Tampereen yliopisto

Jukka Ala-Fossi

HIERARKKISIEN NÄKYMIEN GENEROINTI RELAATIOTIETOKANNAN SANAKIRJADATASTA

Informaatioteknologian ja viestinnän tiedekunta
Pro gradu -tutkielma
Marraskuu 2021

TIIVISTELMÄ

Jukka Ala-Fossi: Hierarkkisien näkymien generointi relaatiotietokannan sanakirjadatasta
Pro gradu -tutkielma
Tampereen yliopisto
Tietojenkäsittelytieteiden tutkinto-ohjelma
Marraskuu 2021

Sanakirjadata on monimuotoista ja sisältää paitsi merkityksiä, myös esimerkiksi termejä, variantteja ja luokitteluja. Se on myös rakenteeltaan monimutkaista, koska sen eri osat voivat olla hierarkkisia. Sanakirja- ja tesaurusmuotoinen sanakirjadata esitetään usein hierarkkisessa muodossa, mutta näiden kahden muodon hierarkiat ovat erilaiset. Sanakirjadatan hierarkkisesta muodosta huolimatta sen tallentaminen relaatiotietokantaan on varsin houkutteleva vaihtoehto, koska relaatiotietokannat ovat kaikkein yleisimpiä tietokantoja. Relaatiotietokantaan tallennettaessa sanakirjadatan hierarkia täytyy ottaa huomioon tietokannan rakenteen suunnittelussa, että se pystyy säilyttämään koko hierarkian muodossa, josta se on uudelleen johdettavissa.

Tämä tutkielma käsittelee hierarkkisien näkymien generointia relaatiotietokantaan tallennetusta sanakirjadatasta. Osana tutkielmaa toteutettiin sovellus, joka tekee relaatiotietokantaan rekursiivisia kyselyitä ja tuottaa niiden perusteella XML- ja HTML-dokumentteja. Nämä dokumentit sisältävät hierarkkisen tesaurusmuotoisen version relaatiotietokannan sisällöstä. Sovelluksen käyttämä relaatiotietokanta sisältää sanakirjamuotoista sanakirjadataa, jota ei alun perin ole lainkaan suunniteltu käytettäväksi tesauruksena. Se kuitenkin sisältää tarvittavan määrän informaatiota tesauruksen rakentamiseksi.

Tutkielmassa käydään läpi eri menetelmiä hierarkkisen datan tallentamiseen ja noutamiseen relaatiotietokantaa käyttäen. Tämän jälkeen esitellään esimerkkitoteutus, joka tuottaa XML- ja HTML-muotoiset tesaurukset. XML-dokumentti mahdollistaa tesauruksen tehokkaan jatkokäytön. HTML-dokumenttiversio toteuttaa relaatiotietokannan datalle tesaurusmuotoisen käyttöliittymän. Esimerkkitoteutus osoittaa, että rekursiiviset tietokantakyselyt voivat olla tehokas menetelmä hierarkkisen sanakirjadatan noutamiseen relaatiotietokannasta. Toteutus myös havainnollistaa, miten tarpeeksi joustavaa relaatiotietokantaa on mahdollista käyttää tavoilla, jotka eivät sisältyneet relaatiotietokannan alkuperäisiin vaatimuksiin.

Avainsanat: relaatiotietokanta, hierarkia, tesaurus, rekursio, XML, HTML

1 Johdanto

1.1 Relaatiotietokannat ja hierarkkinen data

Relaatiotietokannat ovat perinteinen työkalu rakenteellisen datan tallentamiseen, ja ne ovat edelleen relevantteja 50 vuotta alkuperäisen idean kehittämisen jälkeen. Teknologiana relaatiotietokannat ovat kypsiä ja tuttuja suurimmalle osalle kehittäjistä. Tämän takia ne ovatkin yksi yleisimmistä tietokantatyypeistä, mutta niihin liittyy omat heikkoutensa. Yksi niistä on hierarkkisen datan käsittely. Tämä johtuu siitä, että relaatiotietokantojen pohja, relaatiomalli, ei suoraan tue hierarkioita. Tähän heikkouteen löytyy kuitenkin ratkaisunsa, joihin tutustumme myöhemmissä luvuissa. Tästä huolimatta, relaatiotietokannat ovat kohdanneet kritiikkiä esimerkiksi sanakirjojen säilyttämisen suhteen, koska on ajateltu, että relaatiotietokannat eivät pysty kuvaaman kaikkea sanakirjan leksikaalisen datan rakenteen ominaisuuksia [Ide et al., 1994]. Hierarkkisen datan käsittelyä hankaloittaa myös se, että tyypillisesti relaatiotietokantojen kyselykielien antamat vastaukset ovat litteitä. Tämä tarkoittaa sitä, että kun hierarkkista dataa noudetaan relaatiotietokannasta, vastaus ei ole järjestetty hierarkian mukaisesti, vaan se on pelkkä lista hierarkian elementeistä.

Tässä työssä keskitytään leksikografiseen dataan ja tarkemmin relaatiotietokantaan, johon on tallennettu sanakirjan leksikografinen data. Hierarkiat ja osa-kokonaisuussuhteet ovat oleellisia leksikografista dataa tarkastellessa. Tesauruksien hierarkia ja käsitteiden väliset suhteet perustuvat osa-kokonaisuussuhteisiin. Tämä tarkoittaa sitä, että tesaurukset koostuvat niiden eri tasoilla olevista osista ja näiden osien välisistä suhteista [Junkkari 2005]. Työssä esitellään menetelmä, jota käytettiin tesauruksen muodostamiseen tämän kyseisen leksikografisen datan perusteella. Esitelty menetelmä on kuitenkin käytettävissä myös muun kuin leksikografisen datan käsittelyssä. Lopputuloksena menetelmä tuottaa tuloksen XML- ja HTML-merkintäkielillä. Tuotettu tesaurus päätettiin esittää merkintäkielillä, koska sen hierarkkisen leksikografisen datan esittäminen onnistuu niillä yksinkertaisesti.

1.2 Digitaaliset sanakirjat

Tietojenkäsittelyn näkökulmasta sanakirjat voidaan nähdä kokoelmana tarkasti järjesteltyjä typografisia elementtejä, joiden sisältö ja suhteiden muodostama rakenne ovat niiden merkityksen kannalta lähes yhtä tärkeitä. Siinä missä perinteisissä paperisanakirjoissa erilaisia rakenteita kuvattiin typografian avulla, digitaalisessa sanakirjassa se ei ole tarpeellista, eikä edes optimaalista. [Lemnitzer et al., 2009] Käyttäjälle sanakirjan rakenteet voidaan visuaalisesti esittää typografian keinoin, mutta jos sanakirjan tallennusmuoto ottaa rakenteet huomioon, on tietokoneidenkin mahdollista ymmärtää sanakirjan rakenteita. Tämä mahdollistaa sen, että digitaalisessa versiossa

rakenteita voidaan hyödyntää usealla eri tavalla, mahdollisesti jopa tavoilla, joita ei osattu alkuperäistä versiota rakentaessa ennustaa.

Relaatiotietokannoissa data tallennetaan tauluihin, joissa sarakkeet vastaavat kohteen attribuutteja ja rivit vastaavat itse tallennettavia kohteita. Näiden kohteiden välille voidaan määritellä taulujen sisällä ja taulujen välillä relaatioita, käyttäen hyväksi mahdollisuutta viitata kohteille määriteltyihin uniikkeihin tietokantatunnisteisiin. Tietokantatunniste voi olla esimerkiksi kokonaisluku, jonka arvo täytyy olla tietokantataulun jokaisella sarakkeella uniikki. Tyypillisesti relaatiotietokantojen hallintaan käytetään SQL-kyselykieltä. Se mahdollistaa taulujen mallien määrittelemisen sekä datan lisääminen. Merkittävin ominaisuus lienee kuitenkin datan noutaminen tietokannasta erilaisilla SQL-kyselyillä, joissa voi määritellä mitä dataa haluaa ja millä perusteella se valikoidaan.

Perinteisesti on ajateltu, että relaatiotietokannat eivät sovellu kovin hyvin esimerkiksi sanakirjan leksikaalisen datan säilyttämiseen, koska ne eivät pysty kuvaaman kaikkea datan rakenteen ominaisuuksia. Toisaalta myös esimerkkejä onnistuneista digitaalisista sanakirjoista relaatiotietokannassa löytyy [Norri et al., 2020]. Relaatiotietokannan käyttäminen leksikaalisena tietokantana on mahdollista, mutta se tällöin tietokannan rakenne on suunniteltava tarkasti [Vaquero et al., 2014]. Tämän tutkielman yhteydessä toteutettu sovellus käyttää nimenomaan tätä Norrin ja muiden [2020] esittelemää relaatiotietokantaa, ja generoi sinne tallennetun datan ja relaatioiden perusteella XML- ja HTML-muotoisen tesaurusversion sanakirjasta.

Tämä relaatiotietokanta sisältää kokonaisen englanninkielisen historiallisen lääketieteellisen sanakirjan. Toteutetun ohjelman kannalta kiinnostavia tauluja ovat merkitys- ja termi-taulut, koska toteutus generoi tietokannasta tesauruksen juurimerkityksille ja juuritermeille. Molemmat näistä tauluista sisältävät uniikin tunnistesarakkeen sekä sarakkeen, jossa viitataan kohteen vanhempaan. Tämä viittaus on siis relaatio, joko merkityksen tai termin ylätermiin tai ylämerkitykseen. Yksinkertaisimmillaan pelkästään tämä tieto riittää relaatioiden läpi kulkemiseen rekursiivisesti, jolloin voimme kerätä tiedot hierarkkisessa puurakenteessa.

