl. *rollback segments*).

Mullins kehottaa kirjassaan [11, s. 348] harkitsemaan **tallennettujen proseduurien** (engl. *stored procedures*) käyttöä (ks. luku 4.4). Tällöin verkkoliikenteen määrä vähentyy, koska toiminnon suorittamiseksi annetaan vain yksi lyhyt pyyntö. Proseduurin sisällä tietokantapalvelimella voidaan suorittaa useita SQL-kyselyitä, ja vain haluttu lopputulos tuodaan takaisin kutsujalle. Proseduureissa olevat SQL-kyselyt saattavat suoriutua tehokkaammin, koska tietokannanhallintajärjestelmä voi esikääntää ne jo kauan ennen varsinaista suoritusta.

Mullins suosittelee kirjassaan [11, s. 348] **välttämään aritmeettisia operaatioita SQL:ssa** ja suorittamaan tällaiset laskutehtävät sovelluksen lähdekoodissa SQL:n ulkopuolella. Esimerkiksi `LOWER`-funktio muuntaa parametrina tuodun merkkijonon tai kentän sisällöt pienikirjaimisiksi, jolloin merkkijonojen vertailu onnistuu riippumatta kirjainkoosta (engl. *case-insensitive comparison*). Kirjainkokojen muunnoksiin kuluu aina hieman aikaa. Toisaalta ristiriitaisesti Mullins kuitenkin **suosittelee käyttämään SQL-funktioita vähentämään ohjelmoijan työtä**.

Hofferin [7, s. 246] mukaan **kenttien ja vakioiden välisissä vertailuissa tulee käyttää yhteensopivia tietotyyppejä**, jotta vältytään turhilta datanmuunnoksilta.

Viite-eheyksien tarkistukset tulee Mullinsin [11, s. 348] mukaan aina hoitaa **tietokannan avulla**, eikä sovelluksen lähdekoodissa. Hän suosittelee myös tarkkailemaan ”piilossa olevien” liipaisimien (engl. *trigger*) vaikutusta. Rivien poistaminen taulusta `DELETE`-komennolla saattaa suorittaa monta muuta operaatiota vaikkapa silloin, kun viite-eheyteen on liitetty sääntö `ON DELETE CASCADE`. Tällöin poiston kohde poistetaan myös kaikista viittaavista tauluista, jotka taas voivat aiheuttaa muiden liipaisimien suorittamisen. Ohjelmoija voi kuvitella, että tehottomuusongelma johtuu hitaasta `DELETE`-komennosta, vaikka oikeasti jokin huonosti käyttäytyvä liipaisin onkin varsinainen syyllinen.

Hoffer kehottaa kirjassaan [7, s. 246] sovelluksen ohjelmoijaa tiedostamaan sen, **miten tietokannanhallintajärjestelmä hyödyntää indeksejä** SQL-kyselyn suorittamisen aikana. Hänen mukaansa moni tietokannanhallintajärjestelmä käyttää SQL-kyselyn suorituksen aikana vain yhtä indeksiä taulua kohti. Tällöin on hyödyllistä tietää käytettävät indeksit, jolloin turhat indeksit voidaan poistaa.

Hoffer kehottaa kirjassaan [7, s. 246] **välttämään alikyselyitä**. Hänen mukaansa alikyselyitä käyttävä SQL-kysely on yleensä aina tehottomampi kuin kysely, jossa ei

alikyselyitä käytetä. Luvun 8.2 esimerkki havainnollistaa tätä.

Hoffer suosittelee kirjassaan [7, s. 246] **välttämään taulun yhdistämistä itsensä kanssa** (engl. *self-join*). Hänen mukaansa on tehokkaampaa ensin tehdä taulusta kopio, jota käytetään kyselyssä alkuperäisen taulun ohella. Hän kehottaa tallentamaan myös **usein haetut samat tiedot väliaikaistauluihin**, jotta niitä ei tarvitse hakea aina uudelleen jokaista käyttökertaa varten. Tosin haittapuolena tässä on hänen mukaansa se, että väliaikaistaulujen sisältö ei muutu, jos alkuperäiset lähtötiedot muuttuvat kyselyn suorituksen aikana.

Hofferin [7, s. 246] mukaan on hyödyllistä **niputtaa useita päivitys-** eli `UPDATE`-**lauseita yhteen**, jotta tietokannanhallintajärjestelmä voi mahdollisesti suorittaa päivityslauseet rinnakkain. Myös lauseiden käsittelyyn vaadittava yleisrasite (engl. *query processing overhead*) vähentyy.

Hoffer mainitsee kirjassaan [7, s. 247], että aina tulee huomioida myös **ennakoimattomien kyselyiden käsittelyyn ja suorittamiseen vaadittu kokonaisaika** (engl. *total query processing time for ad hoc queries*). Tähän kokonaisaikaan lasketaan kyselyn suorittamisajan lisäksi se aika, joka ohjelmoijalta tai peruskäyttäjältä kuluu SQL-kyselyn kirjoittamiseen. Hofferin mukaan monesti on hyödyllisempää kyseisissä kyselytarpeissa antaa tietokannanhallintajärjestelmälle enemmän vastuuta kyselyn suorittamiseksi kohtuullisen tehokkaalla ratkaisulla, jotta SQL-kyselyn tekijä saa kyselyn nopeammin kirjoitettua. Hänen mukaansa **aina ei kannata kuluttaa ohjelmoijana liikaa aikaa** siihen, että kysely suoriutuisi tehtävästään mahdollisimman nopeasti ja tehokkaasti. Hoffer ehdottaakin, että tällaisia väliaikaisia, tiettyjä tarpeita varten kirjoitettuja kyselyitä tulisi mahdollisuuksien mukaan suorittaa silloin, kun tietokantaan kohdistuu vain vähäinen kuormitus.

7.2 Korppi-järjestelmässä tietojen haku kokonaisuutena tai osissa

Luvussa 7.1 mainittiin Mullinsin ja Hofferin nyrkkisäännöt yksinkertaisina pidettävistä SQL-lauseista ja toisaalta monimutkaisemmasta SQL:n käytöstä, joka saattaa olla tehokkaampaa suorituskyvyn kannalta. Sovelluksen lähdekoodia toteuttaessaan ohjelmoija joutuukin monesti pohtimaan, kannattaako tiedot hakea tietokannasta paloissa pienillä, lyhyillä ja tehokkailla SQL-kyselyillä vai tulisiko koko haluttu lopputulos noutaa monimutkaisemmalla isommalla kyselyllä. Toteutustavan valinta riippuu luvun 7.1 tapaan hyvin paljon tilanteesta, joten yksiselitteistä ja kaikkialla toimivaa nyrkkisääntöä on vaikea antaa.

Korppi-järjestelmän elinkaaren ensimmäisinä vuosina pyrittiin tietokantahakuja kirjoittamaan isojen, kaikki tiedot kerralla hakevien kyselyiden avulla. Tällä tavalla kaikki tarvittavat tiedot saatiin käyttöön helposti ja silloisilla datamäärillä myös nopeasti, eikä lisäksi tarvittu muita tietorakenteita.

Datamäärien kasvaessa huomattiin, ettei isojen tietokantahakujen hyödyntäminen olekaan enää tehokkain ja ylläpidettävin tapa toimia. Ensinnäkin tietokannan tauluissa on **dataa huomattavasti enemmän kuin kyselyn alkuperäisenä kirjoitushetkenä**. Alunperin ehkä nopeasti kiireessä ja sen kummemmin miettimättä kirjoitettu SQL-kysely toimi nopeasti, koska kussakin taulussa oli maksimissaan muutamia satoja tai tuhansia rivejä. Tuolloin tavoitteena oli siis vain jonkinlainen toimivuus, eikä nopein mahdollinen toteutus kyselystä tai tietojen hakutekniikoista. Sittemmin rivien määrä on saattanut sata- tai tuhatkertaistua, jolloin SQL-kyselyn tehokkuuteen on ollut pakko kiinnittää huomiota. Käytännössä minkään Korppi-tietokantataulun rivimäärä ei ajan kuluessa ainakaan pienene, vaan suurin osa tauluista jatkuvasti kasvaa rivimäärältään.

Toinen syy mainitunlaisista isoista ja monimutkaisista kyselyistä luopumiseen on se, että lähes poikkeuksetta tällaiset kyselyt ovat ajan saatossa toteutettujen kenttiin ja hakuehtoihin kohdistuneiden pienten muokkausten ja lisäysten seurauksena palauttaneet hakutuloksenaan **alkuperäiseen kyselyyn nähden useampia kenttiä**. Useimmiten myös **haussa tarvittavien taulujen määrä on tarpeiden lisääntyessä kasvanut**. Erityisesti käsiteltävien taulujen määrän lisääntyminen tarkoittaa lisääntynyttä datamäärää sekä SQL-kyselyn suorittamisen aikana että tuloksen käsittelyvaiheessa. Näistä kumpikin vaihe vähentää omalta osaltaan kokonaistehokkuutta.

Pienten osakyselyiden suoritusajat ja mahdollisesti myös niiden vaatimat muistiresurssit ovat yleensä vähäisempiä yhteen isoon erillisistä osakyselyistä koostuvaan SQL-kyselyyn nähden. Tietokantakyselyn muokkaus tehokkaammaksi onkin turhaa, jos aikaa kuluu enemmän muokatun kyselyn suorittamiseen. Luonnollisesti datan määrän jatkuva lisääntyminen saattaa vaikuttaa oleellisesti siihen, että uudempi, mutta kyseisellä hetkellä hieman hitaampi, versio kyselystä on jatkossa nopeampi. Yksittäistä kyselyä on toisinaan vaikea optimoida siten, että jokin versio kyselystä olisi lopullinen ja ”kiveen hakattu”. Jatkossa kenttien, taulujen ja datan määrä luultavasti muuttuu, joten kyselyä tulee silloin optimoida uudelleen tilanteen mukaan. Myös tietokannanhallintajärjestelmän versionvaihdoksen yhteydessä tulevat sisäiset muutokset voivat vaikuttaa oleellisesti hakujen nopeuteen.

Luvussa 7.4 esiteltävät Korppi-järjestelmän teknisiin ratkaisuihin kuuluvat `RS2`-

ja `RS2s`-luokat sisältävät käytännöllisiä apuvälineitä isojen kyselyiden pilkkomiseen pienemmiksi palasiksi.

7.3 SQL-kyselyiden hakutulosten hyödyntäminen Java-sovelluksesta

Java-kielessä hyödynnetään tietokantoja JDBC-rajapinnan avulla [33]. JDBC-rajapintakutsuilla **luodaan yhteys tietokantaan** tai mihin tahansa taulukkona saatavilla olevaan tietolähteeseen (engl. *tabular data source*), **lähetetään SQL-kyselyitä ja SQL-päivityslauseita** sekä **käsitellään SQL-kyselyiden hakutulokset**.

Kun käyttöön on asennettu PostgreSQL-tietokantojen hyödyntämiseen tarkoitettu JDBC-ajuri, voidaan Java-kielen JDBC-rajapinnan avulla hakea SQL-kyselyn hakutulos sovelluksen käyttöön seuraavasti:

```
import java.sql.*;

try {
  Class.forName("org.postgresql.Driver");
}
catch ( ClassNotFoundException e ) {
  throw e;
}

String dbUrl = "jdbc:postgresql://dbserver.host.com/kanta";
String dbUserName = "testuser";
String dbPassword = "testpwd";

Connection conn = null;

try {
  conn = DriverManager.getConnection(dbUrl, dbUserName, dbPassword);
  Statement stmt = conn.createStatement();

  ResultSet rs = stmt.executeQuery(
     " SELECT henkilöID, sukunimi, etunimet "
   + " FROM henkilöt "
   + " ORDER BY sukunimi, etunimet;");

  if ( rs != null ) {
```

```
    rs.beforeFirst();                                                    27
    while (rs.next()) {                                                  28
       int henkilöID = rs.getInt("henkilöID");                           29
       String suku = rs.getString("sukunimi");                           30
       String etu = rs.getString("etunimet");                            31
       System.out.println("HenkilöID "+henkilöID+": "+suku+" "+etu);      32
    }                                                                    33
                                                                         34
    rs.close();                                                          35
   }                                                                     36
                                                                         37
} catch (java.sql.SQLException e) {                                      38
                                                                         39
   System.out.println("Tietokannan käytössä tapahtui virhe: "          40
    + e.getMessage());                                                   41
   throw e;                                                              42
                                                                         43
} finally {                                                              44
                                                                         45
   if (stmt != null) stmt.close();                                       46
   if (conn != null) conn.close();                                       47
                                                                         48
}                                                                        49
```

Em. esimerkissä rivillä 4 ladataan PostgreSQL:n JDBC-ajuri käyttöön. Mikäli ajuria ei löydy, heitetään poikkeus eteenpäin ja poistutaan sovelluksesta. Riveillä 10–12 annetaan tietokantayhteyteen vaadittavat seuraavassa kuvatut parametrit. `dbserver.host.com` on tietokantapalvelimen osoite ja `kanta` on kyseisellä tietokantapalvelimella olevan tietokannan nimi, johon yhteys halutaan luoda. Tietokantayhteyttä halutaan käyttää tietokantaan luodulla `testuser`-nimisellä käyttäjätunnuksella, jonka salasana on `testpwd`.

Rivillä 17 avataan yhteys tietokantaan edellä annetuin parametrein lohkon `try-catch-finally` sisällä, jotta tietokantayhteyden avaus, virheenkäsittely ja luotettava sulkeminen voidaan taata. Rivillä 18 luodaan "lausekeolio", jonka avulla varsinaiset SQL-kyselyt ja SQL-päivitykset voidaan suorittaa. Riveillä 20–23 suoritetaan SQL-kysely, jonka hakutulos palautetaan `java.sql.ResultSet`-rajapinnan mukaisena oliona. Tämän jälkeen riveillä 25–36 käydään kyseinen hakutulos läpi alusta loppuun `while`-silmukan avulla. Luodut oliot `Statement` ja `Connection` on syytä sulkea mahdollisimman pian käytön jälkeen, jotta tietokantayhteys ei jää

varatuksi. Kyseiset oliot ja niiden käyttämät tietokantaresurssit suljetaan `finally`-lohkossa riveillä 46 ja 47.

Luokan `Statement` toteuttava olio palauttaa rajapinnan `ResultSet` toteuttavia olioita. Kyseisentyyppisiä olioita käytettäessä on tietokantayhteyden pysyttävä auki koko hakutuloksen hyödyntämisen ajan, joka toisinaan kokonaisuutena rasittaa tietojärjestelmiä liikaa. Palvelimilla tietokantayhteyksiä nimittäin sallitaan yleensä rajallinen määrä, jotta ruuhka-aikoina tietokantoja hyödyntävä järjestelmä ei tukkeutuisi sadoista yhtäaikaisista pyynnöistä.

Eidemillerin [5] mukaan JDBC-rajapinnassa on ollut versiosta 2.0 lähtien määriteltynä rajapinta `javax.sql.rowset.CachedRowSet`, jolla **tietokantakyselyn tulos saadaan pidettyä muistissa ilman auki olevaa tietokantayhteyttä**. Kyseinen `CachedRowSet`-rajapinta perii tavanomaisemmin käytetyn `ResultSet`-rajapinnan, joten kaikki `ResultSet`in toteuttavat oliot voidaan yleensä suoraan vaihtaa käyttämään `CachedRowSet`-rajapintaa.

JDBC-rajapinnan version 2.0 lopullinen määritys on joulukuulta 1998 [34]. Eidemillerin mukaan syyskuussa 2004 esitellystä Javan versiosta 5.0 alkaen on sisältynyt kyseisen `CachedRowSet`-rajapinnan toteuttava yleinen viitetoteutus (engl. *generic reference implementation*), jossa on mukana muun muassa `CachedRowSet`-rajapinnan toteuttava luokka `com.sun.rowset.CachedRowSetImpl`. Eidemiller mainitsee artikkelissaan [5], että kyseinen viitetoteutus saadaan käyttöön myös vanhemmassa Javan 1.4.2-versiossa lataamalla Sun Developer Networkin WWW-sivuilta toiminnallisuuden tarjoava JAR-paketti.

Käytettäessä `CachedRowSet`-rajapinnan mukaisia olioita voidaan edellisessä Java-esimerkissä siirtää SQL-kyselylauseen käsittely lohkon `try-catch-finally` ulkopuolelle, koska tietokantayhteyden ei tarvitse olla auki hakutulosta käsiteltäessä. SQL-kyselylauseen suorittaminen ja käsittely muuttuvat seuraavanlaiseksi riviltä 14 alkaen:

```
...                                                                    13
Connection conn = null;                                                14
com.sun.rowset.CachedRowSetImpl rs =                                   15
  new com.sun.rowset.CachedRowSetImpl();                               16
                                                                       17
try {                                                                  18
  conn = DriverManager.getConnection(dbUrl, dbUserName, dbPassword);   19
  Statement stmt = conn.createStatement();                             20
                                                                       21
  rs.populate(stmt.executeQuery(                                       22
```

```
      " SELECT henkilöID, sukunimi, etunimet "                          23
    + " FROM henkilöt "                                                 24
    + " ORDER BY sukunimi, etunimet;"));                                25
                                                                        26
} catch (java.sql.SQLException e) {                                     27
                                                                        28
  System.out.println("Tietokannan käytössä tapahtui virhe: "          29
    + e.getMessage());                                                  30
  throw e;                                                              31
                                                                        32
} finally {                                                             33
                                                                        34
  if (stmt != null) stmt.close();                                       35
  if (conn != null) conn.close();                                       36
                                                                        37
}                                                                       38
                                                                        39
if ( rs != null ) {                                                     40
                                                                        41
  rs.beforeFirst();                                                     42
  while (rs.next()) {                                                   43
    ...                                                                 44
  }                                                                     45
                                                                        46
  rs.close();                                                           47
}                                                                       48
```

Em. esimerkissä SQL-kyselyn hakutulos käsitellään riveillä 40–48 samaan tapaan kuin aiemmassa luvun esimerkissä riveillä 25–36. Etuna jälkimmäisessä ratkaisussa on se, että hakutulosta voidaan nyt käsitellä tietokantayhteyden ollessa suljettuna. Hakutulos voidaan vaikkapa sellaisenaan palauttaa kutsuvalle metodille.

7.4 Korppi-järjestelmän RS2-niminen SQL-kyselyiden hakutulosten välimuisti

Korppi-järjestelmään toteutettiin kesällä 2002 `CachedRowSet`-rajapintaa (ks. luku 7.3) vastaava toiminnallisuus `RS2`-nimiseen luokkaan. Sitä voidaan sovelluksen lähdekoodista hyödyntää samoin metodikutsuin kuin `ResultSet`-rajapinnankin (ks. luku 7.3) mukaisia olioita. `ResultSet`-rajapinnan hyödyntäminen ja `RS2`-luokka ovat erittäin oleellisia Korppi-järjestelmän osia, koska järjestelmä perustuu suurimmaksi osaksi tietokannan ja SQL-kyselyiden tulosten hyödyntämiseen.

`RS2`-luokan toiminta-ajatus on sama kuin `CachedRowSet`-rajapinnan toteuttavan luokan, eli `RS2`-olion tietorakenteeseen kopioidaan `ResultSet`-rajapinnan kautta haetut tietokantakyselyn palauttamat kenttätiedot ja kenttien sisällöt. Tällöin `ResultSet` ja tarvittaessa myös tietokantayhteys voidaan sulkea. Muistiin luotu `RS2`-olio toimii siten välimuistina tietokannan ja Korppi-järjestelmän lähdekoodin välillä tarjoten `ResultSet`-rajapintaa vastaavat palvelut. `RS2`-olion tietorakenteeseen haetut tiedot tallennetaan oikeantyyppisinä olioina, kuten kokonaislukumuuttujat Javan `Integer`-tyyppisinä, merkkijonomuuttujat `String`-tyyppisinä ja totuusarvot `Boolean`-tyyppisinä muuttujina.

Alkuperäisessä `RS2`-luokassa `ResultSet`-rajapinnalta haetut tiedot tallennettiin Javan `ArrayList`-tyyppiseen olioon eli listaksi. Käytännön ja testien kautta tietojen hyödyntäminen havaittiin silloisella Javan versiolla kovin hitaaksi. Siksi `RS2`-luokan sisäistä toimintaa muutettiin siten, että se tallentaakin tiedot nopeampaan kaksiulotteiseen `Object`-taulukkoon. Tällöin hakutuloksessa oleva yksittäisen rivin ja kentän arvo saadaan haettua nopeammin ilman suhteellisen hitaita listankäsittelyoperaatioita.

Pian huomattiin, että mainittua tallennustapaa voidaan edelleen osittain tehostaa. Loppuvuodesta 2003 paranneltiin `RS2`-luokkaa siten, että kaikki hakutuloksen kenttien arvot tallennetaankin tietorakenteeseen merkkijonoina eli Javan `String`-tyyppisinä muuttujina. Uudelle luokalle annettiin nimeksi `RS2s` eli ”`RS2` using strings”. Alkuperäinen `RS2`-luokka haluttiin säilyttää uuden luokan rinnalla, koska tietyissä käyttötapauksissa luvussa 7.5 esitettävän tarkastelun perusteella se on tehokkaampi kuin `RS2s`-luokka.

Useimmissa tapauksissa `RS2s`-olion muodostaminen on huomattavasti nopeampaa kuin `RS2`-olion, mutta toisaalta `RS2s`:n joustavuus on heikompi kuin alkuperäisen `RS2`-luokan. Nimittäin `RS2s`-oliosta ei voida suoraan hakea kokonais-

lukua `getInt`-metodilla, vaan muita kuin merkkijonoarvoja varten on jouduttu tekemään vastaavat metodit, jotka hakevat kentän arvon merkkijonona ja tulkitsevat sen oikeantyyppiseksi. Siten luokan metodeista vaikkapa `getInt` jäsentää (engl. *parse*) merkkijonosta kokonaisluvun ja `getBoolean` totuusarvon.

7.5 SQL-kyselyn hakutuloksen käsittelyaikojen vertailu

Taulukossa 7.1 on verrattu SQL-kyselyn hakutuloksen käsittelyn suoritusaikoja `ResultSet`- ja `CachedRowSet`-rajapinnan mukaisten olioiden sekä `RS2`- ja `RS2s`-olioiden avulla. Toimintojen ja hakujen suoritusajat on esitetty millisekunteina, ja ajat ovat usean testiajon mediaaneja.

Rajapinta tai luokka	**Haku**	**Luonti**	**Läpikäynti**		
			Oikein tyypein	**Merkki-jonoina**	**Yhteensä oikein tyypein**
`ResultSet`	1846	-	535	71	2381
`CachedRowSet`	1846	563	122	104	2531
`RS2`	1846	506	93	86	2445
`RS2s`	1846	124	707	23	2677

Taulukko 7.1: SQL-kyselyn hakutuloksen käsittelyn suoritusajat millisekunteina.