Seuraavaksi luvussa 2 tutustutaan merkintäkieliin ja hierarkkiseen dataan. Merkintäkieliä käytetään usein hierarkkisen datan kuvaamiseen. Kolmannessa luvussa taas tarkastellaan relaatiomallia, johon perustuvat relaatiotietokannat korvasivat hierarkkiseen malliin perustuvat tietokannat. Samassa luvussa käydään myös läpi eri keinoja hierarkkisen datan tallentamiseen relaatiotietokantaan. Neljännessä luvussa esitellään erilaisia menetelmiä hierarkkisien mallien generoinnille. Viides ja kuudes luku esittelevät toteutuksessa käytetyt menetelmät. Lopuksi käydään läpi yhteenveto, viittaukset ja liitteet.

2 Merkintäkielet ja hierarkia

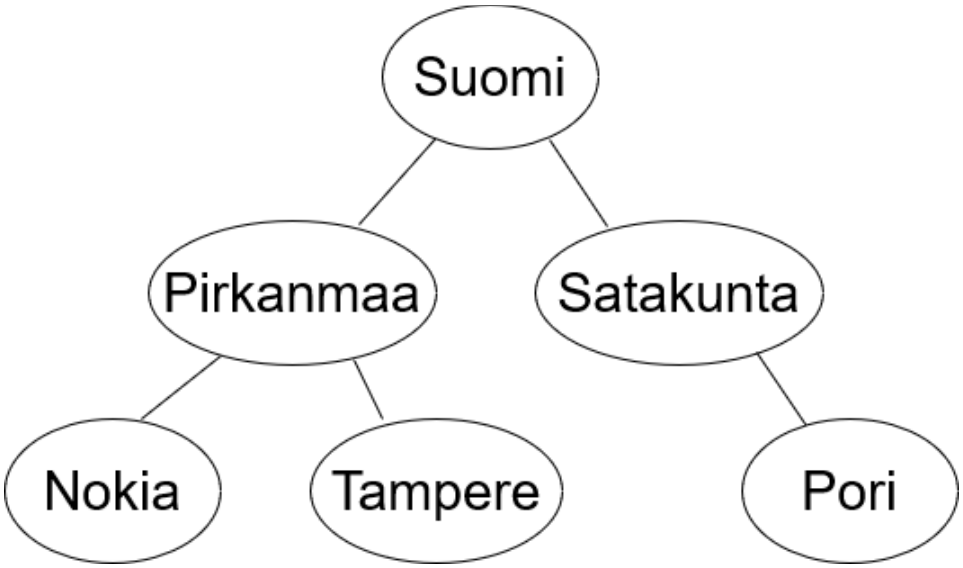
2.1 Hierarkkinen malli

Puumaista tietorakennetta, jossa jokaisella elementillä voi olla yksi vanhempi ja monta lasta, kutsutaan hierarkkiseksi. Hierarkkisen datan rakenne alkaa yhdestä juuresta ja haarautuu yhteen tai useampaan oksaan. Jokainen oksa voi edelleen haarautua uudelleen muodostaen näin hierarkkisen rakenteen. XML ja HTML ovat tyypillisiä esimerkkejä hierarkkisesta datasta, jossa kaikilla dokumentin elementeillä on yksi vanhempi, mutta teoriassa niillä voi olla rajaton määrä lapsia. Hierarkkisessa datassa jokaiseen elementtiin on vain yksi polku. Tämän takia hierarkkisen datan rakenne on ymmärrettävissä vain yhdellä tavalla. Sen käsittelijä voi olettaa, että jokaisella oksalla on täsmälleen yksi vanhempi. Täysin litteään listaan verrattuna datan tallentaminen hierarkkisella rakenteella voi välttää datan toistamista, koska data voidaan järjestää niille yhteisen elementin lapsiksi. Jokaisen elementin ei siis tarvitse esimerkiksi sisältää vanhempansa nimeä, koska datan rakenteen ollessa hierarkkinen on vanhemman nimi noudettavissa suoraan vanhemmalta. Tämä taas on triviaalia, koska jokaisen elementin ainut vanhempi on tiedossa.

Hierarkkiseen malliin perustuu myös tietokantamalli, joka edeltää nykyään yleisimpiä relaatiotietokantoja. Koska hierarkkinen tietokantamalli noudattaa samoja sääntöjä kuin muukin hierarkkinen data, siihen liittyy ongelmia, joilta vältytään relaatiotietokantaa käyttäessä. Hierarkkiseen malliin perustuva tietokanta ei esimerkiksi pysty tukemaan monen suhde moneen -yhteyksiä. Tämä asettaa rajoituksen tietokannan rakenteelle ja voi johtaa tarpeeseen toistaa tietokannassa säilytettävää dataa [Harrington, 2016]. Hierarkia asettaa myös muita rajoituksia datan käsittelylle ja säilyttämiselle. Kaiken tietokantaan syötetyn datan täytyy olla jonkun hierarkian juuressa sijaitsevan elementin lapsi. Kaikki elementin lapset poistuvat tietokannasta elementin poiston yhteydessä. Näistä rajoituksista huolimatta hierarkkiseen malliin perustuvien tietokantojen vahvuuksiksi voidaan mainita mallin yksikertaisuus. Osa näistä heikkouksista voidaan myös kiertää. Monen suhde moneen -yhteyksien toteuttaminen ei suoraan ole mahdollista, mutta duplikaatti datan tallentamista on käytetty ongelman kiertämiseen. Lapsielementtien säilyttäminen tietokannassa poiston jälkeen on myös mahdollista, jos elementin poistamisen sijaan vain sen sisältämä data pyyhitään pois tietokannasta [Tsichritzis and Lochovsky, 1976].

Maan jakaminen maakuntiin ja edelleen kuntiin on esimerkki hierarkiasta, kuten näemme kuvassa 1. Tämän esimerkin hierarkiaan voi muiden maakuntien ja kaupunkien lisäksi lisätä myös uusia tasoja, kuten esimerkiksi maanosat tai kunnan sisäiset alueet. Kuvitteellisessa tilanteessa, jossa Pirkanmaa liitettäisiin Satakuntaan, olisi kuvan 1 hierarkian päivittäminen toteutettavissa yksinkertaisesta päivittämällä Pirkanmaan lapsien vanhempi Satakunnaksi. Päivittämisen jälkeen Pirkanmaa poistettaisiin

hierarkiasta. Tämä tilanne havainnollistaa yhden hierarkkisen datan eduista. Päivitys periytyy kaikille kuntien lapsille, kun kunnan vanhempi päivitetään.



Kuva 1. Osa Suomen maakunnista ja kunnista hierarkiassa.

2.2 Merkintäkielet

Merkintäkielet ovat kieliä, joita käytetään dokumenttien määrittelemiseen. Merkintäkielet ovat luettavassa formaatissa niin ihmisille kuin tietokoneillekin. Tämä mahdollistaa niiden kirjoittamisen sekä suoraan tekstinkäsittelyohjelmalla tai merkintäkielen kirjoittamiseen tarkoitetulla sovelluksella. Merkintäkielet on myös mahdollista erikseen prosessoida visuaaliseen muotoon, jolloin niillä määriteltävä dokumentti voi muodostaa esimerkiksi tietokoneohjelman graafisen käyttöliittymän pohjan. Dokumentti ei siis tarkoita tässä asiayhteydessä pelkästään esimerkiksi sanakirjaa tai opinnäytetyötä. Se määritellään kokoelmana informaatiota, joka prosessoidaan kokonaisuutena ja määritellään tietyn tyyppiseksi dokumentiksi [Maler and Andaloussi, 1995]. Jos merkintäkielellä luo sanakirjan, se on määritelty myös merkintäkielen näkökulmasta sanakirjaksi. Digitaaliset sanakirjat tallennetaan tyypillisesti jollain merkintäkielellä, koska merkintäkielet sallivat vapaan rakenteen kuvaamisen. Hierarkkisen datan esittäminen merkintäkielillä on tyypillistä, koska merkintäkielet sallivat leksikaalisen datan alkuperäisen hierarkian säilyttämisen digitaalisessa muodossa. Sanakirjan muuntaminen XML-merkintäkieleksi on todennäköisesti yleisin tapa digitalisoida sanakirja [Norri et al., 2020].

2.3 SGML

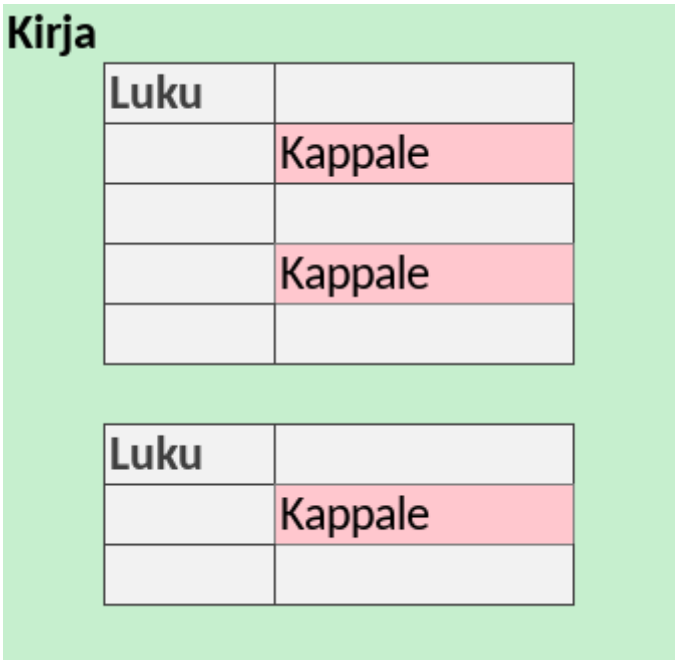
SGML (Standard Generalized Markup Language, ISO 8879) on 80-luvulla kehitetty merkintäkieli, joka on sukua nykyään suosituimmille merkintäkielille. SGML on myös metakieli, mikä tarkoittaa, että sitä voidaan käyttää muiden merkintäkielien

määrittelemiseen. Esimerkiksi osa HTML-merkintäkielen versioista on määritelty sillä. Jokainen SGML-dokumentti koostuu kahdesta osasta. Ensimmäinen osa on dokumentti-instanssi, joka voisi sisältää esimerkiksi sanakirjan sisällön ja toinen osa on DTD-tiedosto (Document Type Declaration). DTD määrittelee dokumenttityypin säännöt, joita kaikkien kyseisen dokumenttityypin instanssien tulee noudattaa [Maler and Andaloussi, 1995].

SGML-merkintäkieli kehitettiin ratkaisemaan standardin puute koneluettavalle datalle. Standardi tarvittiin datan siirtämiseen tietokoneiden ja sovelluksien välillä. Tavoitteena oli myös suurien data määrien pitkäaikainen säilyttäminen koneluettavassa muodossa. SGML-merkintäkieli mahdollistaa nämä tavoitteet, koska se ei ole riippuvainen mistään tietystä sovelluksesta tai tietyn tyyppisestä tietokoneesta. SGML-dokumenttia käsittelevän sovelluksen täytyy vain tuntea standardi SGML-merkintäkieli ja pystyä tulkitsemaan SGML-dokumentin DTD-tiedoston määrittelemät säännöt [Cover et al., 1991].

2.4 XML

XML (Extensible Markup Language) on Word Wide Web Consortiumin 90-luvun lopussa kehittämä, tekstin rakennetta kuvaileva merkintäkielistandardi. Tämän merkintäkielen alkuperäisiin tavoitteisiin kuului helppo käytettävyys internetissä ja laaja tuki eri sovelluksissa. Se suunniteltiin myös yhteensopivaksi SGML-merkintäkielen kanssa [W3C, XML, 2021]. Yksinkertaisuus ja ihmisluettavuus olivat tärkeitä ominaisuuksia sen suunnittelussa. Koska XML suunniteltiin yhteensopivaksi SGML-merkintäkielen kanssa, DTD-tiedostot toimivat myös XML-merkintäkielen kanssa. Käytännössä XML määrittelee syntaksin, jossa erilaiset elementit muodostetaan elementin avaavista ja sulkevista merkinnöistä. Elementtejä voi XML-dokumentissa olla sisäkkäin ja rinnakkain rajattomasti, mistä seuraa, että kieli mahdollistaa erilaisien monimutkaisia puurakenteiden kuvaamisen juuri- ja lapsielementtien avulla. XML tarjoaa tarvittavat työkalut elementtien rakenteiden sääntöjen määrittelemisen dokumentissa. Tämä johtaa siihen, että XML mahdollistaa hierarkkisen datan kuten sanakirjojen tallentamisen [Lemnitzer et al., 2009]. Kuvassa 2 on dokumentin malli, jossa on elementtejä rinnakkain ja sisäkkäin. Tämä dokumentti voisi näyttää XML-muodossa samalta kuin liite 1.



Kuva 2. Elementtien hierarkia.

2.5 HTLM

HTML (Hypertext Markup Language) on merkintäkielistandardi, jota käytetään pääasiassa verkkosivujen mallintamiseen. Alkuperäinen käyttötarkoitus oli tieteellisien julkaisujen kirjoittaminen [W3C, HTML, 2021]. Kuten XML, myös HTML perustuu tagien avaamiseen ja sulkemiseen. Myös HTML-tageja voi olla sisäkkäin ja rinnakkain rajattomasti. Se muodostaa puumaisen rakenteen, joka koostuu elementeistä ja tekstistä. Periaatteessa kokonaisen sanakirjan voisi kuvata yhtenä hierarkkisena HTML-dokumenttina, mutta tämä ei ole tarpeellista. Yksi HTML:n määrittelevistä ominaisuuksista on sen hypertekstuaalisuus. Dokumentit voivat sisältää hyperlinkkejä, jotka ohjaajat käyttäjän dokumentista toiseen. Tämä mahdollistaa esimerkiksi sanakirjan hierarkkisen puunrakenteen esittämisen käyttäjälle loogisiin osiin jaettuna. Kun kokonaisuus on jaettu useampaan HTML-dokumenttiin, jotka on liitetty toisiinsa hyperlinkkien kautta, käyttäjän on mahdollista vapaasti liikkua rakenteessa loogisien kytköksien kautta. Käyttäjän selatessa internettiä hänen verkkoselaimensa lataa verkkosivun HTML-muotoisen dokumentin ja esittää sen käyttäjälle graafisessa muodossa. HTML-dokumenttien hypertekstuaalisuus ja niiden esittäminen käyttäjälle graafisessa muodossa mahdollistavat verkkosivut nykyisessä muodossaan [Bouvier, 1995].

3 Relaatiokannat ja hierarkian käsittely

3.1 Relaatiomalli

Relaatiomalli voidaan kuvailla mallina, jonka koko looginen rakenne muodostuu relaatioista. Edgar Codd kehitti relaatiomallin 1960-luvulla laajentamalla joukko-opin binäärirelaation käsitettä. Tarkoituksena oli luoda tietomalli korvaamaan Coddin mielestä kömpelöt aikaisemmat tietomallit [Harrington, 2016]. Nykyään suurin osa tietokannoista perustuvat relaatiomalliin. Näitä tietokantoja kutsutaan relaatiomallin nimen mukaisesti relaatiotietokannoiksi. Relaatiotietokantoja edelsivät esimerkiksi hierarkkiseen malliin perustuvat tietokannat. Relaatioita kuvataan relaatiotietokannassa tietokantatauluilla, jotka koostuvat sarakkeista ja riveistä. Jokaista riviä vastaa yksi tietokantaan tallennettu kohde, jonka attribuutteja vastaavat sarakkeet. Esimerkiksi, kohde voisi olla henkilö, jolla on attribuutteina nimi ja kaupunki. Rivien järjestyksellä relaatiotietokannan tauluissa ei ole merkitystä ja jokainen niiden sarakkeista on nimetty.

Relaatiotietokannoissa jokaisen rivin täytyy sisältää pääavain, jonka perusteella se voidaan tunnistaa. Usein pääavain on uniikki kokonaisluku tietyssä sarakkeessa, mutta se voi olla myös uniikki yhdistelmä tiettyjen sarakkeiden arvoista. Näiden tunnisteiden avulla pystymme yhdistämään eri tietokantatauluissa olevia kohteita. Tämä onnistuu esimerkiksi tallentamalla työntekijälle työnantajan pääavain vierasavaimena. Vierasavain on tapa viitata toiseen riviin joko samassa tietokantataulussa tai eri tietokantataulussa. Viiteavain työnantajaan mahdollistaisi relaatiotietokannasta datan noutamisen niin, että voimme esimerkiksi rajata tuloksen vain haluamamme työnantajan työntekijöihin. Kuvassa 3 tietokantataulussa on esimerkin mukainen vierasavain.

Työntekijät

Tunniste	Nimi	Kaupunki	Esimies
1	Artola	Tampere	
2	Aho	Helsinki	1
3	Leppo	Seinäjoki	1
4	Linnala	Helsinki	2
5	Palosuo	Espoo	2
6	Puro	Vantaa	3

Kuva 3. Tietokantataulu relaatiotietokannassa.

3.2 Relaatiotietokantojen suunnittelu ja hallinta

Yleinen tapa relaatiotietokantojen suunnitteluun on ER-mallin (Entity-Relationship model) käyttäminen. ER-malli perustuu datan kuvaamiseen entiteetteinä, joilla on

attribuutteja ja elementtien välisiä suhteita. Entiteetit ovat erotettavissa toisistaan niiden spesifien attribuuttien perusteella. Entiteetillä kuvataan usein objektia, jolla on fyysinen vastine kuten esimerkiksi henkilö tai rakennus, mutta ne voivat olla myös abstraktimpia objekteja. Esimerkiksi sanakirjan ER-mallissa sanan merkitys vois olla entiteetti, jolla on attribuutteja ja suhteita. Tämän ER-mallin perusteella tehdyssä relaatiotietokannassa merkitys-entiteettiä vastaisi tietokantataulu, joka sisältää sen attribuutit ja mahdollisesti osan sen suhteista.

Entiteettien välisillä yhteyksillä on seuraavat kardinaalisuudet: yhden suhde yhteen, yhden suhde moneen ja monen suhde moneen [Murthy, 2008]. Nimensä mukaisesti yhden suhde yhteen -yhteydet ovat yhteyksiä, joita jokaisella entiteetillä voi olla vain yksi ja se kohdistuu vain yhteen entiteettiin. Yhden suhde moneen -yhteydet taas sallivat yhteyden suunnasta riippuen yhden tyyppiselle entiteetille yhteyden useaan toisen tyyppiseen entiteettiin, joilla taas voi olla vain yksi yhteys ensimmäisen tyyppiseen entiteettiin. Monen suhde moneen -yhteys sallii rajattoman määrän yhteyksiä molemmille entiteeteille. Länsimainen avioliitto on yksi esimerkki yhden suhde yhteen -yhteydestä. Ystävyyssuhteet taas ovat esimerkki monen suhde moneen -yhteyksistä. Kuvassa 7 näemme esimerkin ER-mallista. Siinä relaatiotietokannan taulut on kuvattu entiteetteinä, joita edustaa laatikot. Niiden väliset yhteydet on merkitty viivoilla, joissa on keskellä timantti, jonka vieressä on merkitty yhteyden kardinaalisuus käyttäen 1-merkkiä, n-merkkiä ja m-merkkiä. Kun kaksi entiteettiä on yhdistetty samaan timanttiin viivoilla, on timantin ja molemman entiteetin välissä yksi näistä kolmesta merkistä. Kaikki eri kardinaalisuudet kuvataan käyttäen näitä merkkejä. Tietokannan taulujen kohdalla 1-merkki tarkoittaa, että yhteydessä on mukana kerrallaan yksi taulun riveistä. Molemmat n-merkki ja m-merkki tarkoittavat, että yhteydessä voi olla mukana yksi tai useampi rivi. Variaatioita on kolme siten, että 1-merkki ja m-merkki eivät esiinny samassa yhteydessä. Nämä kolme variaatiota riittävät kaikkien kardinaalisuuksien kuvaamiseen. ER-mallissa entiteettiin liittyvät attribuutit voidaan esittää entiteettiin liitetyissä soikioissa. Nämä attribuutit on jätetty kuvan 7 yksinkertaistetusta mallista pois.