Haku-sarakkeen aika tarkoittaa käytännössä `ResultSet`-rajapinnan toteuttavan olion luontia, eli SQL-lauseen suorittamista ja kyselytuloksen palauttamista kutsuvalle metodille. Kyselyn hakutulos tarvitaan jokaista tekniikkaa käytettäessä, joten sama suoritusaika esitetään kyseisessä sarakkeessa jokaisella rivillä. Luonti-sarakkeen suoritusaikaan sisältyvät uuden kyseisen rivin mukaisen olion muistiinluontiaika sekä `ResultSet`-olion kaikkien rivien ja sarakkeiden läpikäyntiaika, jolloin samalla sijoitetaan haettujen kenttien arvot luodun olion tietorakenteeseen oikeaan kohtaan.

Sarakkeessa läpikäynti oikein tyypein kuvataan suoritusaikaa sille, että etukäteen luotu olio käydään kertaalleen läpi hakien jokaiselta hakutuloksen riviltä jokaisen kentän arvo oikeantyyppisenä muuttujana, muun muassa kokonaisluvut `getInt`-metodilla, totuusarvot `getBoolean`-metodilla ja merkkijonot `getString`-metodilla. Sarakkeessa läpikäynti merkkijonoina esitetään vertailun

vuoksi suoritusajat kaikkien rivien kaikkien kenttien arvojen lukemiseksi merkkijonoina `getString`-metodin avulla. Oikein muuttujatyypein haku vastaa parhaiten käytännön tilannetta Korppi-järjestelmässä. Toisinaan kuitenkin riittää tulostaa haettu kyselytulos sellaisenaan pyynnön suorittajalle, vaikkapa joillakin Korppi-järjestelmän raporttisivuilla. Tällöin ei välttämättä ole tarvetta ensin kyselyn tuloksesta selvittää esimerkiksi lukuarvoja kokonaislukuina, koska tiedot voidaan tulostaa myös merkkijonoina.

Oikeanpuoleisimmassa sarakkeessa läpikäynti yhteensä oikein tyypein ilmaistaan kokonaisaika sille, kun SQL-kyselyn tulos on saatu muodostettua muistiin kyseisen rivin tyyppisenä oliona ja käyty kerran kokonaisuudessaan läpi oikein muuttujatyypein. Kyseisen sarakkeen summa on siis sarakkeiden haku, luonti ja läpikäynti oikein tyypein suoritusaikojen summa, joka useimmiten vastaa käytännön tilannetta Korppi-järjestelmässä.

Taulukossa 7.1 Korppi-järjestelmän `syslog`-taulusta haettiin SQL-kyselyllä 10000 riviä, joihin sisältyy `boolean`-, `integer`-, `timestamp`- ja `varchar`-tyyppisiä kenttiä. Tavanomaisessa käytössä kyselyiden tuloksena ei näin monta riviä yleensä saada, joten käytännön tilanteessa joka tapauksessa **SQL-kyselyn suoritusnopeus on oleellisempaa kuin hakutuloksen käsittelynopeus**. Oleellista on myös se, että mainittujen **välimuistiratkaisujen ansiosta tietokantayhteyden ei tarvitse pysyä varattuna** hakutuloksen hyödyntämisen ajan.

`RS2`-olion läpikäynti oikein muuttujatyypein on nopeaa, koska `RS2`-oliossa arvot on tallennettu jo valmiiksi oikein tyypein. Tällöin läpikäyntiaika kuvaa käytännössä sitä, kuinka nopeasti kenttien arvot saadaan muunnettua vaikkapa merkkijonoiksi tai Javan `StringBuffer`-tyyppiseen olioon liittämistä varten. Samaten `RS2s`-olion kenttien arvojen läpikäynti merkkijonoina on todella nopeata muihin läpikäyntiaikoihin verrattuna, koska tämäntyyppisissä olioissahan kaikkien kenttien arvot tallennetaan valmiiksi merkkijonoina.

`RS2s`-olion läpikäynti oikein tyypein taasen on suhteellisen hidasta, koska tietorakenteeseen tallennetusta merkkijonosta joudutaan jäsentämään Java-lähdekoodin avulla oikeantyyppinen muuttuja, kuten kokonaisluku, totuusarvo tai aikaleima. `RS2s`-olioita kannattaa käyttää silloin, kun SQL-kyselyllä haetaan pääosin merkkijonoja tai kyselyn tulosta hyödynnetään pääosin merkkijonoina alkuperäisistä kenttien tyypeistä riippumatta. Vastaavasti `RS2`-oliot ovat parhaimmillaan silloin, kun hakutuloksessa kenttien tyypit eivät pääosin ole merkkijonoja.

`CachedRowSet`-rajapinnan toteuttavia olioita kannattaa hyödyntää yleensä

vain silloin, kun hakutuloksen hyödyntämiseksi vaaditaan välimuistiratkaisu sekä ehdoton yhteensopivuus Java-koodille, jota voidaan käyttää muuallakin kuin Korppi-järjestelmässä. Kun hakutulos käydään vain kertaalleen läpi, kannattaa pääosin hyödyntää `ResultSet`-rajapinnan toteuttavia olioita. Tällöin ylimääräisiä muistiresursseja ei tarvita mainittujen välimuistin tarjoavien olioiden luomiseksi. Tällaisessa käyttötilanteessa myöskään tietokantayhteyden aukioloaikaa ei saada lyhennettyä.

Korppi-järjestelmän sisäisessä toiminnassa `RS2`- ja `RS2s`-olioita kannattaa yleensä käyttää mieluummin kuin `CachedRowSet`-rajapinnan toteuttavia olioita. `RS2`-luokkaan on nimittäin toteutettu lisäominaisuuksia, joilla tehostetaan kyselyiden hakutulosten hyödyntämistä ja jatkokäsittelyä. **Hakutuloksen rivit voidaan** kyseisen luokan avulla vaikkapa **järjestää** (engl. *sort*) haluttuun järjestykseen joko suoraan jonkin sarakkeen arvojen mukaan tai rajapinnan `java.util.Comparator` toteuttavan vertaimen avulla. Lisäksi `RS2`-olioon tallennetun **hakutuloksen perään voidaan liittää toisen SQL-kyselyn hakutulos**, tai kyseisen olion tietorakenteesta voidaan etsiä jonkin hakutuloksen kentän arvo yksinkertaisella metodikutsulla ilman lähdekoodiin kirjoitettua läpikäyntialgoritmia.

`RS2`- ja `RS2s`-luokat sisältävät käytännöllisiä apuvälineitä isojen kyselyiden pilkkomiseen pienemmiksi palasiksi. Nämä luokat toteuttaviin olioihin voidaan ensin hakea vaikkapa jonkin tehokkaan peruskyselyn avulla olennaisia tunnistetietoja jatkokäsittelyn perustaksi. Tämän jälkeen useammalla yksinkertaisella ja tehokkaalla lisäkyselyllä täydennetään edellä saatuja tunnistetietoja lopputuloksessa tarvituilla nimitiedoilla, selitteillä ja lukumäärätiedoilla. Lisätiedot voidaan hakea tietokantahaun sijasta toisinaan palvelimen muistissa olevista tietorakenteista.

Kokonaisuutena tietojen koostaminen kuvatulla tavalla osissa on yleensä tehokkaampaa kuin yhden kaikki tiedot kerralla hakevan SQL-kyselyn suorittaminen. Java-lähdekoodia kuvatussa ositusratkaisussa joudutaan toteuttamaan hieman enemmän, mutta kokonaisuutena Java-lähdekoodin ja sovelluslogiikan sekä SQL-lauseiden määrä voi silti olla pienempi kuin ison SQL-kyselyn hyödyntämiseen tarvittava Java-lähdekoodin määrä. Lisäksi **pienempiin osiin jaetun kokonaisuuden ylläpidettävyys ja testattavuus paranee** luvussa 7.1 esitetyn mukaisesti, koska kullakin osalla on oma selkeä ja yksinkertainen tehtävänsä.

7.6 Tietokantayhteyksien varastointi Korppi-järjestelmässä

JDBC-tietokantarajapinnan kautta voidaan toisinaan hyödyntää tietokantayhteyksien varastointia (engl. *database connection pooling*). Tietokantayhteyksien varastointi ja uudelleenkäyttö tehostaa järjestelmien käyttöä, koska suurikin määrä tietokantaan kohdistuvia pyyntöjä eri tietokanta-asiakkailta ja -sovelluksilta voidaan ohjata kulkemaan vain muutaman todellisen tietokantayhteyden kautta.

Yksi tällainen välimuistitekniikka on Korppi-järjestelmässä syksystä 2000 asti käytössä ollut PoolMan-niminen sovellus [31]. Keväällä ja syksyllä 2003 sen käytössä kuitenkin huomattiin ratkaisemattomia ongelmia, joiden vuoksi sovelluksen käytöstä luovuttiin uusien Korppi-palvelimien myötä vuoden 2004 alussa. Käyttöön otettiin samalla standardimpi Korppi-järjestelmän käyttämän PostgreSQL-tietokannanhallintajärjestelmän tarjoama JDBC-tekniikka [12], johon sisältyy tietokantayhteyksien varastointi ja uudelleenkäyttö.

Sinällään PoolMan toimi pienellä tietokantakyselykuormalla hienosti. Se tarjosi Korppi-järjestelmälle useampia loogisia yhteyksiä, jota määrää ei oltaisi fyysisinä yhteyksinä hidastumisen ja resurssienkulutuksen takia voitu käyttää. Lisäksi PoolMan dokumentaationsa mukaan tarjosi myös välimuistin kyselyille ja niiden tuloksille. Korppi-järjestelmän tietokantaan kohdistuneiden tietokantayhteyksien avausajat putosivat oleellisesti PoolManin ansiosta. Tämä on PoolManista saatu oleellinen hyöty, koska yksi Korppi-järjestelmän sivun suoritus saattaa avata kymmeniä tietokantayhteyksiä peräkkäin.

PoolMan-sovelluksessa havaittiin kuitenkin sellaisia perustavanlaatuisia ongelmia, että sen käytöstä oli pakko luopua Korppi-järjestelmän käytön lisääntyessä. Ruuhkahetkinä ilmeisesti juuri PoolManin takia satunnaisesti kaikki tietokantayhteydet lakkasivat toimimasta, jonka takia koko järjestelmän käyttö ruuhkautui tai pysähtyi kokonaan. Tästä tuli käyttäjiltä paljon negatiivista palautetta syksyllä 2003, jonka takia Korppi sai negatiivista julkisuutta toimimattomuudestaan erityisesti ruuhka-aikoina. Alkuvuodesta 2004 Korppi siirrettiin uusiin tehokkaampiin palvelimiin ja samalla PoolManin käytöstä luovuttiin, jonka jälkeen vastaavaa ei ole tapahtunut. On tietysti vaikea arvioida, johtuivatko syksyn 2003 kaatumiset ja toimimattomuus pelkästään PoolManista vai myös silkasta palvelintehon puutteesta. Valitettavasti PoolManista ei ollut saatavilla lähdekoodia, jotta Korppi-kehittäjät olisivat voineet tutkia ja korjata sen ongelmia. Itse sovellustakaan ei enää tueta. Keväällä 2001 ilmestyi PoolManista versio 2.0, mutta kyseistä betaversiota ei ole sen jälkeen enää päivitetty.

PoolMan kuitenkin tarjosi ominaisuuksia, joita ilman Korppi-järjestelmä ei olisi käytännössä edes toiminut. Nämä ominaisuudet huomattiin kunnolla vasta silloin, kun PoolManin käytöstä väliaikaisesti luovuttiin kehitysympäristössä Korppi-sovellusta ohjelmoitaessa. Tämän jälkeen Korppi-järjestelmän lähdekoodia jouduttiin korjaamaan monilta tietokantaan liittyviltä osilta siten, että useita sisäkkäisiä tietokantayhteyksiä ei enää saa käyttää. Lisäksi päätettiin, ettei `ResultSet`in toteuttavia olioita avoimine kantayhteyksineen saa enää viedä parametrina Java-metodeilta toisille samalla, kun kyseisellä `ResultSet`-oliolle varattua auki olevaa tietokantayhteyttä käytetään jo seuraavan `ResultSet`-olion tuottamiseen. Tietokantayhteys pysyy täten kerrallaan varattuna vain enintään yhden SQL-kyselylauseen suorittamista varten.

7.7 Tietokantayhteyksien hyödyntämisperiaatteita Korppi-järjestelmässä

Avattavaa tietokantayhteyttä kannattaa hyödyntää mahdollisimman paljon siten, että **mahdollisimman vähän ylimääräisiä tietokantayhteyksiä** tarvittaisiin toiminnon suorittamiseen. Tietokantayhteyksien kokonaisaukioloaika tulisi lisäksi saada mahdollisimman lyhyeksi. Tämä tarkoittaa samalla myös sitä, että **SQL-kyselyiden suorittamiseen kulutettu aika tulisi minimoida**. Edelleen tietokantayhteyden auki ollessa tulisi suorittaa mahdollisimman vähän sellaista sovelluksen lähdekoodia, joka ei liity tietokannan käyttöön mitenkään. Näin ylimääräinen tuottamaton aika tietokantayhteyksistä saataisiin minimoitua, ja tietokantayhteyksien käyttö olisi mahdollisimman tehokasta.

Alunperin ensimmäisten vuosiensa aikana Korppi-järjestelmässä ideana oli se, että kullekin kirjautuneelle käyttäjälle tai edes kullakin sivulla käytettäisiin vain yhtä tietokantayhteyden tarjoavaa `DB`-oliota kerrallaan. Tämä idea kuitenkin kaatui siihen, että kullakin `DB`-oliolla sai silloisella Korppi-järjestelmän lähdekoodilla olla kerrallaan haettuna vain yksi `ResultSet`-rajapinnan toteuttava olio eli tietokantakyselyn hakutulos. Jokaista rinnakkain tarvittua SQL-kyselyn hakutulosta varten jouduttiin siten luomaan oma tietokantayhteyden tarjoava `DB`-olio, jolloin noin 60 tietokantayhteyden enimmäismäärä tuli hyvin nopeasti vastaan, eikä järjestelmän käyttö enää onnistunut. Tällainen tilanne saattoi tapahtua pahimmillaan jo 20–30 yhtäaikaisella käyttäjällä.

Luvussa 7.4 kuvatut `RS2`- ja `RS2s`-luokat ovat pelastaneet mainitulta ongelmalta

kesästä 2002 alkaen. Kyseisten Java-luokkien avulla tietokantahaun tulos tallennetaan olioiden sisäiseen tietorakenteeseen, jolloin auki olevaa tietokantayhteyttä ei välttämättä enää tarvita SQL-lauseen suorittamisen jälkeen.

Korppi-järjestelmän lähdekoodia on muokattu vuosien saatossa siten, että jo valmiiksi **tietokantayhteyden avannut `DB`-olio voidaan viedä Java-luokkien metodeille parametrina**. Tällöin säästyy aina hieman aikaa verrattuna siihen, että kukin metodi loisi oman `DB`-olionsa ja avaisi sille tietokantayhteyden. Väliähän ei ole sillä, mikä `DB`-olio SQL-kyselyt ja -päivitykset suorittaa, kunhan vain kyselyn tulos saadaan käyttöön kutsuvassa koodilohkossa. Vaikka yhteyden ottaminen kestää yleensä vain muutamia millisekunteja, on se kuitenkin turhan suuri osuus kokonaisajasta, jos kyselyn tai koko sivun suorittaminen kestää esimerkiksi vain muutamia kymmeniä millisekunteja.

Etuna `DB`-olion ja avatun tietokantayhteyden metodille välittämisessä on myös se, ettei kutsuttavaan eli tietokantayhteyttä hyödyntävään lähdekoodiin tarvitse kirjoittaa `try-finally` -lohkoa, joka käsittelee tietokantayhteyden avauksen ja sulkemisen luotettavasti. Jos metodille tuodaan parametrina `DB`-olio, pääsääntöisesti myös tämän mahdollisesti heittämät poikkeukset tulee heittää eteenpäin metodin kutsujalle, joka vasta reagoi niihin tarpeen mukaan.

Mitä useampia tietokantayhteyksiä voidaan samassa ajanjaksossa muodostaa, sitä jouhevampaa järjestelmän käyttö on, sekä sitä nopeammin ja tehokkaammin se käyttäjien kannalta toimii. Erityisesti kiireisenä aikana eli normaalisti arkiiltapäivisin Korppi-järjestelmän lähdekoodin optimointi näkyy siten, että käytön huippukohdat eivät välttämättä enää juurikaan hidasta järjestelmän käyttöä.

Yhtäaikaisia tietokantayhteyksiä on Korppi-järjestelmässä käytettävissä noin sata, jotta tietokantapalvelin ei ylikuormittuisi useista sadoista yhtäaikaisista yhteyksistä. Tällöin avonaisia tietokantayhteyksiä metodien välillä hyödyntäen myös muu samanaikainen käyttö voi tehostua, koska jonkin toisen käyttäjän suoritusvuorossa olevan metodin ei tarvitse odottaa oman tietokantayhteytensä avautumista kovin pitkään.

7.8 Korppi-järjestelmän käyttötilastoja

Korppi-järjestelmän tietokanta- ja sovelluspalvelimien tehoa on kasvatettu vuosien varrella siten, että tietokannan hyödyntämisen ylikuormitustilanteita ei ole enää vuoden 2004 alussa käyttöönotettujen palvelinten jälkeen sattunut. Tammikuussa 2006 otettiin Korppi-järjestelmässä käyttöön suorituskykyisempi tietokantapalvelin sekä kesällä 2007 tehokkaammat sovelluspalvelimet, jotka taas omalta osaltaan ovat poistaneet järjestelmän käytön pullonkauloja.

Korppi-järjestelmän seitsemän vuoden elinaikana kevääseen 2008 mennessä on Korppi-lähdekoodia sekä tietokannan käyttöä tehostettu huomattavasti siten, että keskimääräisessä käytössä vain joka sadannen kyselyn suorittamiseen kuluu yli 200 millisekuntia. Suoritusajaltaan yli sekunnin kestäviä kyselyitäkin on keskimäärin vain 0.2% kaikista suoritetuista kyselyistä. Toisaalta käyttäjät huomaavat kuitenkin erityisesti hitaat sivut hitaine SQL-kyselyineen, vaikka niitä ei lukumääräisesti monta olisikaan.

Useimmissa SQL-kyselyissä on silti edelleen optimointivaraakin. Luvun 7.1 mukaisesti tehostamisella saatava hyöty verrattuna optimointiin kuluvaan aikaan pitää kuitenkin aina harkita tarkkaan. Ei ole juurikaan hyötyä optimoida tietokantakyselyä, joka normaalisti vie muutaman millisekunnin. Tällaisissa tapauksissa optimointiin kulunutta aikaa tuskin koskaan saadaan takaisin ajansäästönä, ellei optimoitua kyselyä suoriteta erityisen usein.

Taulukossa 7.2 on esitetty tilastotietoja Korppi-järjestelmän käytöstä kahden vertailukelpoisen vuorokauden ajalta. Vertailuun valittujen vuorokausien aikana järjestelmää käytettiin kohtuullisen runsaasti, mutta vuoden ajanjaksolla kyseiset vuorokaudet eivät kuitenkaan olleet ruuhkaisinta aikaa. Suoritusajat on esitetty tunteina, minuutteina ja sekunteina, paitsi sivujen keskimääräinen suoritusaika on esitetty millisekunteina. Oikeanpuoleisimmassa sarakkeessa esitetään prosentteina, kuinka moninkertainen syksyn 2007 vertailuvuorokauden lukema on verrattuna kevään 2004 vastaavaan lukemaan. Asteriskilla $\star$ merkityissä lukemissa ei ole huomioitu Korppi-järjestelmän `AutoNumber`-luokan tuottamia tietokantayhteyksiä, eikä siten myöskään näiden tuottamaa ”tyhjäkäyntiaikaa”.

Korppi-järjestelmään kirjautumisten lukumäärä on lisääntynyt keväästä 2004 syksyyn 2007 noin kolminkertaiseksi järjestelmän laajenneen käytön myötä. Vastaavasti sivunavausten lukumäärä on lähes viisinkertaistunut, joka kielii Korppi-järjestelmän uusista ja käytetyistä toiminnoista sekä siitä, että järjestelmää hyödynnetään aiempaa enemmän laitoksilla ja tiedekunnissa. Järjestelmän sivujen ko-

konaissuoritusaika sekä tietokantayhteyksien ja tietokantakyselyiden kokonaisaika ovat kukin lähes kymmenkertaistuneet, mutta suorituskykyisempien palvelimien ja tietokannan tehokkaamman hyödyntämisen ansiosta keskimääräiset tietokantayhteysajat sekä sivujen ja SQL-kyselyiden suoritusajat ovat vain noin kaksinkertaistuneet. Pidentyneet keskimääräiset SQL-kyselyiden suoritusajat kielivät osittain myös siitä, että suoritetut SQL-kyselyt ovat aiempaa monimutkaisempia ja siten yleensä hitaampia. Samaten sivuilla esitetään aiempaa enemmän tietokannasta haettuja tietoja.

Korppi-järjestelmän `AutoNumber`-luokka on syyskuusta 2005 alkaen pitänyt avaamaansa kantayhteyttä auki kerrallaan aina vähintään minuutin ajan, jotta turhia tietokantayhteyden avaamisia ja sulkemisia ei tapahtuisi liikaa. Kyseisen Java-luokan tietokantayhteyksiä avattiin ja pidettiin auki syksyn 2007 vertailuvuorokauden aikana yhteensä 326 kertaa, ja kyseiset yhteydet veivät aikaa hieman alle puolet tietokantayhteyksien kuluttamasta kokonaisajasta. Näiden `AutoNumber`-kutsujen tuottamat SQL-kyselyt veivät aikaa yhteensä 2 minuuttia 7 sekuntia. Kyseinen pariminuuttinen on laskettu mukaan taulukon 7.2 lukemiin, koska kyseiset toimenpiteet kuuluvat tavanomaiseen Korppi-järjestelmän tietokannan hyödyntämiseen.

Ilman em. `AutoNumber`-luokan tuottamia tietokantayhteyksiä veivät JSP-sivukutsujen aiheuttamat tavanomaiset tietokantayhteydet seinäkelloaikaa runsaat 16 tuntia 17 minuuttia, joka on taulukossa 7.2 huomioitu vertailuvuorokauden 19.11.2007 kohdalla asteriskilla $\star$ merkattuna.

Muutamissa kohdissa Korppi-järjestelmän lähdekoodia tietokantayhteys suljetaan jonkin suhteellisen pitkään kestävän toimenpiteen ajaksi. Tämän jälkeen tietokantayhteys avataan uudelleen. Mainitulla kantayhteyden väliaikaisella sulkemisella yhteyttä ei pidetä turhaan auki, koska suoritettavan toiminnon suoritusajasta ei ole välttämättä mitään tietoa. Toiminnon suoritus voi kestää minuuttejakin, jolloin ilman yhteyden sulkemista kyseiseksi ajaksi kantayhteys voisi estää toisten käyttäjien toimintojen suorittamista samaan aikaan. Valitettavasti joskus tämä ennakointi on tarpeetonta, sillä toisinaan yhteys avataan sulkemisen jälkeen jo muutaman millisekunnin kuluttua. Tällaisissa tapauksissa yhteyttä ei olisi tietenkään kannattanut sulkea ollenkaan, koska sulkemiseen ja avaamiseen kuluu aina jonkin verran aikaa ja palvelinresursseja.