SQL (Structured Query Language) on kyselykieli, joka on tarkoitettu relaatiotietokantojen käsittelemiseen. Sitä käyttämällä relaatiotietokantaan voi tehdä muutoksia, lisäyksiä ja hakuja. SQL-kyselykieli voidaan jakaa kolmeen osaan: tietokannan rakenteen määrittely, tietokannan käyttäjien oikeuksien määrittely ja tietokannan datan käsittely [Oppel, 2015]. Rakenteen määrittely tarkoittaa tietokantataulujen luomista. Niille määritellään attribuutit, suhteet ja muut säännöt. Monipuolinen oikeuksien määrittely mahdollistaa eri käyttäjille eri oikeudet tietokantaan. Monimutkaisin osa SQL-kyselykielestä on sillä suoritettava datan käsittely. Select-lauseilla voi määritellä mitä dataa haluaa ja mistä tauluista. Oletuksena dataa haetaan vain yhdestä taulusta, mutta esimerkiksi join- ja union-operaatiot mahdollistavat

monimutkaisemmat select-lauseet, joissa yhdistellään eri taulujen sisältämää dataa. Select-lauseiden lisäksi käytettävissä on insert- ja update-lauseet datan lisäämiseen ja muokkaamiseen. Delete-lausetta käytetään datan poistamiseen. Liitteessä 6 on nähtävissä esimerkki rekursiivisesta SQL-kyselystä.

3.3 Tavat tallentaa hierarkkista dataa relaatiotietokantaan

Vaikka relaatiotietokantoja ei varsinaisesti tunneta niiden joustavuudesta, mahdollistavat ne kuitenkin datan tallentamisen useilla erilaisilla tavoilla. Käydään seuraavaksi läpi eri tapoja, jotka relaatiotietokannat mahdollistavat erityisesti hierarkkisen datan näkökulmasta. Tavoitteena kaikissa tallennustavoissa on, että ne sisältävät tarpeellisen määrän informaatiota, jonka perusteella hierarkkinen data voidaan uudelleen rakentaa sen alkuperäiseen muotoon. Uudelleen rakentaminen tapahtuu tekemällä relaatiotietokantaan yksi tai useampi kysely, ja mahdollisesti käsittelemällä noudettua dataa erillisellä sovelluskoodilla.

Kaikkein yksinkertaisin tapa tallentaa hierarkkista dataa relaatiotietokantaan on vieruslistan (Adjacency List) käyttäminen. Vieruslista on verkkoteoriassa tapa esittää verkko. Vieruslistaesityksessä verkon solmut sisältävät listan naapureistaan. Verkot voivat olla joko suunnattuja tai suuntaamattomia. Suuntaamattomassa verkossa molemmat naapurisolmut sisältävät tiedon niiden välisestä kaaresta. Suunnatussa verkossa tieto kahden solmun välisestä kaaresta ei välttämättä ole molemmissa solmussa. Jos suunnattu verkko käyttää vieruslistaesitystä, sen solmut sisältävät tiedon vain osasta niiden naapureista [Cormen et al., 2009]. Kuvassa 3 näemme esimerkin relaatiotietokannan taulusta, jossa jokaiselle työntekijälle, jolla on esimies, on tämän tietokantatunniste tallennettu suoraan yhdeksi sen attribuuteista. Jos ajattelemme tämän taulun verkkona, näemme että kyseessä on suunnattu verkko, joka esitetään käyttäen vieruslistaesitystä. Verkossa kaarien suunta on työntekijän solmusta esimiehen solmuun. Kuvan 3 mukainen tapa tallentaa hierarkia relaatiotietokantaan on siis vieruslistaesitystä käyttävä verkko, koska verkko on määritelty niin, että solmuun tallennettu esimiehen tietokantatunniste on solmun lista naapureista. Yksi tietokantatunniste riittää vieruslistaksi, koska esimiehiä on vain yksi. Tämä on yksinkertaisin ja siksi myös yleisin tapa tallentaa hierarkkista dataa relaatiotietokantaan. Dataa voidaan noutaa yksinkertaisilla kyselyillä ja se mahdollistaa tarvittaessa myös monimutkaisempien rekursiivisien kyselyiden käyttämisen, mikäli käytetty relaatiotietokanta tukee tätä ominaisuutta.

Polkulaskenta (Path Enumeration) on seuraava tarkastelemamme tapa tallentaa hierarkkista dataa relaatiotietokantaan. Se perustuu siihen, että jokaiselle rakenteen osalle tallennetaan täysi polku sen vanhemmista, yhtenä sen attribuuteista [Celko, 2012]. Jos kuvan 3 esimerkki muunnettaisiin käyttämään polkulaskentaa, voitaisiin esimies-sarake korvata esimiespolku-sarakkeella. Tämä polku olisi siis merkkijono, joka sisältäisi tiedon

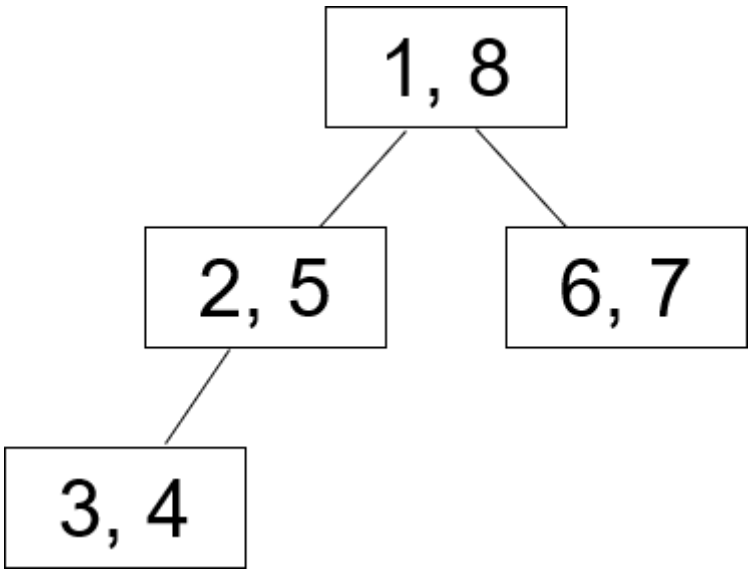
jokaisen rivin esimiehestä, esimiehen esimiehestä ja niin edelleen. Käytännössä tunnisteella 2 esimiespolku-merkkijonon arvo olisi sama 1 kuin se on nytkin, mutta esimerkiksi tunnisteella 6 kentässä olisi eri arvo. Tunnisteelle 6 voitaisiin sarakkeeseen laittaa arvoksi esimerkiksi “1/3”. Tämä tarkoittaisi siis sitä, että tunniste 3 on sen suora esimies, ja tunnisteen 3 esimies on tunniste 1. Koska relaatiotietokantaan tämä tieto täytyisi tallentaa merkkijonona, on /-merkki jakamassa polun osat. Sen tilalla voisi tietysti käyttää jotain muutakin merkkiä. Hyvänä puolena tässä on, että hierarkian koko polku on suoraan jokaiselle riville, mutta käytännössä tässä ratkaisussa on ongelmansa. Merkkijonojen käyttäminen tekee kyselyjen rakentamisesta monimutkaisempaa, ja myös rakenteen ylläpito muuttuu raskaammaksi. Jos hierarkia muuttuu, se tarkoittaa, että jokaista esimiespolku-merkkijonoa, jota päivitys koskee, täytyy myös erikseen päivittää vastaamaan uutta hierarkiaa. Tämä johtuu siitä, että kyseisen merkkijonon ylläpitäminen on täysin relaatiotietokannan käyttäjän vastuulla.

Kolmas vaihtoehto, jota tarkastelemme, on sisäkkäiset joukot (Nested Set), joka perustuu hierarkkisen datan läpi kävelemiseen ja numerointiin. Tietorakenteen jokaisella solmulla on kaksi numeroa, jotka kertovat sen sijainnista suhteessa muihin solmuihin. Solmut kävellään läpi yksi kerrallaan niin, että jokaisessa solmussa käydään kahdesti. Juokseva luku laskee jokaisen solmussa käynnin. Kun solmuun saavutaan ensimmäisen kerran, se saa sen hetkisen juoksevan luvun ensimmäiseksi järjestysnumerokseen. Tämän jälkeen kävellään solmun lapsiin ja kun kaikissa solmun lapsissa on käyty, palataan takaisin solmuun, jolloin se saa toisen järjestysnumeronsa. Kun solmuun palataan, on juokseva luku kasvanut vähintään yhdellä ja jos solmulla on lapsisolmuja se kasvaa 1 + 2 * n, missä n on lapsisolmujen määrä. Tämän prosessin jälkeen lapsisolmujen ensimmäinen järjestysnumero on aina suurempi kuin vanhempisolmun ja samalla lapsisolmujen toinen järjestysnumero on aina pienempi kuin vanhempisolmujen. Sisäkkäisiä joukkoja ja sen järjestysnumeroita havainnollistetaan kuvassa 4. Tässä kuvassa juurisolmun järjestysnumerot ovat 1 ja 8. Juurella on siis pienin ja suurin järjestysluku, koska kaikki muut solmut ovat sen lapsia. Myös sisäkkäisien joukkojen käyttäminen tarkoittaa, että lisäyksien yhteydessä osaa muistakin solmuista joudutaan päivittämään. Tämä johtuu siitä, että jos puuhun lisätään solmu, joudutaan mahdollisesti rakenteessa jo käytössä olevia järjestysnumeroita siirtämään tälle uudelle solmulle.

Tästä seuraa se, että osa järjestysnumeroista joudutaan laskemaan uudelleen. Kaikkia järjestysnumeroita, jotka ovat yhtä suuria tai suurempia kuin lisätyn solmun numerot, täytyy kasvattaa kahdella, koska uudella solmulla on kaksi järjestysnumeroa. Jos taas poistetaan solmu, järjestysnumeroita ei voida yksinkertaisesti laskea uudelleen tekemällä lisäyksen vastakohtaa, koska jos poistettavalla solmulla on lapsisolmuja, myös nämä lapsisolmut poistuvat. Tämän takia järjestysnumeroiden uudelleen laskemisessa täytyy ottaa huomioon poistuvan solmun järjestysnumeroiden erotus. Kaikista

järjestysnumeroista, jotka ovat suurempia kuin poistettavan solmun järjestysnumerot, vähennetään poistettavan solmun järjestysnumeroiden erotus + 1. Ensimmäinen eli pienempi järjestysnumero vähennetään suuremmasta eli toisesta järjestysnumerosta. [Gyorodi et al., 2016] testeissä nähdään, että vaikka sisäkkäiset joukot tarvitsevatkin ylläpitoa hierarkian muuttuessa, niitä käyttäessä relaatiotietokantaan tehtävät kyselyt ovat yksinkertaisempia – ja joissain tilanteissa suorituskykyisempiä – kuin vastaavat kyselyt, jos relaatiotietokannassa olisi käytetty vieruslistaa.

Jos esimerkiksi halutaan etsiä hierarkiasta kaikki solmut, joilla ei ole lapsia, täytyy vieruslistaa käyttäessä jokaisen solmun kohdalla selvittää viittaako toinen solmu siihen vanhempanaan. Vastaava kysely sisäkkäisien joukkojen kanssa onnistuu yksinkertaisesti tarkistamalla, kunkin solmun kohdalla, onko sen ensimmäinen järjestysluku yhtä kuin toinen järjestysluku + 1. Tämä on mahdollista, koska jos solmulla olisi yksi tai useampi lapsi, sen järjestyslukujen erotus olisi suurempi kuin 1.



Kuva 4. Sisäkkäisen joukon järjestysnumeroiden laskeminen.

Viimeinen tarkastelemamme tapa hierarkkisen datan tallentamiseen relaatiotietokantaan on sulkeumataulukko (Closure Table). Se perustuu erillisen tietokantataulun luomiseen, johon tallennetaan hierarkian relaatiot tallentamalla vanhempi- ja lapsisolmujen tunnisteparit. Hyvänä puolena tässä menetelmässä on se, että monesti relaatioihin voi liittyä tietoja, kuten esimerkiksi voimassaoloaika. Järjestelmän suunnittelija saattaa myös haluta jättää talteen relaatioiden historian. Nämä ominaisuudet eivät olisi yhtä aikaa mahdollisia pelkällä vieruslistalla, koska vieruslistaa käyttäessä jokainen solmu tietää vain sen hetkisen vanhempansa. Sulkeumataulukolla taas voimme jättää täyden historian talteen ja liittää jopa vanhoihin relaatioihin niihin liittyviä tietoja. Kuitenkin heikkoutena tässä menetelmässä on Chernyshin ja muiden [2020] mukaan se, että sulkeumataulukon mahdollistava tietokantataulu saattaa hierarkian koosta riippuen

kasvaa erittäin suureksi, millä voi olla huomattavia negatiivisia vaikutuksia suorituskykyyn.

On kehittäjien vastuulla vertailla näitä eri vaihtoehtoja, ottaen huomioon niiden vahvuudet ja heikkoudet. Toteutuksen tavoitteet voivat myös vaikuttaa siihen, mikä lähestymistapa on paras. Tarkempaa vertailua on suorittanut Chernysh ja muut [2020]. Testeissä vieruslistaa käytettiin yhdessä rekursiivisien tietokantakyselyiden kanssa. Tämä lähestymistapa on kaikista lähinnä tässä tutkielmassa esiteltävää toteutusta. Testien perusteella eri relaatiotietokantavaihtoehdot voivat vaikuttaa suorituskykyyn yhtä paljon kuin hierarkkisen datan säilytystapa. Sulkeumataulukko oli kuitenkin testissä keskimäärin suorituskykyisin, vaikka sen mahdollistavalla relaatiot säilyttävällä taululla voi olla joissain tilanteissa suorituskykyyn negatiivinen vaikutus. Huomioitava on myös se, että vieruslista rekursiivisien kyselyiden kanssa ja sulkeumataulukko ovat esitellyistä vaihtoehdoista ainoat, joiden eheyden relaatiotietokanta pystyy takaamaan, koska ne perustuvat olemassa oleviin tietokantatunnisteisiin viittaamiseen. Sisäkkäiset joukot ja polkulaskenta vaativat, että relaatiotietokantaan tallennettua hierarkiaa ylläpidetään sen muuttuessa.

4 Datan muuntaminen

4.1 Muuntamisen menetelmät

Relaatiotietokannoista on muodostunut yleinen tapa tallentaa erilaisien organisaatioiden dataa. Samaan aikaan XML-dokumenteista on tullut yksi tärkeimmistä työkaluista datan siirtämiseen [Fong and Shiu, 2012]. Kun tietoa halutaan siirtää esimerkiksi kahden relaatiotietokannan välillä siirtämällä data ensin XML-dokumentiksi, puhutaan datan muuntamisesta. Datan muuntaminen on prosessi, jossa datan muoto muutetaan toiseen formaattiin, säilyttäen kaikki dataan liittyvä informaatio ja semantiikka. Fong [2006] kuvaa 4 erilaista menetelmää, jolla data voidaan muuntaa formaatista toiseen.

Ensimmäinen menetelmä on kehittää kyseistä datanmuuntotapausta varten uniikki sovellus, joka lukee datan ensimmäisestä formaatista ja muuntaa sen toiseen formaattiin. Tämä menetelmä on toimiva, mutta se saattaa olla työläs ja kallis siihen nähden, että sovellusta ei välttämättä tarvita useammin kuin kerran. Sovellus suunnitellaan kyseistä käyttötarkoitusta varten, mistä seuraa, että sitä ei voida käyttää eri syötteen tai kohteen kanssa. Jos syöte tai kohde vaihtuu, täytyy tilannetta varten toteuttaa oma sovelluksensa. Esimerkiksi tässä tutkielmassa esitellään tämän ensimmäisen menetelmän mukainen sovellus, joka muuntaa dataa relaatiotietokannasta XML- ja HTML-formaatteihin. Molempia kohdeformaatteja muodostaessa relaatiotietokannasta data tallennetaan samaan väliaikaiseen muotoon. Tämän jälkeen väliaikaista dataa käytetään XML- ja HTML-tiedostojen muodostamiseen, mutta siinä vaiheessa prosessit eroavat toisistaan. Vaikka eri formaatteja varten täytyykin kirjoittaa omat toteutuksensa, hyötyivät molemmat toteutukset kuitenkin toisistaan, koska syötteen ollessa sama tietokanta, tietojen noutaminen pystyttiin tekemään lähes samalla tavalla.

Toinen menetelmä on käyttää tulkkaavaa muuntajaa. Tässä menetelmässä muuntaja sovellukselle luodaan määritelmät, joissa kuvataan syötedata, muunnoksen kohdedata ja niiden väliset yhteydet. Muuntaja tulkitsee määritelmiä ja muuntaa datan sen perusteella. Hyötynä tässä menetelmässä on, että jos tarvetta vastaava tulkkaava muuntaja on tarjolla, ei käyttäjän tarvitse itse ohjelmoida sovellusta. Toisaalta käyttäjän tarvitsee määritellä syötedatalle ja kohdedatalle mallit, ja näiden mallien määrittelemistä varten käyttäjä saattaa joutua opiskelemaan uuden mallin määrittelykielen.

Kolmas menetelmä on käyttää muuntajageneraattoria. Generaattorille annetaan määritelmät, joissa kuvataan syötedata, muunnoksen kohdedata ja niiden väliset yhteydet. Näiden määritelmien perusteella generoidaan lähdekoodi sovellukselle, joka pystyy toteuttamaan datan muunnoksen. Menetelmä on käyttäjäystävällinen, mutta lähdekoodin generoinnissa on omat ongelmansa. Määritelmien muuttuessa lähdekoodi täytyy aina generoida uudelleen, mutta jos aikaisemmin generoitua versiota on muokattu manuaalisesti, muutokset täytyy siirtää manuaalisesti myös uuteen versioon. Toinen

vaihtoehto on generoida kerran ja sen jälkeen kehittää sovellusta tarpeen tullen manuaalisesti eteenpäin ilman uusia generointeja.

Neljäs ja viimeinen [Fong, 2006] kuvaama menetelmä on loogisen tason muuntaminen. Tässä menetelmässä luodaan sovellus, joka ymmärtää syötteen ja kohteen rakenteet. Sovellus lukee syötedatan käytännölliseen välimuotoon ja syöttää välimuodon kohde formaatin. Koska sovellus pystyy tulkitsemaan syötteen rakennetta, kehittäjien ei tarvitse itse tutustua syötteen rakenteeseen. Menetelmän haittapuolena on kuitenkin se, että pitkälle viedyn automaation takia kehittäjällä ei ole kovin paljon mahdollisuuksia vaikuttaa lopputuloksen muotoon.

4.2 Muunnos relaatiotietokannasta hierarkkiseksi XML-dokumentiksi

Datan muuntamista relaatiotietokannasta XML-muotoiseksi voi lähestyä usealla eri tavalla, mutta tarkastelemme seuraavaksi [Fong et al., 2001] esittelemää menetelmää. Siinä tuotetaan relaatiotietokannan rakenteen pohjalta välimuotomalli, jonka perusteella generoidaan tätä mallia vastaava XSD-muotoinen XML-skeema.