Selite	**22.3.2004**	**19.11.2007**	**Kasvu-%**
Kirjautumisia (kpl)	2776	8187	195%
Sivunavauksia (kpl)	28825	130416	352%
Sivujen suoritusaika (h:min:s)	1:48:19	17:57:10	894%
Sivujen keskimääräinen suoritusaika (ms)	225	495	120%
SQL-kyselyitä (kpl)	546548	2542993	365%
SQL-kyselyiden kokonaisaika (h:min:s)	1:31:15	13:43:26	802%
SQL-kyselyiden keskimääräinen suoritusaika (ms)	10	19	90%
SQL-päivityksiä (kpl)	–	138872	–
SQL-päivitysten kokonaisaika (h:min:s)	–	0:45:30	–
SQL-päivitysten keskimääräinen suoritusaika (ms)	–	19	–
Tietokantayhteyksiä (kpl)	356845	1482592	315%
Tietokantayhteyksien kokonaisaika (h:min:s)	1:50:18	* 16:17:32	786%
Tietokantayhteyksien keskimääräinen yhteysaika (ms)	18	39	116%
Tietokantayhteyksien tyhjäkäyntiaika (h:min:s)	0:19:03	* 2:34:06	708%

Taulukko 7.2: Korppi-järjestelmän käyttö ja käytön lisääntyminen kahden vertailukelpoisen vuorokauden välillä.

Etukäteen on kuitenkin yleensä vaikea arvioida, milloin yhteys kannattaa sulkea jonkin toiminnon ajaksi. Tässä mielessä Korppi-järjestelmän tulisi olla itseoppiva tai itsesäätelevä järjestelmä, jolloin se tietyn toimettoman ajan jälkeen sulkisi tietokantayhteyden ja avaisi sen taas normaaliin tilaan yhteyttä uudelleen tarvittaessa. Parhaassa tapauksessa kyseinen tekniikka otettaisiin käyttöön vain silloin, kun yhtäaikaisista tietokantayhteyksistä alkaa olla pulaa. Tällöin on ehdottomasti pidettävä huolta siitä, että aktiivisia yhteyksiä ei turhaan suljeta, koska se kuormittaisi järjestelmää vielä enemmän kuin yhteyksien pitäminen auki.

7.9 Tietokantayhteyksien aukioloaikojen ja SQL-kyselyiden suoritusaikojen minimointi

SQL-kyselyiden suorittamiseen vaadittava kokonaisaika on aina vähemmän kuin tietokantayhteyksien kokonaisaukioloaika. Tämä johtuu siitä, että kunkin SQL-kyselyn suorituksen aikana tietokantayhteyden tulee olla auki, muutenhan kyselyä ei voitaisi lainkaan suorittaa. Kunkin käyttäjän toimien oletetaan olevan peräkkäisiä, joten järjestelmän sivuja avataan peräkkäin, eikä yhtäaikaisesti rinnakkain. Tällöin voidaan olettaa myös, että yhden käyttäjän suorittamat toiminnot tietokantatasolla ovat peräkkäisiä. Toki Korppi-järjestelmän lähdekoodin tietokantayhteyksien osittaisesta optimoimattomuudesta johtuen yhden käyttäjän sivunavauksen aikana voi samaan aikaan olla auki useampia sisäkkäisiä tietokantayhteyksiä, vaikka vain yhdellä niistä kerrallaan suoritetaan SQL-kyselyä. Tällöin auki olevat tietokantayhteydet toimivat ”tyhjäkäynnillä”, mutta niiden kuluttama aika tulee silti laskea tietokantayhteyksien kokonaisaukioloaikaan.

Edellisen kappaleen oletuksista poiketen ei voida kovin luotettavasti arvioida sitä, kuinka kyselyiden suoritukseen kuluva kokonaisaika suhtautuu sivujen suoritukseen kuluvaan kokonaisaikaan. Tämä vaatisi tarkempaa analyysiä avatuista sivuista eli siitä, mitä sivut ovat käytännössä tehneet. Osa Korppi-järjestelmän sivuista ei nimittäin käytä tietokantaa lainkaan, joten ne eivät myöskään luo tietokantaolioita, eivätkä siten suorita SQL-kyselyitä edes Java-luokissa olevien metodien kautta. Jotkin Korppi-järjestelmän sivut taas hyödyntävät tietokantaa todella runsaasti saadakseen sivun näyttämään pyydetyt tiedot.

Luvun 7.8 taulukossa 7.2 **SQL-kyselyiden ulkopuolelle jäävät tietokantayhteyksien aukioloajat ovat siis minimoitavia aikoja**. Kyseinen aika on maaliskuun 2004 vertailuvuorokautena runsas 19 minuuttia ja marraskuussa 2007 run-

sas 2.5 tuntia. Kokonaistehokkuutta saadaan toki eniten parannettua sillä, että **pyritään lyhentämään ensisijaisesti SQL-kyselyiden suoritukseen kuluvaa aikaa**. Tähän tavoitteeseen päästään vaikkapa tehostamalla indeksien käyttöä (ks. luku 5.7), muokkaamalla SQL-kyselyitä nopeammiksi (ks. luku 7.1) tai muokkaamalla sovelluksen lähdekoodia tehokkaammaksi tietokannan käytön osalta (ks. luku 7.4).

Luvussa 7.4 esitelty `RS2`-luokka auttaa tietokantayhteyksien kokonaisaukioloajan minimoimisessa. Kyseisen luokan avulla nimittäin kaikki tietokannasta tarvittavat tiedot voidaan yleensä hakea Java-lähdekoodissa yhdessä tiiviissä `try-finally` -lohkossa, jonka jälkeen tietokantayhteyden ei enää tarvitse olla auki. Tämän jälkeen `RS2`-olion tietoja voidaan hyödyntää ilman auki olevaa tietokantayhteyttä, jolloin säästetään kallisarvoisia tietokantaresursseja järjestelmän muuta käyttöä varten. Toki `RS2`-tekniikankin osalta tulee tapauskohtaisesti harkita se, kannattaako `RS2`-olioiden hyödyntäminen. Jos tietokantakyselyn tulosta hyödynnetään vaikkapa vain tulostukseen tai merkkijonon loppuun lisäämiseen, voi olla tehokkainta käyttää perinteistä ja nopeinta `ResultSet`-tekniikkaa. Luvun 7.5 taulukossa 7.1 vertaillaan muun muassa `ResultSet`- ja `RS2`-olioiden luontinopeuksia sekä tietojen läpikäyntinopeuksia.

8 Korppi-järjestelmän toiminnan nopeuttamisesimerkkejä

Luvussa tarkastellaan muutamia esimerkkejä, joiden avulla käytännössä on saatu tehostettua Korppi-järjestelmän toimintaa. Suurin osa esimerkeissä käytetyistä tauluista ja kentistä esitellään Korppi-järjestelmän tietokantaraportissa [9]. Liitteessä D kuvataan tarkemmin `EXPLAIN ANALYZE` -tulosteen tulkintaa.

8.1 Käyttäjän varattavien aikojen haku indeksiä hyödyntäen

Esimerkissä halutaan hakea tietyltä aikaväliltä käyttäjän varattavat tapaamisajat. Hakutuloksena halutaan tapahtumien perustiedot sisältävästä `event`-taulusta tapaamisajankohta `begintime`- ja `endtime`-kentistä. Varatulle ajalle annettu nimi haetaan ryhmien perustiedot sisältävän `eventgroup`-taulun `name`-kentästä, jonka lisäksi tarvitaan kyseisen taulun `eventgroupid`-kentästä varauksen tekevän ryhmän tunniste. `space`-taulun `code`-kentästä saadaan selvitettyä huonetila, jossa varattu tapaamisaika järjestetään. Näiden lisäksi kyselyssä tarvitaan ryhmien henkilöjäsenet sisältävää `groupparticipant`-taulua, jonka `personid`-kentän henkilöt sisältävään `person`-tauluun kohdistuvan viite-eheyden avulla voidaan selvittää varattavan ajan tarjolle asettaneen ohjaajan henkilötiedot.

Kysely Korppi-käyttäjän kaikkien vuoden 2004 varattavien aikojen hakemiseksi on perusmuodossaan seuraavanlainen:

```
SELECT e.begintime, e.endtime, eg.name AS eventname,
s.code, eg.eventgroupid
FROM eventgroup AS eg, groupparticipant AS gp, event AS e,
eventspace AS es, space AS s
WHERE gp.personid = 4 AND gp.deleted = false
AND gp.userlevel > 0 AND gp.eventgroupid = eg.eventgroupid
AND eg.deleted = false AND eg.grouptypeid = 10
AND eg.eventgroupid = e.eventgroupid AND e.deleted = false
AND e.visibilityid = 4 AND e.eventid = es.eventid
AND es.spaceid = s.spaceid
AND '2004-01-01 00:00:00' <= e.begintime
AND e.endtime <= '2004-12-31 23:59:59';
```

Käytännössä kyseisiä varattavia aikoja kannattaa hakea yleensä yhden viikon ajalta, jotta ne voidaan esittää järkevästi viikkokalenterissa, eikä ylimääräisiä tietoja haeta turhaan.

Testiaineistolla kyselyn suorittaminen kestää kaikilla käytössä olevilla PostgreSQL:n versioilla noin 400 ms. Suoritusaika ei niinkään ole riippuvainen siitä, kenen ohjaajan varattavia aikoja haetaan, vaan varattavien aikojen lukumäärästä. Tämä johtuu lähinnä `event`-taulun aikaehtojen vertailusta. Käytännön kautta on huomattu, että Korppi-järjestelmän suorittamista hitaiksi luokiteltavista SQL-kyselyistä varsin suuri osa liittyy rivimääriltään isojen taulujen `timestamp`-tyyppisten kenttien vertailuun, joten aikavertailuja tulisi yleisemmin tehostaa tavalla tai toisella.

Kysely suoriutuu kohtalaisen tehokkaasti jo em. muodossa, mutta koetetaan silti, saataisiinko sitä tehostettua yksinkertaisin keinoin. Hakua voitaisiin kokeilla ensin jakaa alikyselyiden avulla pienempiin osiin. Koska pääkyselyssä halutaan näytettäväksi vain taulujen `event`, `eventgroup` ja `space` kenttiä, voitaisiin muut `FROM`-osassa olevat taulut käsitellä alikyselyjen avulla. Koska `space`-taulu yhdistetään muihin tauluihin `eventspace`- ja `event`-taulujen välityksellä, alikyselyyn voitaisiin siirtää ainoastaan `groupparticipant`-taulu. Kysely voitaisiin siis esittää `IN`-rakenteen avulla seuraavasti:

```
SELECT e.begintime, e.endtime, eg.name AS eventname,
s.code, eg.eventgroupid
FROM eventgroup AS eg, event AS e, eventspace AS es, space AS s
WHERE eg.eventgroupid IN (
  SELECT eg.eventgroupid FROM groupparticipant AS gp
  WHERE gp.personid = 4 AND gp.deleted = false
  AND gp.userlevel > 0
)
AND eg.deleted = false AND eg.grouptypeid = 10
AND eg.eventgroupid = e.eventgroupid AND e.deleted = false
AND e.visibilityid = 4 AND e.eventid = es.eventid
AND es.spaceid = s.spaceid
AND e.begintime >= '2004-01-01 00:00:00'
AND e.endtime <= '2004-12-31 23:59:59';
```

Vastaavasti EXISTS-rakenteen avulla esitettynä alikysely olisi seuraavanlainen:

```
...
WHERE EXISTS (
  SELECT true FROM groupparticipant AS gp
  WHERE gp.personid = 4 AND gp.deleted = false
  AND gp.userlevel > 0 AND gp.eventgroupid = eg.eventgroupid
)
...
```

Taulukossa 8.1 on esitetty em. kahden IN- ja EXISTS-rakenteilla testausympäristössä eri PostgreSQL-versioilla suoritettavien SQL-lauseiden keskimääräiset suoritusajat millisekunteina. Huomioitavaa on, että kysely suoritettuna PostgreSQL:n versiolla 7.1 oli aina nopeampi kuin versiolla 7.3. Mitään selkeää syytä tähän ei saatu selville, koska data ja taulujen rakenne indekseineen on kaikilla versioilla sama.

Tekniikka	**PG 7.1**	**PG 7.3**	**PG 7.4**
alkuperäinen	400	400	400
IN	1000	3500	420
EXISTS	420	900	740

Taulukko 8.1: SQL-kyselyiden keskimääräiset suoritusajat millisekunteina.

Jokaisella PostgreSQL:n versiolla kysely suoritetaan alikyselyn avulla liian hitaasti, sillä mitään parannusta alikyselyn käytöstä ei saatu. EXPLAIN ANALYZE -tulosteesta (ks. luku 6.3) nähdään kyselyn suoritettavan ”takaperoisesti” siihen nähden, miten se kannattaa suorittaa. Alikyselyjä käyttävät versiot kyselystä suoritetaan käyttäen periaatteena eräänlaista tekniikkaa ”hae kaikki ja poista ylimääräiset”. Tästä johtuen kyselyn eri vaiheissa käsitellään turhaan suuria datamääriä. Esimerkkitapauksessa ja -aineistolla alikysely suoritetaan lähes 7000 kertaa, vaikkakin käyttäen groupparticipant-taulussa olevaa indeksiä. Puolet suoritusajasta kuluu siihen, että saadaan haettua event- ja eventgroup-tauluista lopputulokseen vaikuttavat oikeat rivit. Erityisesti event-taulun kaikkien noin 300000 rivin läpikäyminen peräkkäishaulla hidastaa kyselyn suorittamista. Kyselyä tulee siis mieluummin tehostaa alkuperäisen mallin mukaan, jossa kaikki taulut ovat mukana pääkyselyssä.

Tutkitaan seuraavaksi voitaisiinko indeksien avulla lyhentää alkuperäisen kyselyn suoritusaikaa. Koska seuraavassa olevan EXPLAIN ANALYZE -tulosteen mu-

kaan kysely käyttää `event`-taulun lukemiseen testiaineistolla peräkkäishakua (engl. *sequential scan*), kokeillaan, saataisiinko se vaihdettua indeksin lukemiseksi.

```
...
  ->  Seq Scan on event e  (cost=0.00..7148.30 rows=14818 width=32)
        (actual time=1.12..329.40 rows=1649 loops=1)
      Filter: ((deleted = false) AND (visibilityid = 4)
      AND (begintime >= '2004-01-01 00:00:00+02'::timestamp
        with time zone)
      AND (endtime <= '2004-12-31 23:59:59+02'::timestamp
        with time zone))
...
```

`Filter`-kohdassa mainituilla ehdoilla luetuista riveistä suodatetaan esille ne rivit, joita kyselyn lopputuloksessa tarvitaan. Lisätään `event`-tauluun indeksi näillä ehdoilla seuraavasti:

```
CREATE INDEX event_del_vis_times
ON event (deleted, visibilityid, begintime, endtime);
```

Kyselyn suorittamisessa käytetään luotua indeksiä tällä testiaineistolla vain PostgreSQL:n versiolla 7.1, jolla **suoritusaika putoaa 80 millisekuntiin**. Muilla PostgreSQL:n versioilla indeksiä ei käytetä, joten suoritusaika ei näillä muutu.

Lisätty indeksi olikin oikeastaan vain tätä nimenomaista kyselyä varten suunniteltu. `Event`-taulun `visibilityid`-kenttää käytetään vain murto-osassa muita Korppi-järjestelmän suorittamia kyselyitä, joten pudotetaan ainakin se pois kyselyn määrittelystä. Sen sijaan loput kolme kenttää ovat sellaisia, jotka esiintyvät lähes poikkeuksetta kaikissa `event`-taulua käsittelevissä kyselyissä. Tästä johtuen arvataan, että kolme kyseessä olevaa kenttää sisältävä indeksi auttaisi tehostamaan muitakin kuin vain tätä tutkinnan alla olevaa kyselyä. Luodaan uusi indeksi seuraavasti:

```
DROP INDEX event_del_vis_times;
CREATE INDEX event_del_times ON event (deleted, begintime, endtime);
```

Kyseistä indeksiä ei kuitenkaan tutkittavaa kyselyä suoritettaessa käytetä millään käytössä olevalla PostgreSQL:n versiolla, joten suoritusajatkaan eivät muutu mihinkään suuntaan.

Koska `eventgroupid`-kenttä on lähes poikkeuksetta mukana kaikissa `event`-taulua koskevien kyselyiden vertailuehdoissa, kokeillaan lisätä tämä kenttä indeksiin em. aikakenttien perään seuraavasti:

```
DROP INDEX event_del_times;
CREATE INDEX event_del_times_eg
ON event (deleted, begintime, endtime, eventgroupid);
```

Tämä oli kuitenkin muutos huonompaan, koska nyt **kyselyn suoritusajat ovat aina 45–80 sekuntia**! Luotua indeksiä käydään läpi testiaineistolla runsaat 700 kertaa, joka on suoraan verrannollinen datamäärään. Jokainen läpikäynti vie aikaa 60 millisekuntia, joten indeksin läpikäyminen koko kyselyn osalta on suurin hidastava tekijä. Aikaa kuluu kuhunkin 735 läpikäyntiin 60 ms eli yhteensä 44100 ms.

Siispä poistetaan luotu indeksi ja koetetaan siitä toisenlaista versiota. Seuraavassa versiossa mukana ovat `deleted`-kenttää lukuunottamatta samat kentät eri järjestyksessä.

```
DROP INDEX event_del_times_eg;
CREATE INDEX event_eg_times ON event (eventgroupid, begintime, endtime);
```

Tämän muutoksen seurauksena suoritusajat näyttävät jo huomattavasti paremmilta. Indeksin ansiosta **PostgreSQL:n versiolla 7.1 kysely suoritetaan 80 millisekunnissa, versiolla 7.3 noin 110 millisekunnissa ja versiolla 7.4 noin 50 millisekunnissa**. Kokeillaan vielä vaikuttaako se, jos indeksissä on mukana pelkästään `eventgroupid`-kenttä.

```
DROP INDEX event_eg_times;
CREATE INDEX event_eg ON event (eventgroupid);
```

Nyt PostgreSQL 7.1 ei enää suostu käyttämään indeksiä lainkaan, jolloin suoritusaika kasvaa alkuperäiseen 400 millisekuntiin. Uudemmat PostgreSQL-versiot käyttävät indeksiä, joten versiolla 7.3 suoritusaika on 100 ms ja versiolla 7.4 noin 35 ms. Jos indeksiin lisätään `eventgroupid`-kentän perään järjestyksessä kentät `begintime`, `endtime` ja `deleted`, vain PostgreSQL:n versiolla 7.1 muutos vaikuttaa. Nyt kyseisellä PostgreSQL:n versiolla indeksiä käytettäessä kyselyn suoritusaika putoaa 80 millisekuntiin, joten kysely suoritetaan testiaineistolla kuitenkin kyseisellä PostgreSQL:n versiolla 7.1 edelleen nopeammin kuin uudemmalla PostgreSQL:n versiolla 7.3.

Tämän nopeammaksi kyselyä ei tarvinne saadakaan, koska optimointiin kuluvaa aikaa ei varmaankaan koskaan saada takaisin hyötynä hivenen nopeammasta kyselystä. Haettaessa jonkun ohjaajan varattavia aikoja yhden viikon ajalta suoritetaan kysely 20–40 millisekunnissa, jos aikoja ei löydy.

8.2 Alikyselyiden käyttäminen ei aina ole tehokasta

Alikyselyä käyttävissä kyselyissä on oleellista se, että **tietokantaohjelmisto pakotetaan suorittamaan kysely tietyllä tavalla ja tietyssä järjestyksessä.** Ohjelmoija siis olettaa, että hän tietää paremmin, miten kysely kannattaa suorittaa. Siten tietokannanhallintajärjestelmälle ei anneta vapauksia päättää parhaasta suorittamistavasta, kun otetaan huomioon taulujen rakenne, indeksit, datan määrä ja muut ehdot.

Ongelmana varsin usein Korppi-järjestelmän tapauksessa on ollut ja on edelleen se, että muun muassa tietokantakyselyt kirjoitetaan turhan nopeasti miettimättä parasta toteutustapaa. Ongelman perimmäisenä syynä on se, että jotenkuten toimiva versio pitää saada nopeasti aikaan. Tällöin myöhemmin tulisi varata enemmän aikaa kartoittaa ja optimoida lähdekoodissa olevia hitaasti suoriutuvia SQL-kyselyitä.

PostgreSQL:n versiossa 7.4 on dokumentaationsa [30] mukaan huomattavasti optimoitu alikyselyissä `IN`-rakennetta (ks. luku 5.4) käyttävien SQL-kyselyiden suoritusnopeutta. Kyseistä alikyselyrakennetta käyttävien kyselylauseiden kerrotaan olevan vähintään yhtä nopeita kuin alikyselyissä `EXISTS`-rakennetta (ks. luku 5.4) käyttävien kyselyiden. Korppi-järjestelmän lähdekoodissa on `IN`-rakenteita muokattu `EXISTS`-rakenteiksi, koska Korppi-järjestelmässä tehokkaampi 7.4-sarjan PostgreSQL-palvelinohjelmisto otettiin käyttöön vasta vuoden 2006 alussa [10]. PostgreSQL 7.4 julkaistiin loppuvuodesta 2003.

`IN`-rakenne voidaan varsin helposti muokata `EXISTS`-rakenteeksi (ks. luku 5.8). Tarkasteltavassa esimerkissä alkuperäinen tietokantakysely hakee Korppi-järjestelmän tietokannasta erään käyttäjäkyselyn tiettyjen kysymysten (`questionfield`-taulun `questionid`-kentästä) kaikki vastaukset ja vastaajat (`questionresult`-taulun kentistä `personid` ja `value`) seuraavasti:

```
SELECT qr.questionfieldid, qr.personid, qr.value
FROM questionresult AS qr
WHERE qr.deleted=false AND qr.questionfieldid IN (
  SELECT qf.questionfieldid
  FROM questionfield AS qf
  WHERE qf.deleted=false AND qf.questionid IN (
    1389,1340,1095,1096,1097,1100,1101,1102,1103,1104,
    1112,1113,1114,1115,1116,1117,1323,1324,1325,1326
  )
);
```

SQL-kysely lienee alunperin muotoutunut tällaiseksi, koska sen käyttämät taulut ja ehdot on saatu sovelluksen ohjelmoijan kannalta näyttämään varsin rakenteiselta ja yksinkertaiselta, ja siten ylläpidettävältä.

Alunperin keväällä 2004 Korppi-järjestelmän tietokannan suoritettavaksi annettiin kyselylause, jossa oli sisemmässä `IN`-listassa 122 kysymyksen ID-arvoa. Kyseisen alkuperäisen `IN`-alikyselyrakennetta ja pitkää ID-arvolistaa käyttävän kyselyn suorittaminen **kesti runsaat kaksi tuntia**, jolloin kyselyn hitaus huomattiin. **`EXISTS`-rakenteella** esitettynä sama 122 ID-arvoa sisältävä kysely suoritettiin **muutamassa sekunnissa**.

Luvun kyselyissä on mainitusta pitkästä ID-arvolistasta esitetty vain pieni osajoukko, mutta supistettukin ID-lista tuottaa optimoimattomana erittäin raskaan kyselyn tietokannalle. **ID-arvolistaltaan lyhennetty `IN`-rakennetta käyttävä kysely suoritetaan runsaassa kuudessa sekunnissa.** Vaikka kysely ei ole tämän pitempi, eikä siinä ole pitkää `IN`-listaa, eikä muutakaan olennaisesti hitauteen viittaavaa, on sen viemä aika kuitenkin niin pitkä, että kyselyä kannattaa ehdottomasti alkaa tutkia.

Mikä toteutusratkaisu kyselyssä siis vie aikaa? Tämä selviää tarkastelemalla kyselyä luvussa 6.3 kuvatulla `EXPLAIN ANALYZE` -moodilla. Ennen tätä voidaan kysely kuitenkin muokata käyttämään `EXISTS`-rakennetta `IN`-rakenteen sijasta. Nimittäin käytännössä useimmiten `EXISTS`-rakenteen avulla kysely suoritetaan nopeammin ja tehokkaammin kuin `IN`-rakennetta käytettäessä. Tämän nimenomaisen kyselyn pääkyselyssä tarvitaan `questionresult`-taulun tietoja ainoastaan silloin, kun alikyselyn tuloksena saadaan vähintään yksi täsmäävä rivi `questionfield`-taulusta. Tällöin `EXISTS`-rakenteen käyttö on myös loogisesti perusteltua `IN`-rakenteen sijasta (ks. luku 5.4), koska kaikkia täsmääviä alikyselyn tuloksia ei tarvita. Samasta syystä `EXISTS`-alikysely voi palauttaa vaikkapa pelkän arvon tosi (engl. *true*) taulun kentän sijasta.

`EXISTS`-rakennetta käyttävä esimerkin kysely on seuraavanlainen:

```
SELECT qr.questionfieldid, qr.personid, qr.value
FROM questionresult AS qr
WHERE qr.deleted=false AND EXISTS (
  SELECT true
  FROM questionfield AS qf
  WHERE qr.questionfieldid=qf.questionfieldid
  AND qf.deleted=false AND qf.questionid IN (
    1389,1340,1095,1096,1097,1100,1101,1102,1103,1104,
    1112,1113,1114,1115,1116,1117,1323,1324,1325,1326
  )
);
```

Seuraavassa on esitetty tämän `EXISTS`-kyselyn `EXPLAIN ANALYZE` -tuloste. Sisimmän kyselyn `Filter`-ehtoja on lyhennetty, mutta `Filter`-ehdon idea selviää jäljellejääneistäkin riveistä.

```
Seq Scan on questionresult qr  (cost=0.00..876117.79 rows=97837 width=28)
  (actual time=711.01..2100.12 rows=1247 loops=1)
  Filter: ((deleted = false) AND (subplan))
  SubPlan
    ->  Index Scan using questionfield_pkey on
        questionfield qf  (cost=0.00..4.43 rows=1 width=0)
        (actual time=0.01..0.01 rows=0 loops=195649)
          Index Cond: ($0$ = questionfieldid)
          Filter: ((deleted = false) AND (
               (questionid = 1389) ...
            OR (questionid = 1325)
            OR (questionid = 1326)))
```

Esimerkissä ulomman `IN`-alikyselyn käyttö pakottaa tietokannanhallintajärjestelmän suorittamaan kyselyn tietyssä järjestyksessä siten, että jokaiselle `questionresult`-taulun riville suoritetaan `IN`-alikysely. Käytännössä alikysely suoritetaan vajaa 200 000 kertaa, ja vaikka yhden alikyselyn suorittaminen sujuu millisekunnin sadasosassa, yhteensä kaikkien kertojen suorittaminen vie kyselyn kokonaissuoritusaikana noin kaksi sekuntia. Vaikka alikyselyssä käytetäänkin `questionfield`-taulun perusavaimen indeksiä, on suoritusaika ja resurssien kulutus silti kokonaisuutena liian suuri. Lisäksi käytännön tilanteessa `IN`-lista sisältää usein huomattavasti enemmän alkioita kuin edellä on esitetty.

Kun tietokannanhallintajärjestelmälle annetaan vapaus suorittaa kysely parhaaksi katsomallaan tavalla, yhdistetään alikysely pääkyselyyn. Seuraavassa on esitetty SQL-kysely tällaisena.

```
SELECT qr.questionfieldid, qr.personid, qr.value
FROM questionresult AS qr, questionfield AS qf
WHERE qr.deleted=false
AND qr.questionfieldid=qf.questionfieldid AND qf.deleted=false
AND qf.questionid IN (
  1389,1340,1095,1096,1097,1100,1101,1102,1103,1104,
  1112,1113,1114,1115,1116,1117,1323,1324,1325,1326
);
```

Kysely suoritetaan nyt erittäin nopeasti verrattuna alkuperäiseen kyselyyn ja siihen datamäärään nähden, jota se joutuu käsittelemään. Kyselyn **suoritusaika on 370 ms**, kun käsiteltävänä on yhteensä 10000 kysymyskenttää kertaa 196000 vastausta kysymyksiin eli noin kaksi miljardia tietoalkiota. Seuraavassa esitettävän `EXPLAIN ANALYZE`-tulosteen `Hash`- ja `Hash join` -merkinnät tarkoittavat sitä, että PostgreSQL on yhdistänyt kaikki rivit tarvittavista tauluista, ja käsittelee isoa tulosjoukkoa sisäisesti nopeammin. Kummankin tarvittavan taulun rivit luetaan läpi vain kerran, joten nopeushyöty on valtava.

```
Hash Join  (cost=375.76..6224.84 rows=3051 width=32)
(actual time=126.05..370.50 rows=1247 loops=1)
  Hash Cond: ("outer".questionfieldid = "inner".questionfieldid)
  ->  Seq Scan on questionresult qr
    (cost=0.00..4343.39 rows=195674 width=28)
    (actual time=0.01..282.67 rows=195649 loops=1)
        Filter: (deleted = false)
  ->  Hash  (cost=375.38..375.38 rows=154 width=4)
    (actual time=0.57..0.57 rows=0 loops=1)
        ->  Index Scan using questionid_questionfield_key, ...
          on questionfield qf (cost=0.00..375.38 rows=154 width=4)
          (actual time=0.06..0.51 rows=72 loops=1)
              Index Cond: ((questionid = 1389) OR (questionid = 1340)
              ... OR (questionid = 1352) OR (questionid = 1353))
              Filter: (deleted = false)
```

Kysely käyttää peräkkäishakua (engl. *sequential scan*) `questionresult`-taulun lukemiseen, mutta se ei tämän kyselyn suorituksessa haittaa. Taulun rivit nimittäin joudutaan joka tapauksessa lukemaan ainakin kertaalleen läpi. Lisäksi apuna on `questionfield`-taulun `questionid`-kenttään luotu indeksi. `questionid`-arvothan annettiin kyselyssä `IN`-listan avulla, joten indeksistä haku onnistuu nopeasti.

Pelkällä kyselyn muokkauksella saatiin nyt siis tulos murto-osassa alkuperäisestä ajasta. Käytännössä nopeutus on vielä tätäkin suurempi, koska esimerkissä datan määrää vähennettiin tarkoituksella. Datan määrän lisääntyminen ei pidennä alkuperäisen hitaan kyselyn suoritusaikaa lineaarisesti, vaan joissakin tapauksissa jopa eksponentiaalisesti.

8.3 Kyselyn hakuperiaatteen vaihtaminen tehostaa hakua

Tarkasteltavassa esimerkissä halutaan hakea kaikkien niiden henkilöiden nimet, joilla on hakuhetkellä (kyselyssä `current_timestamp`) tallennettuna Korppi-järjestelmässä voimassaoleva virkanimike tiettyyn organisaatioon (kyselyssä `pr.organisationid=3`). Nimikkeen voimassaolo hakuhetkellä päätellään kenttien `pr.startdate` ja `pr.enddate` arvojen avulla, joiden lisäksi totuusarvotyyppiseen kenttään `pr.approved` tulee olla tallennettuna arvo tosi (engl. *true*) nimikkeen virallisuuden merkkinä. Kummastakin mainitusta päivämääräkentästä saa arvo puuttua, jolloin nimikkeen alkamisaikaa tai päättymisaikaa ei ole rajoitettu. Kysely on alkuperäisessä muodossaan seuraavanlainen:

```
SELECT p.personid, p.lastname, p.callname
FROM person AS p
WHERE p.deleted=false AND EXISTS (
  SELECT true FROM prefix AS pr, prefixtype AS prt
  WHERE pr.prefixtypeid=prt.prefixtypeid
  AND pr.deleted=false AND prt.deleted=false
  AND p.personid=pr.personid AND pr.approved=true
  AND (pr.startdate IS NULL OR pr.startdate <= current_timestamp)
  AND (pr.enddate IS NULL OR current_timestamp <= pr.enddate)
  AND pr.organisationid = 3
)
ORDER BY p.lastname, p.callname, p.personid;
```

Tällaisenaan kyselyn suorittaminen **kestää testiaineistolla 45–50 sekuntia**, mikä on käytännön tilanteessa aivan liian pitkä aika.

Tutkitaanpa tarkemmin kyselyä `EXPLAIN ANALYZE` -tekniikan keinoin (ks. luku 6.3). Kyselyä suoritettaessa luetaan rivejä kolmesta taulusta, ja jokaisesta taulusta tiedot luetaan peräkkäishauilla (engl. *sequential scan*). Peräkkäishaut toistetaan testiaineistolla vieläpä lähes 23000 kertaa. Kysely kuluttaa aikansa lähinnä `prefix`-taulun läpikäymisessä alla olevan `EXPLAIN ANALYZE` -tulosteen koroste-

tun `Nested Loop` -osan tulosteen mukaisesti (22863 läpikäyntiä ja 2.09 ms kutakin läpikäyntiä kohden kuluttaa aikaa yhteensä 47783 ms).

```
Sort  (cost=2121782.95..2121811.52 rows=11428 width=24)
 (actual time=47882.92..47882.95 rows=83 loops=1)
  Sort Key: lastname, callname, personid
  ->  Seq Scan on person p  (cost=0.00..2121012.68 rows=11428 width=24)
  (actual time=636.17..47882.30 rows=83 loops=1)
        Filter: ((deleted = false) AND (subplan))
        SubPlan
          ->  Nested Loop  (cost=0.00..91.95 rows=1 width=8)
          (actual time=2.09..2.09 rows=0 loops=22863)
                Join Filter:
                  ("outer".prefixtypeid = "inner".prefixtypeid)
                ->  Seq Scan on prefix pr
                  (cost=0.00..89.48 rows=1 width=4)
                  (actual time=2.09..2.09 rows=0 loops=22863)
                      Filter: ((deleted = false) AND ($0$ = personid)
                      AND (approved = true)
                      AND ((startdate IS NULL)
                        OR ((startdate)::timestamp with time zone
                          <= ('now'::text)::timestamp(6) with time zone))
                      AND ((enddate IS NULL)
                        OR (('now'::text)::timestamp(6) with time zone
                          <= (enddate)::timestamp with time zone)))
                ->  Seq Scan on prefixtype prt  (cost=0.00..1.74 rows=59
                      width=4) (actual time=0.00..0.02 rows=14 loops=83)
                      Filter: (deleted = false)
```

Yksinkertaisin ehdotus mahdollisesta tehostuskeinosta kyselyn osalta on se, että `prefix`-tauluun luodun indeksin (ks. luvut 5.6 ja 5.7) avulla kyselyn suoritusta saadaan nopeutettua. Testataan seuraavaa `prefix`-taulun `personid`-, `startdate`- ja `enddate`-kentät sisältävää indeksiä:

```
CREATE INDEX prefix_person ON prefix (personid, startdate, enddate);
```

Indeksi auttaa kyselyn suorittamista PostgreSQL:n versioilla 7.1 ja 7.3, mutta poikkeuksellisesti ei versiolla 7.4. Ensinmainituilla suoritusaika on 250-300 ms, mutta viimeksimainitulla jopa 2500 millisekuntia! Ero johtuu siitä, että versiolla 7.3 kysely suoritetaan lukemalla `prefix`- ja `prefixtype`-tauluja ”rinnakkain”, sekä vasta lopuksi yhdistetään näiden lukemisesta saadut rivit. Versiolla 7.4 kysely taasen suoritetaan siten, että `prefixtype`-taulun riveihin yhdistetään `prefix`-

taulusta saadut rivit ja vasta sitten tarkistetaan `prefixtype`-taulun ehtojen määräämät rivit. Jokaisen `EXPLAIN ANALYZE` -tulosteen `prefixtype`-osan tarkistukseen tuodun rivin tarkistus PostgreSQL:n versiolla 7.4 kestää 0.112 ms, ja koska näitä tarkistuksia tehdään lähes 23000 kertaa, kuluu kyselyn suorittamisaika lähes kokonaisuudessaan tässä kohdassa (22863 läpikäyntiä ja 0.112 ms kutakin läpikäyntiä kohti kuluttaa aikaa yhteensä 2560 ms). PostgreSQL:n versioilla 7.1 ja 7.3 vastaavaan tarkistukseen kuluu kymmenesosa ajasta, joten kyselyn kokonaisaikakin on kymmenesosa eli noin 250 ms.

Koska kuitenkin huomattiin luodusta indeksistä olevan hyötyä, tutkitaan voitaisiinko indeksin määrittelyn muokkauksen kautta saada se tehokkaaksi kaikilla PostgreSQL:n versioilla. Koska `prefixtypeid`-kentässä oleva viite-eheys on yhdistävä tekijä `prefix`- ja `prefixtype`-taulujen välillä, kokeillaan lisätä indeksiin kyseinen kenttä aiempien kenttien perään seuraavasti:

```
DROP INDEX prefix_person;
CREATE INDEX prefix_p_dates_prtype
ON prefix (personid, startdate, enddate, prefixtypeid);
```

Kysely suoritetaan edelleen samoin kuin aiemminkin kokeillulla indeksillä, joten `prefixtypeid`-kentän lisäyksestä indeksiin ei ollut hyötyä ainakaan tällä aineistolla testattuna. Tarvitaan siis toisenlainen lähestymistapa, jos kysely halutaan suorittaa tehokkaasti kaikilla käytössä olevilla tietokantapalvelimen versioilla. Toisaalta ei välttämättä ole järkeä yrittää tukea vanhoja tietokannanhallintajärjestelmän versioita, koska uudemmat versiot suorittavat kyselyitä yleensä muutenkin tehokkaammin kuin vanhat versiot.

Muokataan kyselyä siten, ettei siinä enää käytetä alikyselyä, jolloin kysely on seuraavanlainen:

```
SELECT p.personid, p.lastname, p.callname
FROM person AS p, prefix AS pr, prefixtype AS prt
WHERE p.deleted=false AND pr.deleted=false AND prt.deleted=false
AND pr.prefixtypeid=prt.prefixtypeid
AND p.personid=pr.personid AND pr.approved=true
AND (pr.startdate IS NULL OR pr.startdate <= current_timestamp)
AND (pr.enddate IS NULL OR current_timestamp <= pr.enddate)
AND pr.organisationid = 3
ORDER BY p.lastname, p.callname, p.personid;
```

Nyt huomataan, että kysely ei palautakaan enää samaa määrää rivejä kuin aiemmin. Tämä johtuu siitä, että muutamilla henkilöillä on hakuajankohtana voi-

massa useampia nimikkeitä. Lisätään siis `DISTINCT`-määre (ks. luku 5.4) `SELECT`-lausekkeeseen, jotta saadaan kultakin henkilöltä vain yksi tulosrivi. Nimikkeiden tietoja voimassaolopäivämäärineenhän ei haluttu mukaan lopulliseen tulokseen `SELECT`-lausekkeessa, joten tämä ratkaisu käy tässä tapauksessa mainiosti.

Toinen huomio on se, että kysely suoritetaan nyt huomattavasti nopeammin kuin edellisissä tehostamiskokeiluissa. `EXPLAIN ANALYZE` -tulosteista nähdään, että **PostgreSQL on vaihtanut kyselyn suorittamistapaa oleellisesti**. Muokatussa haussa haetaankin ensin `prefix`- ja `prefixtype`-tauluista tarvittavat rivit, ja vasta lopuksi yhdistetään tämä tulos `person`-tauluun. Aiemmin tietokantapalvelin **pakotettiin** tutkimaan jokainen `person`-taulun rivi ja päättämään, kuuluuko se lopulliseen tulosjoukkoon alikyselyn tietojen perusteella. Siirtämällä alikyselyssä olleet ehdot pääkyselyyn annettiin samalla tietokantapalvelimelle vapaus suorittaa kysely parhaalla mahdollisella tavalla.

Kolmanneksi huomataan, että edellä luotua indeksiä ei käytetä lainkaan! Sen sijaan käytetään `person`-taulun `personid`-perusavaimeen automaattisesti luotua `person_pkey`-nimistä indeksiä. `prefix`- ja `prefixtype`-taulut luetaan peräkkäishaulla läpi vain kerran, ja `person`-taulunkin indeksiä luetaan testiaineistolla ainoastaan 95 kertaa eli niin monta kertaa kuin `prefix`-taulusta löytyi ehtoihin sopivia voimassaolevia nimikkeitä. **Näin suoritusajat ovat kaikilla käytössä olevilla PostgreSQL:n versioilla vain muutamia millisekunteja.**

Turha edellä viimeksi luotu indeksi voidaan poistaa, koska siitä ei ainakaan luvun muokatussa esimerkkihaussa ole mitään hyötyä. Kyselyssä kohdistetaan hakuehto `prefix`-taulun `organisationid`-kenttään, joten tähän kenttään kannattaa vielä kokeilla indeksiä seuraavasti:

```
DROP INDEX prefix_p_dates_prtype;
CREATE INDEX prefix_orgid ON prefix (organisationid);
```

Kyseinen yhden kentän sisältävä indeksi pudottaa esimerkin muokatun haun suoritusajan vielä noin kuudesosaan, joten indeksin käyttöönotto saattaa olla kannattavaa riippuen siitä, hyödyttääkö indeksi myös muita kyselyitä.

Esimerkin opetuksena voidaan todeta, **ettei alikyselyitä pidä välttämättä käyttää, vaikkei pääkyselyssä niistä tarvitakaan mitään tietoja**. Alikyselyiden käyttö yleensä pakottaa tietokantapalvelimen suorittamaan kyselyn tietyssä järjestyksessä, joka ei tehokkuuden kannalta välttämättä ole paras mahdollinen ratkaisu.

8.4 Käyttäjien nimet ja sähköpostiosoitteet ulkoliitoksella

Korppi-järjestelmässä halutaan varsin usein hakea **tietyn käyttäjäjoukon sukunimet, etunimet sekä sähköpostiosoitteet**. Haun kohdejoukkona on käytännössä aina vain pieni osa Korppi-käyttäjistä, jolloin kyselyyn on sisällytetty vaikkapa nimihakuehtoja.

Koska sähköpostiosoite ei ole käytännössä pakollinen Korppi-järjestelmässä, tulee käyttäjälista sähköpostiosoitteineen hakea käyttäen SQL:n ulkoliitosta eli `OUTER JOIN` -rakennetta (ks. luku 5.8). Tätä rakennetta tulee käyttää toimintatapansa mukaisesti sen vuoksi, että jokainen käyttäjä saadaan listaan riippumatta siitä, onko hänellä sähköpostiosoitetta vai ei. Hakuun käytetty SQL-lause on seuraava:

```
SELECT p.personid, p.lastname, p.firstnames,
pp.personparametervalue AS email
FROM person AS p LEFT OUTER JOIN personparameter AS pp
ON (p.personid = pp.personid
   AND pp.personparametertypeid = 3 AND pp.deleted = false)
WHERE p.deleted = false;
```

Haku kestää pahimmassa tapauksessa varsin kauan (12–18 sekuntia) sillä datamäärällä, joka on tallennettuna `personparameter`-tauluun henkilöiden parametreiksi sisältäen muun muassa sähköpostiosoitteet (kyselyssä käytetty `pp.personparametertypeid = 3`), WWW-osoitteet ja Jyväskylän yliopiston JORE-opintorekisterin tietyt viralliset opiskelutiedot. Testaushetkellä `personparameter`-taulussa oli tuotantotietokannassa runsaat 766000 riviä ja testitietokannoissa hieman alle 630000 riviä.

Selkeäksi syyksi kyselyn hitauteen paljastuu ulkoliitos eli `OUTER JOIN` -rakenne. Koska nimenomaan halutaan jokainen käyttäjä eli jokainen `person`-taulun rivi, sekä tähän yhdistettynä mahdollinen `personparameter`-tauluun sijoitettu sähköpostiosoite, joutuu tietokantaohjelmisto käymään `person`-taulun lisäksi koko `personparameter`-taulun läpi mahdollisesti useitakin kertoja.

PostgreSQL:n versiot 7.1 ja 7.3 eivät ilmeisesti osaa ulkoliitoksessa käyttää tehokkaasti mitään nopeutusapuja, joten ne joutuvat käymään läpi `personparameter`-taulun jokaisen rivin ja tarkistamaan, sopiiko rivi annettuihin ulkoliitoksen ehtoihin. PostgreSQL:n versiolla 7.1 ulkoliitoshaku kesti noin 18 sekuntia ja haku ilman ulkoliitosta noin 2 sekuntia. PostgreSQL:n versiolla 7.3 mainitut haut suoritettiin hieman nopeammin kuin versiolla 7.1.

PostgreSQL:n versiolla 7.4 haut suoritetaan huomattavasti nopeammin kuin aiemmilla versioilla. Käytännössä kyseisellä versiolla ulkoliitosta käyttävä haku suoritetaan yhtä nopeasti kuin haku ilman ulkoliitosta, ja ulkoliitoshaku suoritetaan jopa nopeammin kuin PostgreSQL:n versiolla 7.3 suoritettu haku ilman ulkoliitosta.