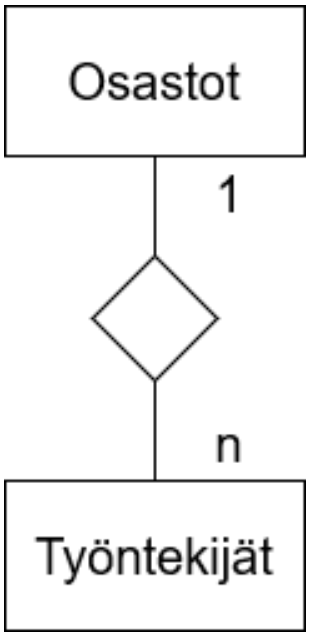
Tämä prosessi alkaa relaatiotietokannan rakenteen analysoinnilla. Analyysin perusteella relaatiotietokannan rakenne kuvataan EER-mallina. EER-malli laajennettu versio normaalista ER-mallista, joka on tarkoitettu nimenomaan tietokantojen mallintamiseen [Fidalgo et al., 2012]. Kun relaatiotietokannan rakenne on kuvattu EER-mallissa, seuraavana vaiheena on sen muuttaminen XML-skeemaksi. Nämä XML-skeemat ovat XML-dokumentteja, jotka määrittelevät niitä käyttävien XML-dokumenttien rakenteen ja sisältämän datan muodon. Se on formaatti, jota sekä ihmiset että tietokoneet voivat lukea. [W3C, XML Schema, 2021]. EER-mallin muuntaminen XML-skeemaksi tapahtuu vaihtamalla EER-mallin osat niiden XML-vastikkeiksi. Esimerkiksi [Fong et al., 2001] muuntaa EER-mallin entiteetit XML-elementiksi ja relaatiot href-referenssiksi. XML-merkintäkielen joustavuuden ansiosta XML-skeeman toteuttajan ei kuitenkaan tarvitse noudattaa tässä vaiheessa mitään tiettyä kaavaa, vaan hän voi suunnitella XML-skeeman oman tarpeensa mukaisesti. Kun XML-skeema on valmis, sen avulla voidaan syöttää relaatiotietokannan sisältö varsinaiseen XML-dokumenttiin.

Edellä kuvattu prosessi on vain yksi mahdollinen tapa muuntaa relaatiotietokannan rakenne XML-formaattiin. Yksinkertaisessa esimerkissä XML-tiedoston lopullinen formaatti saattaa olla lähes itsestään selvä, mutta tarkastelemme kuitenkin seuraavaksi askel kerrallaan yksikertaisen esimerkin yksittäiset vaiheet. Jos laajennamme kuvan 1 esimerkkiä kuvassa 5 voimme käyttää sitä esimerkkinä tässä muunnosprosessissa. Kuvassa 6 taas näemme tämän yksinkertaisen relaatiotietokannan perusteella tehdyn EER-mallin.

Osastot	
Tunniste	Kaupunki
1	Tampere
2	Helsinki
3	Seinäjoki

Työntekijät			
Tunniste	Nimi	Kaupunki	Osasto
1	Artola	Tampere	1
2	Aho	Helsinki	2
3	Leppo	Seinäjoki	3
4	Linnala	Helsinki	2
5	Palosuo	Espoo	2
6	Puro	Vantaa	3

Kuva 5. Relaatiotietokanta osastot- ja työntekijät-tauluilla.



Kuva 6. EER-malli kuvan 5 relaatiotietokannasta.

Kun esimerkkirelaatiotietokantamme on mallinnettu EER-malliksi, voimme aloittaa XML-skeeman muodostamisen. Osasto- ja työntekijä-entiteetit muunnetaan XML-skeemassa elementeiksi. Niiden välinen relaatio mallinnetaan tässä esimerkissä järjestelemällä työntekijäelementit osastojen sisälle relaatioita vastaavassa hierarkiassa. Kun XML-skeema on valmis, on sitä vastaavan XML-dokumentin luominen melko suoraviivaista. XML-skeema esitellään liitteessä 2 ja valmiista XML-dokumentista esitetään typistetty versio liitteessä 3.

4.3 SEML

Yksi esimerkki [Fong, 2006] määrittelevästä tulkkaavasta datan muuntimesta on [Fong and Shiu, 2012] esittelemä SEML (Structured Export Markup Language) -tulkki. Sen

tavoitteena on olla nopeasti omaksuttava ja helppokäyttöinen työkalu. Lisäksi sen on tarkoitus täyttää suurin osa käyttäjän mahdollisista tarpeista generoida dataa XML-muotoon.

SEML-tulkin käyttämä kieli on XML-pohjainen. Sitä käyttäessä luodaan siis XML-dokumentti, joka perusteella generoidaan SQL-kysely. XML valittiin kieleksi, niin että käyttäjien ei tarvitse opetella uutta ohjelmointikieltä SEML-tulkin käyttämistä varten. Käyttäjän, joka jo hallitsee XML-merkintäkielen, olisi siis mahdollista määritellä tarvitsemansa SEML-dokumentti XML-dokumenttien generointia varten. SEML-tulkille syötettävä XML-dokumentti sisältää kaiken tiedon, minkä se tarvitsee generoidakseen SQL-kyselyn, joka hakee relaatiotietokannasta toivotun informaation. Dokumentin generointi tapahtuu siis kolmessa vaiheessa. Ensin käyttäjä luo SEML-tulkille XML-dokumentin, jossa hän määrittelee, minkälaisen XML-dokumentin hän haluaa generoida. Seuraavaksi SEML tulkki käyttää saamaansa XML-dokumenttien SQL kyselyn generointiin, ja noutaa sillä halutun informaation relaatiotietokannasta. Kolmannessa vaiheessa tulkki generoi käyttäjän toivoman XML-dokumentin relaatiotietokannasta saadun informaation perusteella.

Tulkin tarvitseman SEML-dokumentin luomiseen on kaksi lähestymistapaa. Jos generoitavalle XML-dokumentille on jo ennalta määritelty malli, esimerkiksi XSD-(XML Schema Definition) tai DTD-formaatissa, on käyttäjän mahdollista käyttää sitä SEML-dokumentin luonnissa. SEML-tulkki osaa mallin perusteella generoida SEML-dokumentin, johon käyttäjän täytyy enää lisätä puuttuvat linkitykset relaatiotietokannan tauluihin ja sarakkeisiin. Toinen vaihtoehto on aloittaa määrittelemällä tulkilta toivottu XML-tiedosto, ja lisätä sinne linkitykset relaatiotietokannan tauluihin ja sarakkeisiin. Näin XML-dokumentista saa valmiin SEML-tulkkia varten, vaikka käytössä ei olisi valmista mallia XSD- tai DTD-formaatissa.

5 XML-muotoisen tesauruksen toteutuksen esittely

Osana tutkielmaa toteutettiin työkalu, joka johtaa tietonsa [Norri et al., 2020] esittelemän sanakirjan relaatiotietokannasta. Sen pääasiallinen toiminto on generoida XML-muotoinen tesaurussanakirja. Generoinnissa hyödynnetään tietokantaan tarkasti suunniteltua rakennetta ja elementtien puumaista rakennetta, joka mahdollistaa datan noutamisen rekursiivisesti. Toteutus tapahtui kahdessa erillisessä vaiheessa. Ensin toteutettiin naiiviratkaisu, joka todisti, että tarvittavat tiedot on mahdollista noutaa tietokannasta, ja että niiden perusteella saadaan generoitua toivottu XML-dokumentti. Ensimmäinen ratkaisu oli toteutukseltaan yksinkertainen ja se näkyi suorituskyvyssä. Sen suorituskyky on erittäin heikkoa verrattuna toiseen toteutukseen, joka toteutettiin, kun ensimmäinen versio oli todistanut, että suorituskykyisemmän version toteuttamiseen kannattaa käyttää aikaa. Ensimmäisen toteutuksen lähdekoodi on kuitenkin selkeä tapa käydä läpi molempien toteutuksien ytimessä oleva logiikka. Tutkimukseen liittävä lähdekoodi on julkaistu MIT-lisenssillä GitLabissa [Ala-Fossi, 2021].

Molemmat toteutukset siis perustuvat samaan ajatukseen, mutta ongelman ratkaisun lähestymistapa on erilainen. Kahden toteutuksen olemassaolo mahdollistaa myös vertailun. Lähinnä tuloksien vertailun toisiinsa, koska suorituskyvyn vertailusta ei voi juuri vetää muita johtopäätöksiä kuin sen, että ensimmäinen toteutus ei ollut optimaalinen. Kahden toteutuksen tuottamien XML-dokumenttien vertailu oli myös hyödyllistä, koska se mahdollisti nopean tavan tarkistaa, onko XML-dokumentti generoitu oikein. Jos generoidaan kokonainen termi-tesaurus XML-dokumenttina, se on noin 250 000 riviä pitkä. Eli käytännössä sen tarkistaminen edes kehittyneillä työkaluilla on hankalaa, puhumattakaan manuaalisesta tarkistamisesta. Koska tiedostoja oli useampi, niitä oli mahdollista vertailla. Suora vertailu esimerkiksi Unix-tyyppisien tietokoneiden diff-komentorivityökalulla ei ollut kuitenkaan mahdollista, koska toteutukset generoivat XML-tiedoston, jossa sanakirjan elementit olivat eri järjestyksessä. Tiedostojen pituus oli selkein keino nähdä suoraan, voivatko tiedostot sisältää saman sanakirjan.

Molemmat toteutukset käyttävät samaa alustaa ja samoja työkaluja. Ne myös jakavat osan koodista, kuten esimerkiksi varsinainen XML-dokumentin generoiva sovelluskoodi Python-natiiveista tyypeistä on jaettu molempien toteutuksien kesken. Lopullinen työkalu sisältää kummankin toteutuksen ja ne ovat käytettävissä rinnakkain. Tämä on mahdollista, koska ne tuottavat relaatiotietokannasta saman muotoista dataa, josta XML-dokumentti generoidaan. Koska sovellus on toteutettu palvelimena, voi käyttäjä kokeilla eri toteutuksia tekemällä kutsut eri osoitteisiin.