Alla on esitetty haku ilman ulkoliitoksen käyttöä, joka ei siis täysin toteuta kyselylle asetettuja vaatimuksia.

```
SELECT p.personid, p.lastname, p.firstnames,
pp.personparametervalue AS email
FROM person AS p, personparameter AS pp
WHERE p.personid = pp.personid
AND pp.personparametertypeid = 3 AND pp.deleted = false
AND p.deleted = false;
```

Käytännössä esimerkin mukainen haku kohdistuu poikkeuksetta pieneen osaan Korppi-järjestelmään tallennetuista henkilöistä yleensä joko henkilön nimeen tai tunnistekenttään (`personid`) kohdistuvana hakuna. Tällaisen kohdennetumman haun suorittamisessa PostgreSQL osaa käyttää indeksejä, joten hakukin suoriutuu yleensä tehokkaasti. Jos kohdennettua hakua ei voida käyttää, on mahdollista jakaa haku kahteen osaan. Ensimmäisellä lyhyellä kyselyllä noudetaan ainoastaan henkilöiden perustiedot `person`-taulusta ja toisella kyselyllä haetaan sähköpostiosoitteet `personparameter`-taulusta. Lopuksi yhdistetään näiden tiedot johonkin tietorakenteeseen (vaikkapa luvussa 7.4 esitellyn `RS2`-tekniikan avulla) tai tulostetaan tiedot halutulla tavalla, jos tiedot yhdistävää tietorakennetta ei tarvita jatkokäsittelyä varten. **Toisinaan kuitenkin ainoa ratkaisu kyselyiden tehostamiseksi on ottaa käyttöön uudempi ja tehokkaampi tietokannanhallintajärjestelmä,** jos vanhempi versio järjestelmästä ei osaa tehokkaasti käyttää indeksejä tai muita nopeutusapuja hyväkseen.

8.5 Enemmän tietoa kerralla, vähemmän kyselyitä

Opettajan kurssin pääsivulla haetaan rooleittain lomakkeisiin vastanneiden lukumäärät. Rooleja ovat opiskelijat, tuntiopettajat ja opettajat. Laskennassa otetaan huomioon valintaruutujen ja radioryhmien (kyselyssä `fieldtypeid IN (2,4)`) valintoihin tehtyjen merkintöjen (kyselyssä `r.value=1`) lukumäärät.

Kurssin opiskelijoiden (`usergroupid=1`) vastanneiden lukumäärien laskenta on suoritettu lomake kerrallaan. Mikäli kurssilla on monta lomaketta, **kestää koko sivun suoritus näiden lukumäärähakujen takia pahimmassa tapauksessa jopa 15–**

20 sekuntia. Alkuperäinen erään kurssin (`courseinstanceid=7886`) yksittäisen lomakkeen (`fieldgroupid=2961`) lukumäärätiedon hakeva kysely on seuraavanlainen:

```
SELECT COUNT(*) AS count
FROM person AS p, courseparticipant AS cp
WHERE p.personid=cp.personid AND cp.courseinstanceid=7886
AND p.deleted=false AND cp.deleted=false AND cp.usergroupid=1
AND p.personid IN (
  SELECT r.personid FROM result AS r WHERE r.deleted=false
  AND r.value=1 AND r.fieldid IN (
    SELECT fieldid FROM field
    WHERE fieldgroupid=2961 AND deleted=false
    AND fieldtypeid IN (2,4)
  )
);
```

Eräällä kurssilla tämän yksittäisen kyselyn suoritus kesti tuotantotietokantapalvelimella normaalina päivänä runsaat kolme sekuntia. Samalla kurssilla on lisäksi puolen tusinaa muutakin lomaketta, joten näiden kaikkien `IN`-rakenteita käyttävien hakujen yhteisvaikutuksena **sivun suoritus kesti lähemmäs 20 sekuntia**.

`IN`-alikyselyllä toteutettu versio on sittemmin optimoitu käyttämään `EXISTS`-rakennetta ja palauttamaan lukumäärät myös muille kuin opiskelijaroolille (muokatussa kyselyssä `usergroupid IN (1,2,3)`). Kyselyn `EXISTS`-versio vie testiaineistolla aikaa noin 700 ms PostgreSQL:n versioilla 7.1 ja 7.3, ja versiolla 7.4 vain pari millisekuntia. Jostain syystä nimittäin vanhemmat versiot PostgreSQL:stä eivät käytä `result`-tauluun luotuja indeksejä, kun taas versio 7.4 käyttää niitä tehokkaasti.

Koska kyselyä ei tällaisenaan suoriteta tehokkaasti kaikilla PostgreSQL:n versioilla, tutkitaan, voidaanko hakua tehostaa kyselyä muokkaamalla. Useammalle kurssin roolille tiedot hakeva ja `EXISTS`-rakennetta käyttävä kysely on seuraavanlainen:

```
SELECT cp.usergroupid, COUNT(*) AS count
FROM person AS p, courseparticipant AS cp
WHERE p.personid=cp.personid AND cp.courseinstanceid=7886
AND cp.deleted=false AND p.deleted=false
AND cp.usergroupid IN (1,2,3) AND EXISTS (
   SELECT true FROM result AS r, field AS f
   WHERE r.deleted=false AND f.deleted=false
   AND r.value=1 AND r.fieldid=f.fieldid AND f.fieldgroupid=2961
   AND p.personid=r.personid AND f.fieldtypeid IN (2,4)
)
GROUP BY cp.usergroupid;
```

Testaustilanteessa huomataan, että ensimmäinen kyselyn suoritus kestää 1500–3000 ms, kun taas seuraavat saman kyselyn suoritukset kestävätkin tästä sadasosan eli muutaman kymmenen millisekuntia. Samanlainen tilanne tulee vastaan varsin usein muutenkin. Pahimmassa tapauksessa kuitenkin samanlaistenkin kyselyiden suoritusajat kumuloituvat, joten varaudutaan pahimpaan tilanteeseen tehostamalla kyselyä ja JSP-sivun lähdekoodia.

Mainitun JSP-sivun Java-lähdekoodi toimii periaatteessa seuraavasti:

```
lomakkeet = hae_kaikkien_lomakkeiden_perustiedot();
FOR ( lomake IN lomakkeet ) {
  vastanneet = hae_lomakkeeseen_vastanneiden_lukumäärät(lomake);
  tulosta_vastanneiden_lukumäärät(lomake, vastanneet);
  vastaukset = hae_lomakevastausten_lukumäärät(lomake);
  tulosta_lomakevastausten_lukumäärät(lomake, vastaukset);
}
```

Toiminta olisi huomattavasti tehokkaampaa, mikäli kaikkiin lomakkeisiin vastanneiden lukumäärät haettaisiin yhdellä tietokantakyselyllä. Tällöin sivu suorittaisi kyseisen ohjelmalohkon seuraavasti:

```
lomakkeet = hae_kaikkien_lomakkeiden_perustiedot();
vastaukset = hae_kaikkien_lomakkeiden_vastaukset(lomakkeet);
FOR ( lomake IN lomakkeet ) {
  lomakkeen_vastaukset = erittele_vastaukset(lomake, vastaukset)
  tulosta_vastanneiden_lukumäärät(lomake, lomakkeen_vastaukset);
  tulosta_lomakevastausten_lukumäärät(lomake, lomakkeen_vastaukset);
}
```

Muokataan SQL-kyselyä siten, että sen avulla voidaan hakea kerralla kaikkien kurssin lomakkeiden vastausten lukumäärä. Kyselyyn tarvitaan tällöin lisäksi lo-

makkeen ID-numero, jota hakutuloksen rooli ja lukumäärä koskee. Kysely muotoutuu seuraavanlaiseksi:

```
SELECT fg.fieldgroupid, cp.usergroupid, COUNT(*) AS count
FROM person AS p, courseparticipant AS cp,
courseinstance AS ci, fieldgroup AS fg
WHERE p.personid=cp.personid AND cp.courseinstanceid=7886
AND cp.courseinstanceid=ci.courseinstanceid
AND fg.courseinstanceid=ci.courseinstanceid AND fg.deleted=false
AND ci.deleted=false AND cp.deleted=false AND p.deleted=false
AND cp.usergroupid IN (1,2,3) AND EXISTS (
  SELECT true FROM result AS r, field AS f
  WHERE r.deleted=false AND f.deleted=false AND r.value=1
  AND r.fieldid=f.fieldid AND f.fieldgroupid=fg.fieldgroupid
  AND p.personid=r.personid AND f.fieldtypeid IN (2,4)
)
GROUP BY fg.fieldgroupid, cp.usergroupid;
```

Kyselyn pääkyselyyn lisättiin siis `courseinstance`- ja `fieldgroup`-taulut, jolloin saadaan palautettua myös lomakkeen tunniste (`fieldgroupid`). Myös `GROUP BY` -osaan tarvitaan kyseinen lomakkeen tunniste (ks. `SELECT`-lausekkeen selitys luvussa 5.3), koska hakutulos halutaan ryhmitellä roolin lisäksi lomakkeen tunnisteen perusteella. **Nyt kurssin kaikkien lomakkeiden vastauslukumäärät selviävät yhdellä tietokantakyselyllä**, jonka suoritusaika on sama kuin aiemmin kutakin yksittäistä lomaketta kohti suoritetun kyselyn suoritusaika eli 200–300 ms.

`EXPLAIN ANALYZE` -tulosteista huomataan, että `fieldgroup`-taulun läpikäymiseen käytetään peräkkäishakua. Ehtona taulusta haettaessa on `courseinstanceid`-kenttä, joten luodaan indeksi kyseiseen kenttään seuraavasti:

```
CREATE INDEX fieldgroup_cid ON fieldgroup (courseinstanceid);
```

Kyseistä indeksiä käytetään ainoastaan PostgreSQL:n versiolla 7.4, joka suoriutuukin koko kyselystä varsin kiitettävässä 22 millisekunnissa. Myös PostgreSQL:n versiot 7.1 ja 7.3 suoriutuvat hausta melko nopeasti, 50–80 millisekunnissa. Käyttöön on otettu tämä versio kyselystä.

Luvuissa 8.2 ja 8.3 huomattiin, että alikyselyt pääkyselyyn yhdistämällä saatetaan huomattavasti nopeuttaa kyselyn suorittamista. Kokeillaan kyselystä vielä versiota, jossa alikysely on yhdistetty pääkyselyyn seuraavasti:

```
SELECT fg.fieldgroupid, cp.usergroupid, COUNT(*) AS count
FROM person AS p, courseparticipant AS cp, courseinstance AS ci,
fieldgroup AS fg, result AS r, field AS f
WHERE p.personid=cp.personid AND cp.courseinstanceid=7886
AND cp.courseinstanceid=ci.courseinstanceid
AND fg.courseinstanceid=ci.courseinstanceid
AND fg.deleted=false AND ci.deleted=false AND cp.deleted=false
AND p.deleted=false AND cp.usergroupid IN (1,2,3)
AND r.deleted=false AND f.deleted=false AND r.value=1
AND r.fieldid=f.fieldid AND f.fieldgroupid=fg.fieldgroupid
AND cp.personid=r.personid AND f.fieldtypeid IN (2,4)
GROUP BY fg.fieldgroupid, cp.usergroupid;
```

Muokattu kysely toimii edelleen erittäin nopeasti PostgreSQL:n versiolla 7.4, kun taas vanhemmilla PostgreSQL-versioilla suoritusaika kasvaa vähintään pariinsataan millisekuntiin ja versiolla 7.1 jopa useisiin sekunteihin. Tämä johtunee siitä, että vanhemmat PostgreSQL:n versiot eivät ole kovin tehokkaita silloin, kun kyselyn sisäisesti käsiteltävät tietomäärät ovat suuria. Tässäkin tietomäärät ovat erittäin suuria, koska jo pelkästään `courseparticipant`- ja `result`-taulujen rivimäärien takia on kyselyn aikana käsiteltävänä satoja miljardeja rivejä.

Tätä viimeisintä versiota kyselystä ei ole otettu käyttöön, koska siitä ei ole käytännössä vastaavaa hyötyä. Samalla kyselyn rakenne ja ymmärrettävyys muuttuisi hieman epäselvemmäksi, koska alikyselyllä on alunperin selkeä tehtävä palauttaessaan tiedon vain siitä, onko lomakkeella merkintöjä vai ei.

Aina ei siis ole mahdollista tai hyödyllistä pelkästään tietokantakyselyä tai tietokannan ominaisuuksia muokkaamalla tehostaa toimintaa. Pelkkä kyselyiden muokkaaminen tai indeksien lisääminen ei aina ole järkevääkään, koska muillakin optimointikeinoilla, vaikkapa algoritmeja tehostamalla, voidaan saada huomattavia tehokkuusparannuksia. Luvun esimerkissä jouduttiin **indeksin määrittelyn lisäksi muokkaamaan SQL-kyselyä ja erityisesti JSP-sivun lähdekoodin hakualgoritmia**, jolloin yhteisvaikutuksena suoritusnopeutta saatiin kasvatettua oleellisesti.

8.6 Isojen tulosjoukkojen käsittely

Muun muassa Suomen Virtuaaliyliopiston palvelin (`http://www.vy.fi/`) hakee Korppi-järjestelmästä kurssitietorajapinnan kautta kurssitietoja aktiivisista kursseista. Rajapinnan tuottama haku ilman rajoitteita tuottaa varsin massiivisen kuorman Korppi-palvelimelle sekä datan määrässä että resurssien kulutuksessa.

Alunperin rajapintasivu haki ensin niiden kurssien perustiedot, joista ollaan rajapintahakuun annettujen hakurajoitteiden perusteella kiinnostuneita. Kurssien perustiedot hakevan SQL-kyselyn jälkeen haettiin erillisillä lyhyillä kyselyillä tarkemmat lisätiedot niistä kursseista, jotka rajapinnan kautta halutaan lopulta tulostaa. Kun haluttujen kurssien tunnisteet (`courseinstanceid`) on saatu perustiedot hakevan kyselyn tuloksena, opetusryhmien tarkat tiedot `eventgroupd`-taulusta hakeva kysely oli seuraavankaltainen:

```
SELECT eventgroupid, name, comment
FROM eventgroup
WHERE courseinstanceid IN (i,j,k)
AND muut ehdot;
```

Tällainen ehtorakenne toimii silloin, kun haluttuja kursseja on vähän, eli `courseinstanceid`-lista `IN`-ehdossa on lyhyt. Kun rajapinnan kautta halutaan hakea vaikkapa tiedekunnan koko lukuvuoden opetus, kasvaa `courseinstanceid`-lista jopa muutaman sadan alkion kokoiseksi. Tällöin kyselyn suoritusaika voi olla jopa kymmeniä minuutteja ja on siis tarvetta nopeuttaa kyselyn suorittamista.

Kyselyä voidaan tehostaa neljällä tavalla:

1. Suoritetaan kaikki kysely(t) jokaiselle kurssille erikseen. Yksittäiset kyselyt ovat tällöin lyhyitä ja nopeita, mutta niitä on paljon.

2. Muokataan kyselyn hitaita ehtoja tarpeen mukaan `IN`-, `EXISTS`- tai `BETWEEN`-rakenteilla (ks. luku 5.4) käytettäväksi.

3. Laajennetaan kyselyä siten, että mukana on lopputuloksen kannalta tarpeettomiakin tietoja. Keskusmuistia käytetään ylimääräisten tietojen tallentamiseen.

4. Jätetään pois koko hidastava ehto ja jatketaan tavan 3 mukaan. Muistiresursseja vaaditaan huomattavasti enemmän.

Tapa 1 voidaan jättää harkinnasta lähes välittömästi, koska tämä tuottaisi runsaasti tietokantakyselyitä ja kuormittaisi siten palvelimia tarpeettomasti sekä olisi

lisäksi käytännössä varsin tehotonta. Tapa 2 saattaa toimia, mutta kysely voi silti olla varsin hidas, koska aikaa edelleen kuluu ehtolistojen käsittelyyn. Lisäksi toisinaan ei näilläkään tehostamiskeinoilla saada aikaan tyydyttävää lopputulosta. Tavan 3 käyttöönottamiseksi pitää varmistaa tietojen hyödyntämisvaiheessa huomioitavan vain alunperin halutut kurssit, jolloin ylimääräiset kurssit voidaan jättää huomiotta.

Tavassa 4 toimitaan kuten tavassa 3, mutta tässä hakujen tuloksena on saatu ylimääräisiä tietoja huomattavan paljon ja siten tarvitaan oleellisesti enemmän muistiresursseja. Tavassa 4 käytettävä kysely saattaa silti suoriutua hieman nopeammin kuin tavassa 3 käytettävä kysely, koska hitaimman ehdon mukaisia rajoitteita ei tietokantapalvelimen tarvitse tutkia. Muistinkulutus on ratkaiseva tekijä, sekä monenko kurssin tiedot lopulta rajapinnan kautta ulos halutaan verrattuna siihen, monenko kurssin tiedot tietokannasta on haettavissa.

Kurssitietorajapinnassa otettiin käyttöön tapa 3. Koska ensimmäisenä tietokantakyselynä haetaan rajapintahaussa annettujen parametrien perusteella kohteena olevien kurssien tunnistetiedot, saattaa jatkokyselyiden tuloksena tulla ylimääräisten kurssien tietoja. Kurssitietojen tulostussilmukassa ollaan kiinnostuneita kerrallaan ainoastaan yhdestä kurssista. Tällöin tietojen hyödyntämisen kannalta on samantekevää, onko tietorakenteissa vain kyseisellä hetkellä tarvitun yhden kurssin tiedot vai vaikkapa miljoona ylimääräistä tietoriviä. Joka tapauksessa tarvittavan kurssin tiedot joudutaan etsimään muistissa olevasta tietorakenteesta yksi kerrallaan. Tehostamisapuna voitaisiin muistissa olevista tietorakenteista poistaa jo hyödynnetyt tiedot sekä ne tiedot, joita varmasti ei lainkaan tulla tarvitsemaan. Tiedossahan koko ajan ovat niiden kurssien tunnisteet, joista ollaan kiinnostuneita ja joiden tarkat tiedot jossain vaiheessa tulostetaan rajapinnan kautta.

Kyseisen rajapintasivun tietokantahakuja muutettiin tehokkaammiksi siten, että kurssien tunnisteet sisältävästä hakutuloksesta otetaan talteen sekä pilkuilla erotettu lista `courseinstanceid`-arvoista että kyseisten arvojen minimi ja maksimi. Jos haluttuja kursseja on vähemmän kuin jonkin tietyn kokeilemalla määritellyn raja-arvon mukainen määrä, käytetään sivun suorittamissa lopuissa tietokantahauissa `IN`- tai `BETWEEN`-listaa tunnisteista. Jos kursseja taasen on vähintään raja-arvon mukainen määrä, käytetään hauissa pitkien listojen sijasta ainoastaan edellä saatuja minimiä ja maksimia. Tällä tavalla toimimalla tietokantahaut toimivat suhteellisen nopeasti suurellakin joukolla kursseja. Keskusmuistia tosin kuluu enemmän, koska haut tuottavat ehkä runsaastikin ylimääräisiä kurssitietoja. Tulostussilmukan kannalta on samantekevää, kumpaa tapaa on edellä käytetty. Tulostusvuorossa olevan

kurssin tiedot joudutaan joka tapauksessa etsimään lisätiedot sisältävistä tietorakenteista erikseen, mutta tämä ei ole kovin hidasta tietojen ollessa jo valmiina muistissa.

Muokattua tekniikkaa voidaan nopeuttaa siten, että muistissa pidettävät tietorakenteet pidetään kurssien tunnisteiden eli `courseinstanceid`-arvojen mukaisessa järjestyksessä, mutta kurssitietojen etsintää jatketaankin tietorakenteen alun sijaan siitä kohdasta, johon viimeksi päädyttiin. Mikäli käsiteltävän kurssin tiedot löytyvät ID-arvojen järjestyksen mukaisesti kurssitietorakenteesta ennen viimeksi käsiteltyä ID-arvoa, tulee etsinnän aluksi siirtyä tietorakenteen alkuun. Muussa tapauksessa voidaan tavanomaiseen tapaan jatkaa etsimistä tietorakenteen loppuun. Näin käydään keskimäärin enintään puolet tulosjoukon riveistä läpi kullakin hakukerralla.

Edelleen voitaisiin tietojen hakuvaihetta tehostaa siten, että kyselyiden hakutuloksia läpikäytäessä haetaan niistä halutut tiedot valmiiksi tietorakenteisiin lopullista tulostusta varten. Näin käytännössä jokainen hakutulos käytäisiin läpi ehkä vain kerran. Tämä tapa tosin aiheuttaisi ehkä huomattavastikin lisävaatimuksia muistin määrälle, koska kaikkien kurssien tiedot olisivat muistissa samanaikaisesti vähintään kahteen kertaan eri tietorakenteissa. Välttämättä nopeushyötyäkään tästä ei saataisi, koska muistissa olevan tietorakenteen läpikäyminen tietojen hyödyntämisvaiheessa ei kestä kauaa.

Edellä kuvattua tehostamistapaa on käytetty kurssitietorajapinnassa muidenkin isojen tulosjoukkojen hakemisessa tietokannasta. Käytännössä jokaisessa SQL-tietokantakyselyssä on kurssien määrästä riippuen joko yksi `IN`-lista sisältäen kaikki kohteena olevat kurssien ID-arvot tai yksi `BETWEEN`-ehto hyödyntäen arvoalueen minimiä ja maksimia. Näillä toimenpiteillä kurssitietorajapinta palauttaa haluttujen kurssien tiedot siedettävässä ajassa. Entisten kymmenien minuuttien sijasta sama kurssitietodata palautetaan nyt muutamassa minuutissa, josta tietokantayhteys on auki alle puoli minuuttia. Kuitenkin tuona aikana on tietokannasta ehditty hakea kymmenisen tulosjoukkoa, joissa enimmillään on ollut satoja tuhansia rivejä. Loppuaika kuluu datan tulostamiseen rajapinnan kautta sekä tiettyjen Java-olioiden luomiseen, jotka alustavat omat tietonsa itsenäisesti.

8.7 Havaintoja ja suosituksia

Tietokannan hyödyntämisen tehostaminen vaatii tietokannan valvojalta ja tietokantaa hyödyntävän sovelluksen kehittäjältä varsin usein **pitkäjänteisyyttä ja kärsivällisyyttä**. Ongelman paikallistaminen suuresta määrästä lähdekoodia ja SQL-kyselylauseita saattaa olla haastava tehtävä. Ongelman **paikallistamisessa apuna** toimivat lähdekoodin tarkastelun ja sovelluskehitysympäristön virheenjäljittimen ominaisuuksien lisäksi mm. tietokannanhallintajärjestelmän keräämät **käyttötilastot tietokannan hyödyntämisestä** (ks. luku 6.2) sekä sovellusten kirjoittamat **lokitiedostot** (ks. luku 6.5).

Kun ongelmallinen kohta on löydetty vaikkapa lähdekoodista, tulee selvittää **ongelman laajuus**. Ilmeneekö se aina riippumatta tietokannan datan määrästä, sovelluksen käyttäjästä sekä käyttäjän ja ympäristön asetuksista? Jos ongelma esiintyy vain harvoilla käyttäjillä tai vain tietyissä harvinaisissa tilanteissa, tulee punnita se, onko tehokkaamman ratkaisun löytäminen ehdottoman välttämätöntä, vai voisiko sovelluksen kehittäjä keskittyä oleellisempiin kehitettäviin kohteisiin. Kuten luvussa 6.4 todettiin, viidesosa sovelluksen sisältämistä kyselyistä ja päivityksistä tuottaa 80 prosenttia kaikista tietokantaan kohdistuvista operaatioista tarkasteltaessa suurta määrää tietokantaoperaatioita. Täten keskittämällä sovelluksen **kehityksen voimavarat oleellisiin kohtiin** saadaan todennäköisesti suhteellisesti eniten hyötyä ja sovellukseen tehokkuutta.

Kun ongelmallista yksittäistä tehotonta SQL-kyselyä on päädytty tutkimaan tarkemmin, kannattaa se useimmiten kopioida tekstieditoriin, jonka avulla erilaiset tehostamiskokeilut ja toteutusratkaisuiden versiot kommentteineen saadaan kätevästi talteen. Ensimmäisenä kannattaa alkuperäinen kysely kopioida tekstieditorissa ns. työversioksi, jonka muokattujen versioiden avulla kokeillaan nopeutusratkaisuja. Alkuperäinen versio kyselystä jätetään näkyville vertailukohdaksi ja uusien kokeiluversioiden pohjaksi. Käytettäessä PostgreSQL-tietokannanhallintajärjestelmää kannattaa kyselylauseen ensimmäiseksi riviksi jo tässä vaiheessa kirjoittaa luvun 6.3 mukaisesti **`EXPLAIN ANALYZE` -määrite**. Tällöin kyselyn suorituksesta näytetään aina tarkka suorittamistapa, eikä pelkästään kyselyn lopputulosta ja suorittamiseen kulunutta aikaa. Joissakin tietokannanhallintajärjestelmän käyttöliittymissä on mahdollista suorittaa kysely mainitussa `EXPLAIN ANALYZE` -moodissa käyttöliittymän omin keinoin, joten kyseisiä apukeinojakin kannattaa hyödyntää.

SQL-kyselyissä on usein mukana tauluja, joiden mukanaolo ei vaikuta varsinaisen ongelman käyttäytymiseen juuri millään tavalla. Ylimääräisten epäoleellisten

taulujen mukanaolo kyselyssä vaikeuttaa ongelman hahmottamista ja analysointia, vaikkapa monimutkaistamalla `EXPLAIN ANALYZE` -moodissa esitettyä kyselyn suorittamistapaa. Muokattavasta SQL-kyselystä kannattaa siten **poistaa epäoleelliset osat** yksi kerrallaan testaten aina välillä, että tehottomuusongelma on edelleen olemassa lyhyemmälläkin SQL-kyselyllä. Ongelmallinen useita tauluja hyödyntävä SQL-kysely saattaa tällöin lyhentyä muutamaa taulua hyödyntäväksi kyselyksi, josta saattaa olla helpompi nähdä ongelman syy ilman jatkotestauksiakin.

Tarkemman analysoinnin tuloksena saatetaan päätyä vaikkapa siihen, että tietokannan tauluihin tulisi luoda tietojen hakua nopeuttavia indeksejä (ks. luvut 5.6, 5.7 ja 8.1). Tämä tarve saattaa näkyä kyselyn suorittamistavan analysointitulosteessa vaikkapa peräkkäishakujen (engl. *sequential scan*) käyttönä. Kyselyiden suorittamista tehostavien **indeksien löytäminen** onnistuu valitettavasti useimmiten vain **yleisten nyrkkisääntöjen ja tietokantoihin liittyvän kokemuksen tuoman tietämyksen avulla**. Ongelmaa on hyvä tarkastella ensin nyrkkisääntöjen pohjalta (ks. luku 5.7), jolloin jonkin yleisesti käytetyn tehostamisratkaisun avulla saattaa tarkasteltuun ongelmaan löytyä suhteellisen helposti riittävästi nopeuttava ratkaisu.

Jos indeksien luonti ei riitä nopeuttamaan hakua riittävästi, toisinaan joudutaan **muokkaamaan SQL-kyselyä tehokkaammaksi** (ks. luku 7.1). Tämä onnistuu vaikkapa siirtämällä kyselyssä käytetty alikysely osaksi pääkyselyä (ks. luvut 8.2 ja 8.3) taikka jakamalla iso kysely yksinkertaisempiin ja selkeämpiin osiin alikyselyiden avulla. Kukin toteutusratkaisun versio kannattaa dokumentoida etuineen ja haittoineen, koska tarkastelusta saattaa olla hyötyä myöhemminkin datan määrän lisääntyessä ja tehottomuusongelmien ilmetessä uudelleen.

Jos SQL-kyselyn optimointi ei nopeuta toiminnon suorittamista tarpeeksi, on mahdollista **muokata sovelluksen lähdekoodin algoritmeja ja tietorakenteita tehokkaammiksi** (ks. luvut 8.5 ja 8.6). Tällöin vaikkapa iso ja monimutkainen SQL-kysely voidaan yleensä jakaa suoritettavaksi pienemmissä palasissa, joiden tulokset kootaan varsinaiseksi lopputulokseksi sovelluksen lähdekoodin keinoin (ks. luku 7.2). Myös tehokkaampia tietorakenteita ja tietokantakyselyiden välimuistiratkaisuja voidaan hyödyntää lähdekoodia muokkaamalla (ks. luvut 7.3 ja 7.4).

Yhtenä nopeutusvaihtoehtona kannattaa useimmiten joka tapauksessa kokeilla kyselyä, josta on poistettu alikyselyt ja mahdolliset muut tietokannanhallintajärjestelmän kyselyoptimoijaa määräävät rajoitukset. Tällöin **kyselyoptimoijalle annetaan mahdollisimman suuri vapaus** suorittaa kysely tehokkaimmaksi päättelemällään tavalla.

Ennen kuin toimivaksi todettu kyselyn suorittamista nopeuttava toteutusratkaisu löytyy, joudutaan tehostamisen aikana kyselyn monimutkaisuudesta riippuen usein **kokeilemaan monia erilaisia vaihtoehtoja**, jotka saattavat poiketa toisistaan vain vähän. Kuitenkin **pientenkin muutosten toteuttaminen** lähdekoodiin, SQL-kyselyyn tai taulujen indekseihin **saattaa vaikuttaa oleellisesti kyselyn suoritusnopeuteen**. **Kyselyoptimoijalla tulee kokeilujen aikana olla aina käytössään ajantasainen tieto** taulujen sisältämästä datasta (ks. luku 6.3), jotta kyselyn suorittamistavan optimointi perustuu oikeisiin ja välttämättömiin tietoihin datan jakautumisesta.

Toisinaan on kokeilujen aikana mahdollista löytää monta erilaista versiota, joista jokainen tehostaa oleellisesti kyselyn suoritusnopeutta, mutta joista kukin suoritetaan sisäisesti eri tavalla. Tällaisessa tapauksessa tulee punnita kunkin ratkaisun edut ja haitat, vaikkapa kyselyiden suorittamisen aikana tarvittava kokonaismuistimäärä ja muu resurssien kulutus, tietokannanhallintajärjestelmän asetusten ja kyselyoptimoijan tehokkaiksi päättelemien suoritussuunnitelmien vaikutus sekä datan määrän lisääntymisen vaikutus. **Toteutusratkaisuksi yleensä tulee erilaisista tehokkaista versioista valita se, joka todennäköisesti jatkossakin on tehokkain.** Muut löydetyt nopeuttavat ratkaisut on hyvä kirjata muistiin, jotta jatkossa niitä voidaan tarvittaessa hyödyntää, jos tehostamista kyseisessä kohdassa tai jossain muualla vaaditaan.

Useimmiten tietokannan taulujen sisältämä **datan määrä lisääntyy**, mutta ei suhteessa yhtä paljon jokaisessa taulussa. Siksi erityisen tärkeää on selvittää ne taulut, joiden datamäärä todennäköisesti lisääntyy eniten jatkossa. Tällaisiin tauluihin liittyvät SQL-kyselytkin yleensä hidastuvat suhteellisesti eniten. Jos SQL-kyselyssä luetaan dataa vaikkapa kahdesta taulusta, joista kummankin datamäärä lisääntyy tietyn ajan kuluessa kymmenkertaiseksi, tulee SQL-kyselyn käsiteltäväksi jo pelkästään näistä tauluista satakertainen määrä dataa entiseen verrattuna. **Datamäärän lisääntyminen saattaa vaikuttaa oleellisesti kyselyn suorittamistapaan**, jolloin vaikkapa tiettyjä aiemmin kyselyn suorittamista nopeuttavia indeksejä ei enää käytetäkään. Tällöin kyselyn suorittaminen saattaakin yhtäkkiä muuttua todella hitaaksi ja tehottomaksi, jolloin tietokannan valvojan ja sovelluksen kehittäjän tulee reagoida nopeasti ongelmaan.

9 Jatkotutkimuksen aiheita

Mullinsin [11, s. 295] mukaan tehokkaiden SQL-lauseiden käytöllä ja järjestelmän perinpohjaisella optimoinnilla ei korvata huonosti suunnitellusta tai tehottomasti järjestellystä tietokannasta johtuvia hitausongelmia. Siksi hänen mukaansa on erityisen tärkeää, että tietokannan ja sen sisältämien kohteiden rakennetta seurataan jatkuvasti, sekä rakenteita muokataan muuttuneita tarpeita vastaaviksi. Tietokannan valvojan tulee Mullinsin [11, s. 295] mukaan olla perehtynyt tietokannanhallintajärjestelmän ominaisuuksiin siten, että hän voi tarvittaessa käyttää oikeanlaisia tekniikoita tietokannan rakenteiden tehostamiseksi.

Luvussa esitellään Mullinsin kirjassaan [11, s. 295–316] kuvaamia tekniikoita, jotka useimmissa varteen otettavissa tietokannanhallintajärjestelmissä ovat tarjolla. Tietokannan tauluja voidaan jakaa osiin, jonka lisäksi taulujen ja indeksien tallentamiseen käytetyt tiedostot voidaan fyysisesti hajauttaa eri tallennusmedioille. Denormalisoinnissa samoja tietoja tallennetaan useampaan eri tauluun tehokkuuden lisäämiseksi. Taulujen ryvästämisen avulla mm. arvovälihakujen tulokset saadaan tuotettua nopeammin, koska taulun rivit ovat valmiiksi oikeassa järjestyksessä. Datan tiivistyksen avulla tietokannan levytilan tarvetta saadaan pienennettyä, jolloin datan lukemiseenkin vaaditaan vähemmän levyoperaatioita. Tietokantahajautuksen avulla yhteenkuuluva data voidaan jakaa fyysisesti useampaan eri paikkaan. Luku perustuu pääosin lähteisiin [7, s. 261–263, 321–322 ja 589–592], [11, s. 295–316 ja 505] sekä [35, s. 3–12].

9.1 Tietokannan taulujen ositus

Tietokannan taulujen osituksen (engl. *partitioning*) avulla tietokannan valvoja voi määritellä sen, miten tietokannan taulut tallennetaan tiedostojärjestelmään tiedostoiksi. Kukin taulu voidaan tallentaa yksittäisenä tiedostona, joka on Mullinsin [11, s. 296] mukaan yleisin käyttötapa. Jokainen tauluun lisättävä rivi tallennetaan yhteen ja samaan taulun käyttämään tiedostoon.

Kukin taulu voidaan tallentaa useampana tiedostona. Tätä vaihtoehtoa käytetään Mullinsin [11, s. 296] mukaan useimmiten erittäin suurille tauluille, tai tauluil-

le, joiden data tulee olla fyysisesti eroteltuna tallennusmedialla. Toisaalta useampia tauluja voidaan Mullinsin [11, s. 296] mukaan tallentaa yhteen tiedostoon, jota tapaa käytetään useimmiten pienille tarkistustauluille (engl. *lookup table*) tai kooditauluille (engl. *code table*). Tämä tapa saattaa olla tehokas levyn hyödyntämisen näkökulmasta.

Mullinsin [11, s. 297] mukaan ositus auttaa hyödyntämään rinnakkaisuutta (engl. *parallelism*). Tällöin yksittäisen SQL-lauseen suorittamista varten voidaan käynnistää useampia yhtäaikaisia, rinnakkaisia operaatioita, jotka hyödyntävät vaikkapa eri prosessoreita, levytaltioita tai jopa useampia tietokantoja. Rinnakkaisuuden avulla voidaan mahdollisesti suurestikin lyhentää tietokantakyselyiden suoritusaikoja.

Mullinsin [11, s. 297] mukaan tietokannan tallentamiseen kannattaa hyödyntää mieluummin alustamattomia osioita (engl. *raw partition*) kuin tavanomaiseksi tiedostojärjestelmäksi alustettuja osioita. Käyttöjärjestelmä nimittäin puskuroi tietostojärjestelmän levyllekirjoitukset, joten tietokannanhallintajärjestelmä ei voi olla varma, että tietokantaan kirjoitettu tieto on todella kirjoitettu levylle asti. Jos puskuroinnin aikana ennen levyllekirjoitusta sattuu jokin toimintahäiriö, ei tiedostojärjestelmän avulla tallennettu tietokanta enää välttämättä olekaan täydellisesti takaisinpalautettavissa.

9.2 Tietokannan tiedostojen hajauttaminen

Tietokannan datan sisältävien tiedostojen sijoituspaikalla voi olla Mullinsin [11, s. 307] mukaan erittäin merkittävä vaikutus tehokkuuteen. Koska tietokanta on hänen mukaansa erittäin siirräntävaltainen (engl. *very I/O intensive*), täytyy tietokannan valvojan tehdä kaiken voitavansa minimoidakseen fyysisten levyjen luku- ja kirjoittamiskustannukset.

Mullins [11, s. 308] kehottaa mahdollisuuksien mukaan siirtämään indeksitiedostot erilleen varsinaisista taulutiedostoista, mieluusti eri fyysiselle levylle. Hänen mukaansa tehokkuus huononee, mikäli sekä indeksit että taulujen data sijaitsevat samalla levyllä. Tämä johtuu hänen mukaansa siitä, että tietokantaan kohdistuvat kyselyt toistuvasti tarvitsevat dataa sekä taulusta että taulun indeksistä. Koska fyysisen levyn lukupää joutuu levylohkoja lukeakseen liikkumaan levypinnalla, joudutaan kahden tiedoston yhtäaikaisen lukemisen aikana välillä odottelemaan toiseen tiedostoon kohdistuvan lukuoperaation päättymistä. Samasta syystä usein

yhdessä tarvittavat taulut Mullins ehdottaa eriytettäväksi eri levyillä sijaitseviin tiedostoihin.

9.3 Tietokannan denormalisointi

Luvuissa 4.6–4.10 esitellyn normalisoinnin vastakohta on denormalisointi (engl. *denormalization*). Mullinsin [11, s. 301] mukaan denormalisoinnin idea on tallentaa samoja tietoja useaan eri paikkaan, joka nopeuttaa tietojen hakua niiden muokkaamisen kustannuksella. Hänen mukaansa **ainoa syy relaatiotietokannan denormalisoinnille on tehokkuuden lisääminen**. Mullins [11, s. 138–152] tarkastelee erilaisia denormalisointimahdollisuuksia.

Etukäteen yhdisteltyjä tauluja (engl. *prejoined tables*) voidaan käyttää silloin, kun SQL-kyselyssä liitoksen tekeminen kuluttaa kohtuuttomasti resursseja. Johonkin tauluun voidaan tallentaa liikadataa (engl. *redundant data*), kun SQL-kyselyissä halutaan vähentää taulujen välisiä liitoksia usein tarvittavien tietojen osalta. Raporttitauluihin (engl. *report tables*) voidaan etukäteen tallentaa monimutkaisten SQL-kyselyiden ja vaikkapa osittain sovelluskoodilla tuotetun raportin lopputulos. Johdettavan tiedon (engl. *derivable data*) avulla alkuperäisistä lähtötiedoista on lopputulokset laskettu ja tallennettu taulun kenttään, jolloin johdettua tietoa ei aina tarvitse laskea mahdollisesti monimutkaisilla ja hitailla laskuoperaatioilla uudestaan jokaisessa käyttötilanteessa.

Peilitaulujen (engl. *mirror tables*) avulla osa tauluista kopioidaan järjestelmän eri osien käyttöön. Jaettujen taulujen (engl. *split tables*) avulla yhdelle käyttäjäryhmälle tai sovellukselle voidaan tarjota jonkin taulun osa ja toiselle käyttäjäryhmälle tai sovellukselle taulun loppuosa, jaettuna joko sarakkeittain pystysuuntaisesti (engl. *vertical split*) tai riveittäin vaakasuuntaisesti (engl. *horizontal split*). Yhdistettyihin tauluihin (engl. *combined tables*) yhdistetään yhden suhde yhteen -tyyppisiä (engl. *one-to-one relationship*) tai yhden suhde moneen -tyyppisiä (engl. *one-to-one relationship*) viite-eheyksiä käyttäviä tauluja.

Pikatauluja (engl. *speed tables*) voidaan käyttää, kun vaikkapa monimutkainen hierarkia halutaan purkaa tehokkaammin ja helpommin käytettävään muotoon taulun sisällöksi. Fyysinen denormalisointi (engl. *physical denormalization*) tarkoittaa tietokannan optimointia tietokannanhallintajärjestelmän fyysisiä ominaisuuksia silmällä pitäen. Tämä tapahtuu jakamalla vaikkapa paljon sarakkeita sisältävä taulu

useampaan osaan, jotta yhden osataulun rivi sopii kokonaisuudessaan yhden levylohkon alueelle.

9.4 Taulujen ryvästäminen

Ryvästettyyn tauluun (engl. *clustered table*) kuuluvat rivit tallennetaan Mullinsin [11, s. 302] mukaan fyysisesti levylle jonkin sarakkeen tai useiden sarakkeiden määräämässä järjestyksessä. Ryvästäminen toteutetaan yleensä ryväshakemistona (engl. *clustering index*), joka pakottaa rivit tallennettavaksi indeksoitujen sarakkeiden arvojen perusteella nousevaan järjestykseen. Kussakin taulussa voi olla vain yksi ryppääseen tai ryväshakemistoon perustuva järjestys (engl. *one clustering sequence per table*), koska fyysisesti taulun data voidaan tallentaa vain yhteen järjestykseen.

Mullinsin [11, s. 303] mukaan hyvä käytäntö on ryvästää tauluja, joita hyödynnetään peräkkäishaun (engl. *sequential access*) avulla. Ryväshakemistot siis Mullinsin mukaan tukevat parhaiten arvovälihakuja (engl. *range access*), kun taas muuten kuin ryvästämällä toteutetut hakemistot ovat parhaimmillaan satunnaishauissa (engl. *random access*). Jos tietokannanhallintajärjestelmä tukee ryvästämistä, Mullins ehdottaa nyrkkisääntönä ryväshakemiston luomista jokaiseen tauluun, jos taulu ei ole kooltaan erittäin pieni.

9.5 Datan tiivistys

Mullinsin [11, s. 306] mukaan datan tiivistyksen avulla tietokannan kokoa ja levytilan tarvetta saadaan pienennettyä. Datan tiivistämisellä pyritään siihen, että datan tallennukseen tietokannassa kuluu vähemmän levylohkoja, jolloin datan lukeminenkin vaatii pienemmän määrän levyoperaatioita. Tällöin tiettyjen algoritmien avulla data tiivistetään tietokantaan lisäysvaiheessa ja puretaan lukuvaiheessa. Tietokantaan kohdistuvat operaatiot kuluttavat tiivistetyllä datalla enemmän prosessorin resursseja, koska sen täytyy tiivistää ja purkaa dataa aina kun käyttäjät lisäävät, päivittävät ja lukevat sitä.

Mullinsin [11, s. 307] mukaan jokaista taulua tai indeksiä ei kannata tiivistää, koska pienillä datamäärillä tiivistetty data saattaakin viedä enemmän levytilaa kuin tiivistämätön data. Tämä johtuu hänen mukaansa siitä, että tiivistystekniikka vaatii sisäisesti tilastoja ja hakemistoja tiivistyksen hallinnoimista varten.

9.6 Tietokantahajautus

Özsu tarkastelee kirjassaan [35] perin pohjin tietokantahajauttamisen periaatteita, tekniikoita ja ongelmia. Özsu [35, s. 3] määrittelee hajautetun tietojenkäsittelyjärjestelmän (engl. *distributed computing system*) koostuvan itsenäisistä, mahdollisesti keskenään erilaisista tietojenkäsittely-yksiköistä (engl. *processing element*), jotka on yhdistetty toisiinsa tietokoneverkon välityksellä ja jotka toimivat yhdessä suorittaakseen niille annetut tehtävät. Yksittäinen tietojenkäsittely-yksikkö tarkoittaa Özsun [35, s. 3] mukaan tietojenkäsittelylaitetta, joka pystyy itsenäisesti suorittamaan ohjelmia.

Hajautetulla tietojenkäsittelyllä pyritään Özsun [35, s. 4] mukaan jakamaan nykypäivän suuret ja monimutkaiset ongelmat pienempiin osiin, jotka on helpompi ratkaista erillisinä (”hajoita ja hallitse”). Özsun mukaan tämä on taloudellista, koska ongelman ratkaisemiseksi vaadittavaa tietojenkäsittelykapasiteettia voidaan lisätä ottamalla käyttöön useampia tietojenkäsittely-yksiköitä. Lisäksi ohjelmistojen kehittämiseen vaadittavat kustannukset saattavat pysyä paremmin kurissa, kun kehityksessä on mukana useampia osittain itsenäisiä ryhmiä.

Luvun ensimmäisen kappaleen tietojenkäsittelyjärjestelmän hajautusmääritelmään sisältyy Özsun [35, s. 3] mukaan tietojenkäsittelylogiikan tai tietojenkäsittelyyksiköiden hajautus. Muita mahdollisia hajautettavia tekijöitä Özsun mukaan ovat toiminnot (engl. *function*), data (engl. *data*) tai suoritettavien tehtävien hallinta (engl. *control of the execution of various tasks*).

Özsu määrittelee kirjassaan [35, s. 4] **hajautetun tietokannan** (engl. *distributed database*) olevan kokoelman useammasta tietokannasta, jotka ovat loogisesti yhteenliittyviä ja jotka on kytketty toisiinsa tietokoneverkon välityksellä. **Hajautettu tietokannanhallintajärjestelmä** (engl. *distributed database management system*, DDBMS) hänen mukaansa on ohjelmisto, jonka avulla hallitaan hajautettuja tietokantoja ja jonka avulla hajautus ei näy käyttäjille (engl. *makes the distribution transparent to the users*).

Özsun [35, s. 10–12] mukaan datan itsenäisyys (engl. *data independence*), verkon läpinäkyvyys (engl. *network transparency*), replikoinnin eli datan kopioinnin läpinäkyvyys (engl. *replication transparency*) sekä tietojen osiin jakamisen läpinäkyvyys (engl. *fragmentation transparency*) ovat välttämättömiä selvitettäviä tekijöitä harkittaessa tietokantahajautuksen hyödyntämistä. Täysin läpinäkyvän hajautetun tietokannan avulla kyseistä tietokantaa hyödyntävät käyttäjät ja sovellukset voivat Özsun [35, s. 9] mukaan suorittaa tietokantaoperaatioita välittämättä siitä, missä ha-

jautetun tietokannan osat fyysisesti sijaitsevat tai miten eri fyysisissä tietokannoissa olevia tietoja päivitetään.