5.1 Alkuperäinen sanakirja

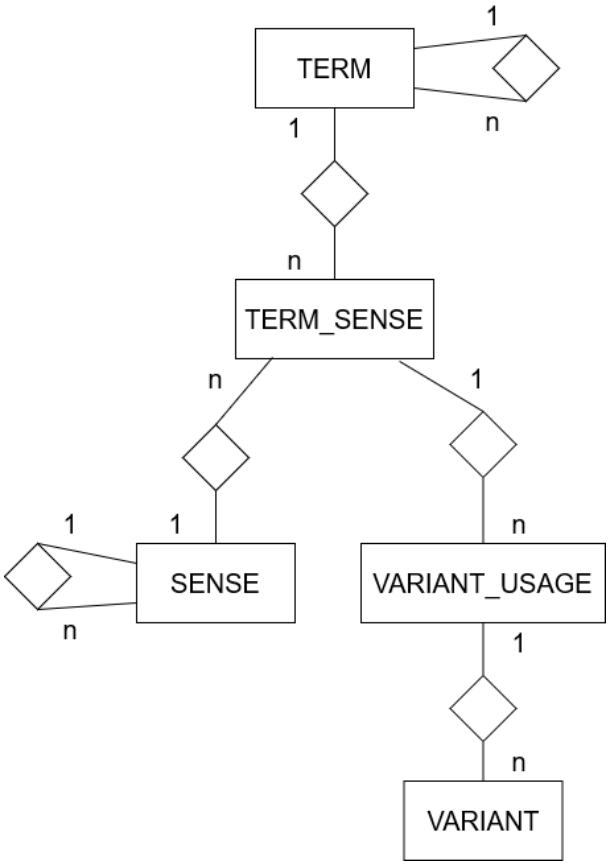
Tässä työssä toteutettu sovellus rakennettiin jo olemassa olevan relaatiotietokannan päälle. Tämä tietokanta on rakennettu säilyttämään englanninkielinen lääketieteellinen sanakirja vuosilta 1375–1550 ”Dictionary of medical vocabulary in English, 1375–1550”.

Tietokanta rakennettiin sanakirjan painetun version pohjaksi, mutta se sisältää myös tietoa, jota ei yleensä eksplisiittisesti sisällytetä sanakirjoihin. Tämä johtuu siitä, että tämä relaatiotietokanta rakennettiin samalla käytettäväksi työkaluna tutkimuksessa. [Norri et al., 2020]

Dictionary of medical vocabulary keskittyy siis keskiaikaiseen englanninkieliseen lääketieteelliseen sanastoon. Tämä tarkoittaa, että se sisältää keskiajalla käytettyjä sanoja, jotka kuvaavat eri sairauksia, ruumiinosia ja erilaisia lääketieteellisiä operaatioita sekä työkaluja. Tämän sanakirjan rakentamiselle oli useampia eri syitä. Ensinnäkin sanakirja kohdistuu aikaan, jolloin englannin kieltä alettiin käyttämään lääketieteellisessä kirjallisuudessa. Tätä ennen englannin sijasta käytetiin latinaa ja ranskaa. Toinen syy on se, että aikaisemmat vastaavan kaltaiset teokset keskittyivät yksinomaan painettuun kirjallisuuteen, vaikka painokoneet olivat keskiajalla vielä melko harvinaisia. Nyt sanakirja rakennettiin yhteensä yli 11 000 sivusta painetusta ja käsin kirjoitetusta tekstistä. Kohdeyleisönä ovat kaikki, jotka ovat kiinnostuneet keskiaikaisesta lääketieteestä, sekä tietysti sanastotieteilijät ja lääketieteen historian tutkijat. [Norri et al., 2020]

5.2 Tietokantataulut ja taulujen mallit

Sovellus toteutettiin jo käytössä olevan sanakirjan relaatiotietokannan päälle, joten relaatiotietokannan muokkaaminen käyttötarkoituksen mukaiseksi ei ollut enää mahdollista. Sitä ei kuitenkaan myöskään nähty tarpeelliseksi, koska relaatiotietokannalla oli jo valmiiksi selkeä rakenne, joka sopi ongelmitta toteutetun sovelluksen käyttötarkoitukseen. Tarkastelemme tarkemmin termeille generoitavaa tesaurusta, joten siihen liittyvät taulut on esitetty kuvassa 7. Myös merkityksille toteutettiin tesauruksen generointi. Siinä käytettävät tietokantataulut ovat vastaavan kaltaiset.



Kuva 7. Termeihin liittyvät taulut ER-mallina.

Termi-tesauruksen generoinnin kannalta tärkeät tietokantataulut alkavat TERM-taulusta. Tämä on tietokantataulu, joka nimensä mukaisesti sisältää termien tiedot. Tesauruksen kannalta tärkein tieto, jonka taulu sisältää on termin vanhempitermin tietokantatunniste. Tämän tietokantatunnisteen perusteella pystymme rekursiivisesti hakemaan kaikki tarvitsemamme termit. Kuvassa seuraavana on TERM_SENSE-taulu, joka on tietokantataulu, joka sisältää tiedon kaikista toisiinsa liittyvistä termi ja merkitys pareista. TERM_SENSE-taulun tietojen perusteella pystymme hakemaan jokaiselle termille niihin liittyvät merkitykset SENSE-taulusta. VARIANT_USAGE-taulu taas sisältää tiedon varianttien käytöstä. Nämä varianttien käytöt sisältävät toteutuksen kannalta tärkeät VARIANT-taulun tietokantatunniste ja TERM_SENSE-taulun tietokantatunniste parit. Eli kun tiedämme termin tietokantatunnisteen voimme TERM_SENSE- ja VARIANT_USAGE-taulujen kautta hakea kaikki termiin liittyvät variantit VARIANT-taulusta. TERM-taulussa käytetään siis vieruslistaa, mutta sen lisäksi tässä relaatiotietokannassa käytetään myös TERM_SENSE-taulua sulkeumataulukkona, joka säilyttää osan hierarkiasta.

Toteutuksessa käytetään SQLAlchemy ORM:ia Python-ohjelmointikielelle. SQLAlchemy koostuu kahdesta osasta, ORM:ista ja Core:sta. ORM (Object Relational Mapper) eli objektien relaatioiden kartoittaja on työkalu, jota käytetään

epäyhteensopivien järjestelmien yhdistämiseen. Core on SQL-abstraktiotyökalu, joka mahdollistaa SQL-kyselyjen generoinnin Python-ohjelmointikielellä kirjoitettujen määritelmien perusteella [SQLAlchemy, 2021]. Tässä tapauksessa mallinnamme PostgreSQL-tietokannan taulut SQLAlchemyn ymmärtämään Python-natiiviin muotoon. Tämä mahdollistaa sen, että voimme käsitellä tietokannan tauluja suoraan Python-ohjelmointikielellä määriteltyjen luokkien kautta. SQLAlchemy mahdollistaa näitä luokkia käyttäen SQL-kyselyjen generoinnin. Voimme siis määritellä, mitä haluamme hakea ja mistä tietokantataulusta, minkä jälkeen SQLAlchemy hoitaa generoi kyselyn ja ajaa sen. Kyselyn vastauksen saamme suoraan käytettäväksi Pythonin-natiivissa muodossa.

Tietokannan taulujen mallintamiseen SQLAlchemyn ymmärtämään muotoon käytettiin sqlacodegen-nimistä työkalua. Sqlcodegen lukee sinne syötteenä annetun tietokannan rakenteen, ja generoi sen perusteella tarvitsemamme mallin koodin. Kun tietokannan malli on generoitu sqlcodegenillä, pystymme käyttämään sitä yhdessä SQLAlchemyn kanssa SQL-kyselyjen generoimiseen [sqlcodegen, 2021]. Tietokannan mallin generointi vie toteutusta lähemmäs Fongin [2006] määrittelemää neljättä menetelmää, eli loogisen tason muuntajaa, koska sqlcodegen ymmärtää syötteenä annettavaa relaatiotietokantaa ja generoi sen perusteella Python-koodia. Tämän jälkeen mallia täytyi vielä muokata, koska generoitu malli ei sisältänyt kaikkia mahdollisia mallin osia. Manuaalisesti lisättäväksi jäi osa taulujen välisistä suhteista. Esimerkiksi SENSE_TERM-taululle sqlcodegen generoi malliin oletuksena suhteen TERM-tauluun. Tätä suhdetta käyttäen pystymme Python-koodissa kutsumaan SENSE_TERM-mallin kautta siihen liittyvää termiä. Vastaavaa suhdetta ei kuitenkaan generoida toiseen suuntaan, eli TERM-malliin täytyi manuaalisesti lisätä suhde SENSE_TERM-malliin. Kun suhde on lisätty TERM-mallin kautta, voidaan pyytää lista kaikista termiin liittyvistä SENSE_TERM-objekteista. Tietokantataulujen välisien suhteet täytyy määritellä, että SQLAlchemystä saa täyden hyödyn, kun tietoa noudetaan tietokannasta.

5.3 Tiedon noutamisen logiikka

Määritellään seuraavaksi korkealla tasolla toteutettu logiikka, joka noutaa tesauruksen sanakirjan tietokannasta. Tekstimuotoisen selityksen lisäksi logiikka esitellään pseudokoodina, joka perustuu alkuperäisen toteutuksen sovelluskoodiin. Esimerkissä rakennetaan XML-muotoinen tesaurus termeille.

Tietokannan termi tietokantataulu sisältää useita termejä, joilla ei ole vanhempaa termiä. Tämä tarkoittaa sitä, että tesaurus muodostuu useammasta puusta. Näiden puiden juuritermeillä ei ole yhteistä vanhempaa termiä tai juurta. Sovellus aloittaa siis etsimällä termi tietokantataulusta kyselyn vanhemmaksi määritellyn termin tai kaikki termit, joilla ei ole vanhempaa, jos käyttäjä haluaa generoida kerralla koko tesauruksen. Tämän jälkeen termit käydään yksitellen läpi ja niiden perusteella tehdään rekursiivisesti ketju kyselyjä,

jotka hakevat kaikki lapset, lapsenlapset ja niin edelleen. Myös termeihin liittyvät merkitykset ja merkityksien variantit noudetaan tietokannasta. Vastaukset kerätään listaan, joka lähetetään eteenpäin muunnettavaksi XML-muotoon.