9.7 Tietokannanhallintajärjestelmien työkalujen vertailu

Hoffer tarkastelee kirjassaan [7, s. 261–263] suorituskyvyn viritystyökaluja (engl. *performance tuning tools*). Hänen mukaansa työkalujen pääasiallisia käyttötarkoituksia ovat suorituskyvyn hallinta (engl. *performance management*) ja suorituskyvyn optimointi (engl. *performance optimization*). Seurantatyökaluilla (engl. *performance monitor*) tietokannanhallintajärjestelmän ja tietokannan suorituskykyä voidaan seurata tosiaikaisesti tai historiatietoihin perustuen. Hyödynnettävä työkalu voidaan Hofferin [7, s. 589] mukaan määrätä lähettämään havaitusta ongelmasta hälytys tietokannan valvojalle tai jopa korjaamaan ongelma itsenäisesti.

Suorituskyvyn arviointityökaluilla (engl. *performance estimation tools*) voidaan Hofferin [7, s. 262] mukaan etukäteen arvioida SQL-lauseiden tai sovellusten suorituskykyä mm. saantipolkuihin ja toimintaympäristöön pohjautuen. Lisäksi tietämyskantoihin voidaan tallentaa vihjeitä, joiden avulla SQL-lauseita voidaan parhaassa tapauksessa automaattisesti muokata nopeampaa suorittamista varten.

SQL-analysointityökaluilla voidaan Hofferin [7, s. 591] mukaan joissakin tapauksissa analysoida vaikkapa kaikki sovelluksen hyödyntämät SQL-kyselyt graafisesti ja tekstimuodossa. Työkalu saattaa varoittaa sovelluksen kehittäjää tehottomista SQL-kyselyissä käytetyistä toteutusratkaisuista, kuten turhista järjestämisistä (`ORDER BY`, ks. luku 5.3) tai `DISTINCT`-määreiden käytöstä (ks. luku 5.4). Analysoinnin ja optimoinnin apuna voidaan hyödyntää em. tietämyskantaa, jonka avulla SQL-kyselyissä hyödynnetyille toteutusratkaisuille voidaan ehdottaa tehokkaampia vaihtoehtoja.

9.8 Tietokannanhallintajärjestelmien optimoijien vertailu

Hofferin [7, s. 321–322] mukaan optimoijien suorittama SQL-kyselyiden suorittamistavan optimointi (engl. *relational optimization*) tapahtuu kussakin tietokannanhallintajärjestelmässä hieman eri tavoin ja käyttäen erilaisia tietoja, mutta periaatteeltaan eri järjestelmät ovat samanlaisia. Optimoija jäsentää SQL-kyselyn ja suorittaa erityyppisiä optimointivaiheita, joiden jälkeen kyselylle muodostetaan saantipolkuja. Hofferin [7, s. 322] mukaan nykyiset tietokannanhallintajärjestel-

mien optimoijat toimivat **kustannusperustaisina** (engl. *cost based optimizer*), jolloin optimoija valitsee useista vaihtoehtoisista suoritussuunnitelmista kustannuksiltaan edullisimman SQL-kyselyn suorittamistavaksi. Kyselyn analysoinnin ja optimoinnin aikana optimoija saattaa Hofferin [7, s. 336] mukaan muokata SQL-kyselyä (engl. *query rewrite*) nopeammaksi vaikkapa vaihtamalla kyselyssä käytettyjä predikaatteja yhtäpitäviksi ehtorakenteiksi, kuten `BETWEEN`-määreitä (ks. luku 5.4) vertailuehdoiksi tai toisinpäin.

Hofferin [7, s. 338] mukaan joissakin tietokannanhallintajärjestelmissä voidaan hyödyntää myös **sääntöperustaista optimoijaa** (engl. *rule based optimizer*). Tällöin optimointipäätökset perustuvat SQL-syntaksiin ja SQL:n rakenteeseen, predikaattien ja taulujen sijoitteluun kyselyssä sekä indeksien olemassaoloon. Hyödynnettäessä sääntöperustaista optimoijaa tulee sovelluksen kehittäjän olla selvillä kyseessä olevista säännöistä, koska muussa tapauksessa kyselyn suorituskyky saattaa heikentyä vaikkapa kyselyssä esiteltyjen taulujen tai kenttien väärän järjestyksen takia.

Elmasrin [6, s. 604 ja 607] mukaan **heuristisen sääntöjen** (engl. *heuristic rules*) avulla optimoidaan tietokannanhallintajärjestelmän optimoijan tuottamaa sisäistä kyselyn suorittamistapaa vaihtamalla hakurakenteita yhtäpitäviksi tehokkaammiksi rakenteiksi. Elmasrin [6, s. 614] mukaan tärkein heuristinen sääntö on SQL-kyselyn `SELECT`- ja `WHERE`-lausekkeiden ehtoja mahdollisimman aikaisessa vaiheessa hyödyntäen pienentää välitulosten rivi- ja sarakemääriä. Lisäksi taulujen liitokset tulisi järjestää siten, että tuotettavaa rivimäärää eniten rajoittavat liitokset muodostetaan ensin.

10 Yhteenveto

Tutkielmassa tarkasteltiin tietokantaa hyödyntävän tietojärjestelmän nopeuttamista tietokantakyselyiden muokkauksen, tietokantaan luotavien indeksien ja sovelluksen lähdekoodin muokkauksen avulla. Nopeuttamiskeinojen toimivuutta analysoitiin Jyväskylän yliopistossa kehitetyssä ja käytössä olevassa Korppi-opintotietojärjestelmässä, mutta suurin osa havainnoista ja suosituksista yleistyy käytettäväksi mihin tahansa tietokannanhallintajärjestelmään ja tietokantaan. Kaikkiin tietokantojen taulurakenteisiin, datan määrään ja tietokannanhallintajärjestelmien ja niiden versioihin soveltuvia nyrkkisääntöjä ei kuitenkaan ollut löydettävissä, vaan tietokantakyselyiden nopeuttaminen vaatii kokemuksen ohella runsaasti erilaisten toteutusratkaisujen kokeiluja ja vertailuja.

Korppi-järjestelmän tietokantaan tallennetun datan määrä on käyttäjien ja järjestelmän laajentuneen hyödyntämisen myötä lisääntynyt valtavasti vuosien varrella, joten järjestelmän kehitystyössä kiireisellä aikataululla toteutettuja ratkaisuja on ajan saatossa jouduttu optimoimaan jouhevan käytön turvaamiseksi. Korppi-järjestelmän palvelinsovellusten kirjoittamien lokitiedostojen sekä järjestelmän tietokannan käyttötapoja kuvaavien tilastojen avulla voidaan selvittää mahdolliset pullonkaulat (ks. luku 6.5). Kun lokitiedostojen ja käyttötilastojen avulla on selvitetty tehottomia ja aikaavieviä SQL-kyselyitä, voidaan näitä analysoida tarkemmin luvussa 6.3 kuvatuin keinoin. Tarkemman analysoinnin tuloksena saatetaan päätyä vaikkapa siihen, että tietokannan tauluihin tulisi luoda tietojen hakua nopeuttavia indeksejä (ks. luvut 5.6 ja 5.7). Jos pelkkä indeksien luonti ei riitä nopeuttamaan hakua riittävästi, toisinaan joudutaan muokkaamaan SQL-kyselyä nopeammaksi (ks. luku 7.1). Tämä onnistuu vaikkapa siirtämällä alikysely osaksi pääkyselyä taikka jakamalla iso kysely yksinkertaisempiin ja selkeämpiin osiin alikyselyiden avulla.

Jos SQL-kyselyn optimointi ei nopeuta toiminnon suorittamista tarpeeksi, on mahdollista muokata sovelluksen lähdekoodin algoritmeja ja tietorakenteita tehokkaammiksi. Tällöin vaikkapa iso ja monimutkainen SQL-kysely voidaan yleensä jakaa suoritettavaksi erillisinä pieninä palasina, joiden hakutulokset kootaan varsinaiseksi lopputulokseksi sovelluksen lähdekoodin keinoin. Myös tehokkaampia tietorakenteita ja tietokantakyselyiden välimuistiratkaisuja (ks. luvut 7.3 ja 7.4) voidaan

hyödyntää lähdekoodia muokkaamalla.

Tutkielmaan on dokumentoitu Korppi-järjestelmän kehittämisen aikana olennaisia toimiviksi ja toimimattomiksi havaittuja menetelmiä. Tietokantoihin liittyvän teorian, luvun 8 nopeuttamisesimerkkien sekä luvun 8.7 havaintojen ja suositusten toivotaan auttavan muun muassa Korppi-järjestelmän jatkokehittäjiä tehostamaan tietokannan hyödyntämistä.

Lähteet

[1] Ashenfelter John Paul, *What's the Big Deal about SQL?*, saatavilla HTML-muodossa <URL: http://www.linuxdevcenter.com/pub/a/linux/2000/10/20/aboutSQL_1.html>, O'Reilly Media, 20.10.2000.

[2] Celko Joe, *Joe Celko's SQL For Smarties: Advanced SQL Programming*, Morgan Kaufmann Publishers, 1995.

[3] Codd Edgar F., *A Relational Model of Data for Large Shared Data Banks*, Commun. ACM, Volume 13, Number 6, 1970.

[4] Codd Edgar F., *Further Normalization of the Data Base Relational Model*, IBM Research Report, RJ909, San Jose, California, 1971.

[5] Eidemiller Sean, *Using CachedRowSet to Transfer JDBC Query Results Between Classes*, saatavilla HTML-muodossa <URL: http://www.onjava.com/pub/a/onjava/2004/06/23/cachedrowset.html>, O'Reilly Media, 23.6.2004.

[6] Elmasri Ramez and Navathe Shamkant B., *Fundamentals of Database Systems, 3rd Edition*, Addison-Wesley, 2000.

[7] Hoffer Jeffrey A., Prescott Mary B. and McFadden Fred R., *Modern Database Management, Sixth Edition*, Pearson Education, 2002.

[8] Korppi-järjestelmän kehitysryhmä, *Korppi-järjestelmän kehitys ja kehityksessä mukana olleet henkilöt*, saatavilla HTML-muodossa <URL: https://korppi.jyu.fi/kotka/help/faq/history.jsp>, Jyväskylän yliopisto, 8.9.2004.

[9] Korppi-järjestelmän kehitysryhmä, *KOTKA-järjestelmän tietokantaraportti*, saatavilla HTML-muodossa <URL: http://sovellusprojektit.it.jyu.fi/kottarainen/dokumentit/viralliset/html/KOTKA-tietokantaraportti/KOTKA-tietokantaraportti.html>, Jyväskylän yliopisto, 30.8.2003.

[10] Korppi-järjestelmän kehitysryhmä, *Kurki- ja Korppi-tekniikkaa*, saatavilla HTML-muodossa <URL: https://korppi.jyu.fi/kotka/help/faq/technique.jsp>, Jyväskylän yliopisto, viitattu 10.12.2006.

[11] Mullins Craig S., *Database Administration. The Complete Guide to Practices and Procedures*, Pearson Education, 2002.

[12] PostgreSQL Global Development Group, *PostgreSQL JDBC Driver*, saatavilla HTML-muodossa <URL: http://jdbc.postgresql.org/>, viitattu 25.11.2007.

[13] PostgreSQL Global Development Group, *PostgreSQL 8.3.1 Documentation*, saatavilla HTML-muodossa <URL: http://www.postgresql.org/docs/current/interactive/>, viitattu 1.6.2008.

[14] PostgreSQL Global Development Group, *PostgreSQL Mailing Lists*, saatavilla HTML-muodossa <URL: http://archives.postgresql.org/>, viitattu 1.6.2008.

[15] PostgreSQL Global Development Group, *PostgreSQL 7.4.19 Documentation. 23.2. The Statistics Collector*, saatavilla HTML-muodossa <URL: http://www.postgresql.org/docs/7.4/interactive/monitoring-stats.html>, viitattu 23.4.2008.

[16] PostgreSQL Global Development Group, *PostgreSQL 7.4.18 Documentation. 11.3. Multicolumn Indexes*, saatavilla HTML-muodossa <URL: http://www.postgresql.org/docs/7.4/interactive/indexes-multicolumn.html>, viitattu 28.10.2007.

[17] PostgreSQL Global Development Group, *PostgreSQL 8.2.5 Documentation. 11.3. Multicolumn Indexes*, saatavilla HTML-muodossa <URL: http://www.postgresql.org/docs/current/interactive/indexes-multicolumn.html>, viitattu 28.10.2007.

[18] PostgreSQL Global Development Group, *PostgreSQL 8.3.3 Documentation. 11.10. Examining Index Usage*, saatavilla HTML-muodossa <URL: http://www.postgresql.org/docs/8.3/interactive/indexes-examine.html>, viitattu 16.6.2008.

[19] PostgreSQL Global Development Group, *PostgreSQL 8.2.5 Documentation. 11.2. Index Types*, saatavilla HTML-muodossa <URL: http://www.postgresql.org/docs/current/interactive/indexes-types.html>, viitattu 28.10.2007.

[20] PostgreSQL Global Development Group, *PostgreSQL 7.4.19 Documentation. CREATE TABLE*, saatavilla HTML-muodossa <URL: http://www.postgresql.org/docs/7.4/interactive/sql-createtable.html>, viitattu 23.4.2008.

[21] PostgreSQL Global Development Group, *PostgreSQL 8.3.1 Documentation. CREATE TYPE*, saatavilla HTML-muodossa <URL: http://www.postgresql.org/docs/8.3/interactive/sql-createtype.html>, viitattu 11.5.2008.

[22] PostgreSQL Global Development Group, *PostgreSQL 7.4.18 Documentation. EXPLAIN*, saatavilla HTML-muodossa <URL: http://www.postgresql.org/docs/7.4/interactive/sql-explain.html>, viitattu 7.10.2007.

[23] PostgreSQL Global Development Group, *PostgreSQL 7.4.19 Documentation. Chapter 43. System Catalogs. 43.9. pg_class*, saatavilla HTML-muodossa <URL: http://www.postgresql.org/docs/7.4/interactive/catalog-pg-class.html>, viitattu 25.4.2008.

[24] PostgreSQL Global Development Group, *PostgreSQL 7.4.18 Documentation. Chapter 24. Monitoring Disk Usage*, saatavilla HTML-muodossa <URL: http://www.postgresql.org/docs/7.4/static/diskusage.html>, viitattu 29.9.2007.

[25] PostgreSQL Global Development Group, *PostgreSQL 7.4.18 Documentation. 16.4. Run-time Configuration*, saatavilla HTML-muodossa <URL: http://www.postgresql.org/docs/7.4/interactive/runtime-config.html>, viitattu 29.9.2007.

[26] PostgreSQL Global Development Group, *PostgreSQL 8.3.1 Documentation. 18.6. Query Planning*, saatavilla HTML-muodossa <URL: http://www.postgresql.org/docs/8.3/interactive/runtime-config-query.html>, viitattu 11.5.2008.

[27] PostgreSQL Global Development Group, *PostgreSQL 7.4.18 Documentation. Chapter 8. Data Types*, saatavilla HTML-muodossa <URL: http://www.postgresql.org/docs/7.4/static/datatype.html>, viitattu 23.9.2007.

[28] PostgreSQL Global Development Group, *PostgreSQL: History*, saatavilla HTML-muodossa <URL: http://www.postgresql.org/about/history>, viitattu 2.12.2006.

[29] PostgreSQL Global Development Group, *PostgreSQL 8.3.1 Documentation. E.89. Release 7.2*, saatavilla HTML-muodossa <URL: http://www.postgresql.org/docs/current/static/release-7-2.html>, viitattu 1.5.2008.

[30] PostgreSQL Global Development Group, *PostgreSQL 8.3.1 Documentation. E.58. Release 7.4*, saatavilla HTML-muodossa <URL: http://www.postgresql.org/docs/current/static/release-7-4.html>, viitattu 23.4.2008.

[31] SourceForge.net, *PoolMan*, saatavilla HTML-muodossa <URL: http://sourceforge.net/projects/poolman/>, viitattu 25.11.2007.

[32] Laine Harri (toim.), *Relaatiotietokantasanasto (1997-12)*, toim. Harri Laine, saatavilla HTML-muodossa <URL: http://www.cs.helsinki.fi/u/laine/relaatiosanasto/>, SYSTA-tiedonhallintatyöryhmä, Tietotekniikan kehittämiskeskus ry, viitattu 2.12.2006.

[33] Sun Microsystems Inc., *JDBC Overview*, saatavilla HTML-muodossa <URL: http://java.sun.com/products/jdbc/overview.html>, viitattu 27.4.2008.

[34] Sun Microsystems Inc., *JDBC 2.0 Standard Extension API*, saatavilla PDF-muodossa <URL: http://java.sun.com/products/jdbc/jdbc20.stdext.pdf>, 7.12.1998.

[35] Özsu M. Tamer and Valduriez Patrick, *Principles of Distributed Database Systems, 2nd Edition*, Prentice Hall, 1999.

Liitteet

A Korppi-järjestelmän tietokannan ja lähdekoodin kehitystilastoja

Korppi-järjestelmän tietokanta varmuuskopioidaan määräajoin otettujen tietokantavedosten (engl. *database dump*) avulla, joten levylle tallennettuja tietokannan käytössä olevia tiedostoja ei kopioida. Tietokantavedokseen PostgreSQL-tietokannanhallintajärjestelmä sisällyttää kaikkien tietokannassa olevien taulujen, indeksien ja muiden tietorakenteiden määrittelemiseksi tarvittavat SQL-lauseet sekä kaikkien taulujen datan SQL-lauseina.

Taulukossa A.1 esitetään tietokantavedosten koot megatavuina sekä taulujen ja indeksien lukumäärät. Indeksien lukumääriin ei lasketa taulujen perusavaimiin automaattisesti luotuja indeksejä. Taulukossa A.2 esitetään muutamien oleellisimpien Korppi-järjestelmän tietokannan taulujen rivien lukumäärät. Taulukoiden A.1 ja A.2 tiedot on selvitetty kunakin vuonna tammi-helmikuussa otetusta tietokantavedoksesta.

Vuosi	Koko (MB)	Tauluja (kpl)	Indeksejä (kpl)
2002	3 MB	113	0
2004	75 MB	232	58
2006	620 MB	271	116
2008	2200 MB	363	160

Taulukko A.1: Korppi-järjestelmän tietokantavedosten koot sekä taulujen ja indeksien lukumäärät.

Korppi-järjestelmän lähdekoodi koostuu JSP-sivuista ja Java-luokista. Taulukossa A.3 esitetään Korppi-järjestelmän JSP-sivujen ja Java-luokkien lukumäärät sekä rivimäärät vuosien varrelta. Versiohallinta otettiin Korppi-järjestelmän kehityksessä käyttöön lokakuussa 2002.

Vuosi	course	courseinstance	event	groupparticipant	organisation	person
2002	30	26	1400	4500	44	1900
2004	4900	6000	105000	175000	180	13000
2006	33000	28000	436000	651000	1200	37000
2008	62000	53000	799000	1454000	1650	52000

Taulukko A.2: Korppi-järjestelmän oleellisimpien taulujen rivien lukumäärät.

Kuukausi ja vuosi	**JSP-sivuja (kpl)**	**JSP-sivuilla lähdekoodia (riviä)**	**Java-luokkia (kpl)**	**Java-luokissa lähdekoodia (riviä)**
tammikuu 2002	272	29000	48	5600
lokakuu 2002	420	65000	135	52000
huhtikuu 2004	420	92000	340	125000
huhtikuu 2006	800	151000	670	243000
huhtikuu 2008	880	155000	1080	317000

Taulukko A.3: Korppi-järjestelmän JSP-sivujen ja Java-luokkien lukumäärät ja koodirivien lukumäärät.

B PostgreSQL:n ja SQL-92 -standardin tietotyypit

Taulukoissa B.1–B.4 esitellään PostgreSQL-tietokannanhallintajärjestelmän tukemat oleellisimmat tietotyypit [27]. Mikäli tietotyyppi on mukana myös SQL-92 -standardissa, on asiasta maininta taulukoiden SQL-92 -sarakkeessa [2, s. 47–94]. Taulukoissa hakasulkeilla `[ ]` esitetään valinnaiset osat, joita ei tietotyyppiä käytettäessä tarvitse kirjoittaa näkyviin.

Mainittujen tyyppien lisäksi PostgreSQL:ssa on tarjolla myös muutamia tyyppejä geometristen kuvioiden sekä verkko-osoitteiden käsittelyyn. Kaikki PostgreSQL:n tukemat tietotyypit löytyvät dokumentaatiosta [27].

SQL-99 -standardi mahdollistaa omien tietotyyppien luomisen (*User-Defined Datatype*, UDT) `CREATE TYPE` -komennolla. Myös PostgreSQL tukee omien tietotyyppien luomista kyseisellä komennolla, mutta komennon yksityiskohdat eivät ole yhteensopivia SQL-99 -standardin kanssa [21].

Nimi	**Kuvaus**	**SQL-92:ssa**
`bigint` tai `int8`	Etumerkillinen 64-bittinen kokonaisluku.	-
`double precision`	Kaksoistarkkuuden liukuluku.	on
`integer` tai `int` tai `int4`	Etumerkillinen 32-bittinen kokonaisluku.	`integer`
`numeric [(`*p*`,`*s*`)]` tai `decimal [(`*p*`,`*s*`)]`	Tarkka kokonaisluku halutulla tarkkuudella, jossa *p* (*precision*) on merkitsevien numeroiden lkm koko luvun esityksessä ja *s* (*scale*) on desimaalien lkm.	molemmat
`real` tai `float4`	Yksinkertaisen tarkkuuden liukuluku.	`real`
`smallint`	Etumerkillinen 16-bittinen kokonaisluku.	on
`serial`	Automaattisesti numeroituva 32-bittinen kokonaisluku.	-

Taulukko B.1: PostgreSQL:n ja SQL-92 -standardin numeeriset tietotyypit.

Nimi	Kuvaus	SQL-92:ssa
`character varying(`*n*`)` tai `varchar(`*n*`)`	Enintään *n* merkkiä pitkä merkkijono.	molemmat
`character(`*n*`)` tai `char(`*n*`)`	*n* merkin kiinteämittainen merkkijono.	molemmat
`text`	Vaihtuvamittainen merkkijono.	-

Taulukko B.2: PostgreSQL:n ja SQL-92 -standardin merkkien esittämiseen tarkoitetut tietotyypit.

Nimi	Kuvaus	SQL-92:ssa
`date`	Päivämäärä (vuosi, kuukausi, päivä).	on
`interval(`*p*`)`	Suhteellinen aikajakso.	on
`time [without time zone]`	Kellonaika ilman aikavyöhyketietoa.	on
`time with time zone` tai `timetz`	Kellonaika aikavyöhyke huomioiden.	-
`timestamp [without time zone]` tai `timestamp`	Päivämäärä ja kellonaika ilman aikavyöhykettä.	on
`timestamp with time zone` tai `timestamptz`	Päivämäärä ja kellonaika aikavyöhyke huomioiden.	-

Taulukko B.3: PostgreSQL:n ja SQL-92 -standardin aikojen käsittelyyn liittyvät tietotyypit.

Nimi	Kuvaus	SQL-92:ssa
`bit`	Kiinteämittainen bittimerkkijono.	on
`bit varying(`*n*`)`	Enintään *n*-bittinen bittimerkkijono.	on
`boolean`	Boolen totuusarvo eli tosi tai epätosi.	on
`bytea`	Binääridata.	-
`money`	Valuutta.	-

Taulukko B.4: PostgreSQL:n ja SQL-92 -standardin muut tietotyypit.

C Esimerkki taulujen luonnista, muokkauksesta ja poistamisesta

Liitteessä esitetään luvuissa 4.7 ja 4.9 normalisoitu esimerkkitaulurakenne, joka esitetään taulukoissa 4.3, 4.4 ja 4.5. Esimerkkitaulurakenteen taulut luodaan seuraavilla SQL-lauseilla:

```
CREATE TABLE valtio (
 valtiokoodi CHAR(3) NOT NULL,
 valtio VARCHAR(100) NOT NULL,
 CONSTRAINT valtio_pk PRIMARY KEY (valtiokoodi)
);

CREATE TABLE henkilö (
 henkilöID INTEGER NOT NULL,
 sukunimi VARCHAR(100) NOT NULL,
 etunimi VARCHAR(100) NOT NULL,
 valtiokoodi CHAR(3) NULL,
 CONSTRAINT henkilö_pk PRIMARY KEY (henkilöID),
 CONSTRAINT henkilön_valtio FOREIGN KEY (valtiokoodi)
  REFERENCES valtio(valtiokoodi) ON UPDATE CASCADE ON DELETE RESTRICT
);

CREATE TABLE henkilön_omaisuus (
 henkilöID INTEGER NOT NULL,
 hyödyke VARCHAR(100) NOT NULL,
 lukumäärä INTEGER NOT NULL,
 CONSTRAINT henkilön_omaisuus_pk PRIMARY KEY (henkilöID, hyödyke),
 CONSTRAINT henkilön_omaisuus_henkilöID FOREIGN KEY (henkilöID)
  REFERENCES henkilö(henkilöID) ON UPDATE CASCADE ON DELETE RESTRICT
);
```

Tauluihin luodaan hakuja tehostavia indeksejä seuraavasti:

```
CREATE INDEX henkilö_sukunimi ON henkilö(sukunimi);
CREATE INDEX henkilö_valtiokoodi ON henkilö(valtiokoodi);
CREATE INDEX henkilön_omaisuus_hyödyke ON henkilön_omaisuus(hyödyke);
```

`valtio`-taulussa ei samaa valtiota kohti saa olla useampia rivejä. Valtiokoodi sen sijaan toimii kyseisessä taulussa perusavaimena, joten samaa valtiokoodia ei voida missään tapauksessa tallentaa useammalle riville. Lisäksi voidaan olettaa, että kullakin henkilöllä tulee kutakin hyödykelajia kohti olla `henkilön_hyödykkeet`-tauluun tallennettuna korkeintaan yksi rivi. Näiden vaatimusten toteuttamiseksi tauluihin luodaan yksilöivät indeksit seuraavasti:

```
CREATE UNIQUE INDEX valtio_nimet ON valtio(valtio);
CREATE UNIQUE INDEX henkilön_hyödykkeet
 ON henkilön_omaisuus(henkilöID, hyödyke);
```

Taulujen sisältö voidaan määrittää tyhjiin tauluihin seuraavilla SQL-lauseilla:

```
INSERT INTO valtio (valtiokoodi, valtio) VALUES ('fi', 'Suomi');
INSERT INTO valtio (valtiokoodi, valtio) VALUES ('sv', 'Ruotsi');
INSERT INTO valtio (valtiokoodi, valtio) VALUES ('ru', 'Venäjä');
INSERT INTO valtio (valtiokoodi, valtio) VALUES ('us', 'Yhdysvallat');

INSERT INTO henkilö (henkilöID, sukunimi, etunimi, valtiokoodi)
 VALUES (1, 'Meikäläinen', 'Matti', 'fi');
INSERT INTO henkilö (henkilöID, sukunimi, etunimi, valtiokoodi)
 VALUES (3, 'Kolmonen', 'Jaakko', 'fi');
INSERT INTO henkilö (henkilöID, sukunimi, etunimi, valtiokoodi)
 VALUES (5, 'Palme', 'Olof', 'sv');
INSERT INTO henkilö (henkilöID, sukunimi, etunimi, valtiokoodi)
 VALUES (6, 'Doe', 'John', NULL);
INSERT INTO henkilö (henkilöID, sukunimi, etunimi, valtiokoodi)
 VALUES (7, 'Presley', 'Elvis', 'us');

INSERT INTO henkilön_omaisuus (henkilöID, hyödyke, lukumäärä)
 VALUES (1, 'tietokone', 1);
INSERT INTO henkilön_omaisuus (henkilöID, hyödyke, lukumäärä)
  VALUES (1, 'asunto', 1);
INSERT INTO henkilön_omaisuus (henkilöID, hyödyke, lukumäärä)
  VALUES (3, 'wokkipannu', 2);
INSERT INTO henkilön_omaisuus (henkilöID, hyödyke, lukumäärä)
  VALUES (7, 'kitara', 4);
INSERT INTO henkilön_omaisuus (henkilöID, hyödyke, lukumäärä)
  VALUES (7, 'Cadillac', 1);
INSERT INTO henkilön_omaisuus (henkilöID, hyödyke, lukumäärä)
  VALUES (7, 'asunto Gracelandissa', 1);
```

Esimerkkitaulurakenteen taulut sisältöineen saadaan poistettua seuraavilla SQL-poistolauseilla:

```
DROP TABLE henkilön_omaisuus;
DROP TABLE henkilö;
DROP TABLE valtio;
```

Jotta taulujen ja niiden kenttien rakenteisiin voidaan tehdä muutoksia, tulee olemassa olevat taulut pääosin ensin poistaa taulurakenteesta. Toisinaan rakenteita on mahdollista muokata sekä vaikkapa lisätä ja poistaa yksittäisiä kenttiä ilman, että

koko taulu joudutaan poistamaan. Taulun poistamisen yhteydessä tulee huolehtia myös siitä, että taulun sisältö palautetaan tarvittaessa uuden korvaavan taulun sisällöksi. Taulut tulee poistaa luontijärjestykseen nähden päinvastaisessa järjestyksessä, koska viite-eheyksien takia perustauluja ei voida poistaa ennen niihin viittaavia tauluja.

D SQL-lauseiden analysointiesimerkkejä

Luvussa 6.3 esitellään kyselyiden analysointityökaluja. Liitteessä kuvataan tarkemmin kyselyn suorittamistavan analysointia PostgreSQL-tietokannanhallintajärjestelmän avulla.

Kyselyn suorittamistavan analysointi

Tietokannan taulujen, indeksien ja muiden objektien lukumäärät hakevan kyselyn suorittamistapa analysoidaan antamalla PostgreSQL-tietokannanhallintajärjestelmälle SQL-lause `EXPLAIN ANALYZE` -muodossa seuraavasti:

```
EXPLAIN ANALYZE
SELECT relkind, count(*) AS count
FROM pg_class
GROUP BY relkind
ORDER BY count DESC;
```

PostgreSQL palauttaa suorittamistavan kuvauksen seuraavana tulosteena:

```
Sort (cost=72.74..72.75 rows=6 width=1)                                    1
     (actual time=2.781..2.783 rows=6 loops=1)                             2
  Sort Key: count(*)                                                       3
  -> HashAggregate (cost=72.64..72.66 rows=6 width=1)                      4
                (actual time=2.719..2.724 rows=6 loops=1)                  5
      -> Seq Scan on pg_class (cost=0.00..62.43 rows=2043 width=1)         6
                (actual time=0.006..1.193 rows=2049 loops=1)               7
Total runtime: 2.833 ms                                                    8
(5 rows)                                                                   9
```

Kyselyn suorittamisen aikana läpikäydyt eri operaatiot erotellaan tulosteessa yleensä nuolimerkinnällä `->` ja sisentämällä operaatio siihen liittyvän ylempänä olevan laajemman operaation alle. Tulostetta luetaan eniten sisennetystä osasta alkaen, joka on suoritettu ensimmäisenä.

Em. esimerkissä on ensimmäisenä rivillä 6 luettu `pg_class`-näkymän tietoja peräkkäishakuna (`Seq Scan`). `cost`-kohdissa esitetyt arvoalueet kuvaavat sitä, kuinka raskaita kyseisten operaatioiden suorittaminen oli kyselyoptimoijan arvion mukaan. Kyseiset arvoalueet esitetään levylohkojen lukumäärinä. Rivin 6 `cost`-arvoalueen alkuarvo (`0.00`) on levylohkojen määrä, joka tarvitaan kyseisen operaation läpikäynnin aloittamiseen. Arvoalueen loppuarvo (`62.43`) taasen arvioi niiden

levylohkojen määrän, joka tarvitaan operaation läpikäynnin päättymiseen. Kyseisen sulkuparin sisällä esitetään myös arvot `rows` ja `width`. Nämä kuvaavat arviota siitä, montako riviä operaatiosta palautetaan ja montako tavua kuhunkin palautettuun riviin kuuluu keskimäärin. Tässä tapauksessa siis arvioitiin, että levymuistista tarvitaan noin 62 levylohkon tiedot, jotta kyseinen taulun selausoperaatio voidaan suorittaa loppuun asti.

Koska kysely suoritettiin `EXPLAIN ANALYZE` -muodossa, esitetään operaatioille myös tarkat suoritustiedot. `time`-kohtien arvoalueiden avulla kerrotaan operaation suoritusajat millisekunteina. Tässä tapauksessa rivin 7 merkintä `(actual time=0.006..1.193 rows=2049 loops=1)` kertoo, että operaation suorittamisen aloittamiseen kului aikaa kuusi millisekunnin tuhannesosaa ja operaatio saatiin valmiiksi noin 1.2 millisekunnissa. Operaation tuloksena luettiin `pg_class`-näkymästä kertaalleen (`loops=1`) runsaat 2000 riviä (`rows=2049`).

Seuraavassa vaiheessa rivillä 4 suoritetaan `HashAggregate`-ryhmittelyoperaatio, joka käsittelee edellisessä vaiheessa saatua hakutulosta tietokannanhallintajärjestelmän sisäisesti. Tämän vaiheen tuloksena on edellisen vaiheen rivin 6 hakutuloksen runsaat 2000 riviä ryhmitelty rivillä 4 kuuteen välitulosriviin (yksi kutakin tietokannan objektin tyyppiä kohden) viidessä millisekunnin tuhannesosassa. Seinäkelloaikaa kyselyn suorittamisen aloittamisesta tämän vaiheen loppuunsaattamiseen on kulunut 2.724 ms.

Viimeisessä vaiheessa riveillä 1–3 hakutulos järjestetään rivin 3 `Sort Key` -merkityn avaimen eli ryhmien sisältämien alkuperäisten rivien lukumäärän mukaan (laskevassa järjestyksessä, mutta tätä ei tuloste mainitse). Kun oikea järjestys on mainituille kuudelle riville saatu aikaan, on kulunut seinäkelloaikaa kyselyn suorittamisen aloittamisesta yhteensä 2.783 ms. Lopputuloksena palautetaan kyseiset kuusi riviä järjestettynä halutulla tavalla.

Tulosteen toiseksi viimeisellä rivillä 8 esitetään kyselyn analysoinnin ja suorittamisen kokonaisaika (2.833 ms), joka on yleensä hieman suurempi kuin viimeisessä vaiheessa saatu kokonaissuoritusaika (rivillä 1 esitetty 2.783 ms). Tulostuksen muotoiluun ja muuhun vastaavaan yleisrasitteeseen (engl. *overhead*) kului siis jonkin verran aikaa. Viimeisellä rivillä 9 tulosteessa esitetään `EXPLAIN`-tulosteen rivien lukumäärä, jota ei tule sekoittaa kyselyn lopputuloksena saatujen rivien lukumäärään.

Tulosteen oleellisimpia merkille pantavia tietoja ovat yleensä `actual`-osioiden `time`-arvoalueiden loppuarvot. Kyseiset arvot kuvaavat kyselyn suorittamiseen vaadittua kokonaisaikaa siihen mennessä, kun kyseinen yksittäinen ope-

raatio on kokonaan suoritettu. Toisinaan myös `loops`-merkintöjen lukumääriä tulee seurata varsinkin, jos lukumääräarvo on suuri luku, vaikkapa satoja tuhansia tai miljoonia. Tällaisessa tapauksessa taulua tai indeksiä on läpikäyty arvon mukaisesti tuhansia tai miljoonia kertoja. Tällöin kyselyä tai tietokannan rakennetta tulisi mahdollisesti tehostaa jollain tavalla, vaikkapa indeksejä lisäämällä tai indeksien rakennetta muokkaamalla.

SQL-kyselylauseen analysointi

Luvun 5.8 kaikki henkilöt valtiotietoineen hakeva SQL-kyselylause saadaan analysoitua tarkkoine suoritusaikoineen ja operaatioineen antamalla PostgreSQL-tietokannanhallintajärjestelmälle seuraava komento:

```
EXPLAIN ANALYZE
SELECT h.henkilöID, h.sukunimi, h.etunimi, v.valtiokoodi, v.valtio
FROM henkilö AS h LEFT OUTER JOIN valtio AS v
ON (h.valtiokoodi = v.valtiokoodi)
ORDER BY LOWER(h.sukunimi), LOWER(h.etunimi);
```

PostgreSQL palauttaa kyselyn suorittamistavaksi seuraavanlaisen tulosteen:

```
Sort (cost=2.25..2.26 rows=5 width=40)                                    1
    (actual time=0.101..0.101 rows=5 loops=1)                             2
 Sort Key: lower((h.sukunimi)::text), lower((h.etunimi)::text)            3
 -> Hash Left Join (cost=1.05..2.19 rows=5 width=40)                      4
                (actual time=0.065..0.085 rows=5 loops=1)                 5
     Hash Cond: ("outer".valtiokoodi = "inner".valtiokoodi)               6
     -> Seq Scan on "henkilö" h (cost=0.00..1.05 rows=5 width=29)         7
          (actual time=0.002..0.005 rows=5 loops=1)                       8
     -> Hash (cost=1.04..1.04 rows=4 width=18)                            9
          (actual time=0.019..0.019 rows=0 loops=1)                       10
          -> Seq Scan on valtio v (cost=0.00..1.04 rows=4 width=18)       11
               (actual time=0.008..0.013 rows=4 loops=1)                  12
Total runtime: 0.159 ms                                                   13
```

Tulostetta tulee tarkastella eniten sisennetystä operaatiosta alkaen riveiltä 11–12, jossa siis luetaan `valtio`-taulu kertaalleen läpi peräkkäishaulla. Rivin 11 arvion mukaan kyseisestä operaatiosta palautetaan neljä riviä, joista kukin on kooltaan keskimäärin 18 tavua. Kyseisen haun rinnalla luetaan myös `henkilö`-taulu läpi riveillä 7–8. Näiden kahden lukuoperaation tulos liitetään `valtiokoodi`-kentän arvojen avulla toisiinsa rivillä 6. Kyseisen välituloksen arvioidaan rivillä 4 olevan kooltaan viisi riviä, keskimäärin 40 tavua per rivi. Ulkoliitoksen viisi riviä si-

sältävä lopputulos (`Hash Left Join`) on valmis rivillä 4. Kokonaissuoritusaikaa on tämän tuloksen aikaansaamiseksi kulunut rivillä 5 esitetty 0.085 ms. Lopuksi riveillä 1–3 haun lopputulos järjestetään sukunimen ja etunimen mukaan järjestykseen. Lopputuloksena saadaan viisi riviä eli yksi rivi kutakin henkilöä kohti riippumatta siitä, onko kyseiselle henkilölle asetettu `henkilö`-taulussa valtiokoodia vai ei.

Indeksin hyödyntäminen suorittamistavassa

Kyselyssä halutaan hakea Korppi-järjestelmän tietokannasta henkilöiden tunnisteet (`personid`) ja sähköpostiosoitteiden (`personparametertypeid=3`) lukumäärät kaikilta niiltä henkilöiltä, joilla on hakuhetkellä määriteltynä vähintään kaksi sähköpostiosoitetta. Mukaan otetaan vain ne henkilöt, jotka ovat rekisteröityneet Korppi-järjestelmään sadan ensimmäisen joukossa (`personid < 100`). SQL-kyselylause saadaan analysoitua seuraavasti:

```
EXPLAIN ANALYZE
SELECT pp.personid, COUNT(*) as email_count
FROM personparameter AS PP
WHERE pp.personparametertypeid = 3 AND pp.deleted = false
AND pp.personid < 100
GROUP BY pp.personid
HAVING COUNT(*) > 1;
```

PostgreSQL palauttaa kyselyn suorittamistavaksi seuraavanlaisen tulosteen:

```
GroupAggregate (cost=0.00..7881.05 rows=8 width=4)                          1
            (actual time=1.322..3.768 rows=2 loops=1)                       2
  Filter: (count(*) > 1)                                                    3
  -> Index Scan using personid_personparameter_key                          4
     on personparameter pp (cost=0.00..7879.94 rows=140 width=4)            5
     (actual time=0.027..3.599 rows=103 loops=1)                            6
       Index Cond: (personid < 100)                                         7
       Filter: ((personparametertypeid = 3) AND (deleted = false))          8
Total runtime: 3.825 ms                                                     9
```

Riveillä 4–8 on hyödynnetty `personparameter`-taulun alusta loppuun lukemisen sijasta kyseisen taulun `personid`-kenttään luotua `personid_personparameter_key`-nimistä indeksiä. Rivin 7 indeksin hakuehdon ja rivin 8 lisäsuodatusehtojen hyödyntämisen jälkeen operaation lopputuloksena palautetaan rivin 6 mukaisesti 103 riviä seuraavaan operaatioon. Indeksin

lukemiseen ja lisäsuodatusehtojen hyödyntämiseen kului aikaa 3.599 ms. Rivin 5 arvion mukaan operaatiosta palautettaisiin 140 riviä, joka ei siis täsmälleen pidä paikkaansa.

Riveillä 1–3 muodostetaan ryhmittelyä ja ryhmittelyn suodatusehtoa hyödyntäen kyselylauseen lopputuloksena kaksi riviä, jolloin kokonaisuutena aikaa on kulunut noin 3.8 ms. Korppi-järjestelmän sadasta ensimmäisenä rekisteröityneestä henkilöstä siis kahdella henkilöllä on hakuhetkellä määriteltynä vähintään kaksi sähköpostiosoitetta.

`personparameter`-taulussa on `personid`-kentän lisäksi indeksi myös `personparametertypeid`-kentässä. Em. kyselyä suorittaessaan tietokannanhallintajärjestelmän kyselyoptimoija päätteli kyseisestä taulusta kerättyjen tilastojen perusteella, että jos indeksejä hyödynnetään, kannattaa hyödyntää nimenomaan `personid`-kenttään luotua indeksiä. Tämä johtuu siitä, että sadalla ensimmäisellä Korppi-järjestelmään rekisteröityneellä käyttäjällä (`personid < 100`) on `personparameter`-tauluun tallennettuna yhteensä noin 2000 riviä, kun taas kyseiseen tauluun on kaikille järjestelmän käyttäjille määritelty yhteensä lähes 52000 sähköpostiosoitetta. Yhteensä kyseisessä `personparameter`-taulussa on runsaat 844000 riviä. Tällöin kahdesta mainitusta indeksivaihtoehdosta nimenomaan `personid`-kenttään luotu indeksi rajaa em. kyselyn välitulosta pienemmäksi ennen lisäsuodatusehdon käsittelyä rivillä 8.

Edellä mainitusta syystä johtuen Korppi-järjestelmän taulujen `deleted`-kenttiin ei kannata luoda yksikenttäisiä indeksejä, koska yleensä Korppi-järjestelmän tauluissa suurin osa riveistä on voimassa olevia (`deleted`-kentän arvona on `false`). Mainittuun kenttään luodusta indeksistä saattaisi olla hyötyä ainoastaan silloin, kun haetaan nimenomaan tauluissa poistetuksi merkattuja rivejä (`deleted`-kentän arvona on `true`), eikä mitään muita taulujen indeksejä voida hyödyntää.

PostgreSQL:n asetusten muokkaaminen kyselylauseen analysoinnissa

PostgreSQL-tietokannanhallintajärjestelmän asetusten avulla voidaan tutkia myös sitä, miten kyselylause suoritetaan, jos vaikkapa indeksien hyödyntäminen kielletään kokonaan. Kyselylause suoritetaan ilman indeksejä seuraavalla tavalla:

```
SET ENABLE_INDEXSCAN TO OFF;                                                    1
                                                                                2
EXPLAIN ANALYZE                                                                 3
SELECT pp.personid, COUNT(*) as email_count                                     4
FROM personparameter AS PP                                                      5
WHERE pp.personparametertypeid = 3 AND pp.deleted = false                       6
AND pp.personid < 100                                                           7
GROUP BY pp.personid                                                            8
HAVING COUNT(*) > 1;                                                            9
                                                                                10
SET ENABLE_INDEXSCAN TO ON;                                                     11
```

Ensimmäisellä rivillä annetaan `SET`-komento indeksien hyödyntämisen kieltämiseksi, kolmannelta riviltä alkaen kyselyn suorittamistapa analysoidaan ja lopuksi rivin 11 `SET`-komennolla taas sallitaan indeksien hyödyntäminen. Pelkän peräkkäishaun avulla mainittu kysely suoritetaan noin 380 millisekunnissa, joten indeksiä hyödyntämällä kyselyn suoritusaika lyhenee sadasosaan pelkkää peräkkäishakua käyttävään kyselyyn nähden.

`SET`-komennon avulla muokattu PostgreSQL:n asetus on voimassa kyseisen käyttäjän tietokantaistunnon ajan, joten asetus ei vaikuta muiden tietokantakäyttäjien toimintaan kyseisessä tietokannassa, eikä asetusta tarvitse analysoinnin ja istunnon päätteeksi välttämättä palauttaa entiselleen.

Vastaavaan tapaan voidaan pakottaa indeksit käyttöön kieltämällä peräkkäishaut antamalla seuraava komento:

```
SET ENABLE_SEQSCAN TO OFF;
```

PostgreSQL:n dokumentaatiossa [26] luetellaan kaikki tietokannanhallintajärjestelmän asetukset, joilla kyselyoptimoijan toimintaa voidaan tarvittaessa säätää.