Varsinainen rekursio alkaa, kun toteutus alkaa käymään läpi ensin noudettujen termien lapsia. Koska tietokantataulut ovat mallinnettu Python-ohjelmointikielen luokkina joita SQLAlchemy ymmärtää, pystymme käsittelemään termi-tietokantataulua kuin mitä tahansa muuta oliota ohjelmointikielessämme. SQLAlchemy generoi lennossa tarvittavat kutsut PostgreSQL-tietokantaan. Käymme ensin läpi kaikki termin merkitykset, ja merkityksille käymme läpi kaikki variantit. Tämän jälkeen funktio kutsuu itseään termin lapsille ja lopulta palauttaa termiin liittyvät tiedot ja listan lapsistaan niihin liittyvistä tiedoista.

Generoitu tietorakenne on vastaavan muotoinen.

```
Lista merkityksiä:
    Merkitys:
        Tunniste, Merkitys-teksti, Merkityksen selite
        Listat merkityksille ja varianteille
        Lista lapsimerkityksistä:
            Lapsimerkitys
                Tunniste, Merkitys-teksti, Merkityksen selite
                Listat merkityksille ja varianteille
                    Lista lapsimerkityksistä:
                        Ja niin edelleen...
```

Tämä versio sovelluksesta oli hyvin yksinkertaista toteuttaa ja se tuotti nopeasti laadukkaita tuloksia. Sen ongelma on kuitenkin se, että se ei hyödynnä SQLAlchemyn tai PostgreSQL:n monimutkaisempia ominaisuuksia, ja on sen takia hyvin hidas. Toteutus laukaisee tuhansia yksinkertaisia SQL-kyselyjä, joten pelkkään sovelluksen ja tietokannan väliseen kommunikaatioon kuluu huomattavasti aikaa. Toteutuksella on kuitenkin oma arvonsa puhtaasti loogisessa mielessä. Koodi on erittäin tiivistä ja helposti ymmärrettävää. Teoriassa se voisi toimia nopeastikin, jos käytössä olevat työkalut olisivat erilaisia. Tuhansien SQL-kyselyjen laukaiseminen tähän tyyliin tulee aina olemaan hidasta verrattuna muihin ratkaisuihin. Korkeampi suorituskyky olisi mahdollista saavuttaa esimerkiksi rinnakkaisilla kyselyillä, mutta siinäkin olisi rajansa, koska seuraava kysely tehdään usein edellisen kyselyn vastauksen perusteella. Toteutuksen pseudokoodi on luettavissa liitteessä 4.

5.4 Tehokkaamman version toteuttaminen

Vaikka ensimmäinen toteutus olikin toimiva, se ei hyödyntänyt PostgreSQL-tietokannan monimutkaisempia ominaisuuksia, jotka nopeuttaisivat vastaavan XML-tiedoston generointia huomattavasti. Kuten Fong ja Shiu [2012] huomioivat, jos sovellus, joka rakentaa XML-dokumenttia relaatiotietokannan perusteella tekee SQL-kyselyjä erikseen ensin vanhempielementtien ja sitten lapsielementtien sarakkeille tietokantatauluissa, on todennäköistä, että sovelluksen käyttäjä kohtaa ongelmia liian hitaan suorituskyvyn kanssa. Varsinkin suurempien XML-dokumenttien noutaminen relaatiotietokannasta on erittäin hidasta, jos se toteutetaan muodostamalla suuri määrä SQL-kyselyjä yksittäistä elementtiä kohden.

PostgreSQL on tukenut rekursiivisia kyselyjä jo vuodesta 2009 (versio 8.4.22), joten sen käyttäminen vaikuttaa selkeältä keinolta toteuttaa tesauruksen muodostaminen nopeammin kuin alkuperäisessä toteutuksessa. Tämän takia toinen versio kehitettiin juuri rekursiiviset kyselyt mielessä. Se tekee yhden ison rekursiivisen SQL-kyselyn ja sen jälkeen käsittelee vastauksen samaan muotoon, minkä alkuperäinenkin toteutus tuotti. Uuden toteutuksen nopeus on aivan eri luokassa kuin edellisen. Kokonaisen termi-tesauruksen generointi XML-dokumentiksi kesti vanhalla versiolla noin kolme minuuttia. Sovelluksen uudella versiolla saman tiedoston generointiaika saatiin noin kuuteen sekuntiin.

Rekursiivisen SQL kyselyn rakentaminen SQLAlchemyllä aloitetaan määrittelemällä kysely, joka noutaa joko kyselyn vanhemmaksi määritellyn termin tai kaikki termit, joilla ei ole vanhempaa. Tämä kysely noutaa jokaiselle juuritermille kaikki siihen liittyvät tiedot, jotka tarvitaan tesaurukseen. Kysely yhdistää join-operaatiolla termi, merkitys- ja variantti-taulut. Kyselyssä on mukana myös merkityksen ja variantin noutamiseen käytetyt välitaulut, jotka sisältävät termin relaatiot merkityksiin ja variantteihin. Nämä taulut on jätetty pois pseudokoodista, koska ne eivät ole tarpeellisia logiikan selittämiseen. Nämä termien relaatioita sisältävät tietokantataulut ovat kuitenkin mukana varsinaisessa esiteltävässä SQL-kyselyssä. Tämän jälkeen kysely vanhemmille on periaatteessa määritelty, mutta kysely asetetaankin rekursiiviseksi väliaikaiseksi CTE-tietokantatauluksi (Common Table Expressions). Juuressa olevat termit siis haetaan väliaikaiseen tietokantatauluun, jonka perusteella pystymme tekemään lisää kyselyjä. Väliaikaiset tietokantataulut eivät tue kaikkia tavallisien tietokantataulujen ominaisuuksia ja ne katoavat, kun ne luonut käyttäjä sulkee yhteyden tietokantaan [Ben-Gan and Moreau, 2008].

Lapsitermeille tehdään toinen kysely, joka on hyvin pitkälti samanlainen. Vanhempitermien kysely kuitenkin liitetään join-operaatiolla tähän kyselyyn. Tämä mahdollistaa sen, että kyselyssä voidaan valita sellaisia lapsia, joiden vanhempitermi on joku aikeisemmin luodussa väliaikaisessa CTE-tietokantataulussa olevista vanhemmista.

Kysely siis tässä vaiheessa noutaa kaikki juuritermien lapset, mutta tämä ei vielä riitä lapsenlapsien noutamiseen tai sitä syvempien relaatioiden selvittämiseen. Tämä ratkeaa sillä, että lopuksi nämä kaksi luotua kyselyä liitetään toisiinsa myös union-operaatiolla. Se yhdistää kyselyjen tulokset, ja ratkaisee lapsenlapsien löytämisen, koska lapsikyselyn omia tuloksia voidaan käyttää uusien lapsien etsimiseen. Kyselyn toteuttaminen SQLAlchemyllä esitellään liitteessä 5.

Kyselyjen yhdistämisen jälkeen toteutus onnistuneesti noutaa tietokannasta kaiken datan, minkä tarvitsemme määrittelemämme XML-tesauruksen muodostamiseen. Yhdistettyjen kyselyjen tuottama vastaus on kuitenkin vielä täysin väärässä muodossa XML-generoinnin kannalta, koska kaikki data on yhdessä litteässä vastauksessa. Tästä muodostuukin rekursiivisen kyselyn pullonkaula, jos haluamme generoida kokonaisen termi-tesauruksen. SQL-lauseet tuottavat aina litteitä vastauksia ja eivät sen takia ole optimaalisia hierarkkisen datan noutamiseen. Vastaus sisältää hierarkian, mutta datan tietorakenne täytyy muuttaa hierarkkiseksi, että se voidaan esittää hierarkkisessa formaatissa. Jos haluamme hakea kaikki termit, voisimme yksinkertaisesti vain hakea kaikki termit ilman rekursiota ja saisimme samankaltaisen vastauksen. Kuitenkin jos haluamme generoida XML-muotoisen termi-tesauruksen käyttäen juurena tiettyä termiä, tämä ratkaisu voi olla nopeampi kuin kaiken noutaminen ilman rekursiota. Jos osittaisen sanakirjan generointi toteutettaisiin noutamalla relaatiotietokannasta kaikki tieto joistain tietokantatauluista, ei relaatiotietokannan käyttäminen datan säilyttämiseen olisi enää perusteltua, koska sen ominaisuuksia ei hyödynnettäisi.

Litteä vastaus täytyy siis vielä prosessoida hierarkkiseen tietorakenteeseen. Tämä tehdään useammassa eri vaiheessa, joista ensimmäisessä suodatetaan kyselyn vastauksesta pois duplikaattilapset ja lapsien duplikaattimerkitykset ja -variantit. Duplikaatit johtuvat juuri vastauksen litteästä rakenteesta, jossa saman lapsen alle tulevia useita merkityksiä ja variantteja ei voi esittää millään muulla tavalla. Kuten relaatiotietokannassa, SQL-kyselyn tuottamassa vastauksessa on rivi jokaista relaatiota kohden. Suodatuksen jälkeen meillä on lista lapsista, niiden merkityksistä ja varianteista. Tämän jälkeen yhdistämme lapset listoihin niiden vanhemman perusteella. Käymme siis läpi kaikki lapset ja sijoitamme jokaisen listaan, joka käyttää listassa olevien lapsien vanhemman tunnistetta. Listat tallennetaan Python-ohjelmointikielien sanakirjaan, joka on tarkoitettu avain- ja arvoparien tallentamiseen. Tämä sanakirja sisältää siis kaikki tesauruksen vanhempia vastaavat avaimet. Näiden avaimien perusteella voimme toteuttaa viimeisen vaiheen, jossa käymme läpi kaikki sanakirjan sisältämät termit, ja tarkistamme jokaiselta, onko kyseisen termin tietokantatunniste avaimena sanakirjassa. Jos löydämme termin, joka on sanakirjassa avain, tiedämme että sillä on lapsi termejä ja sijoitamme sen sisälle viittauksen listaan, joka sisältää sen lapset. Näin pystymme rakentamaan hierarkkisen tietorakenteen rekursiivisen SQL-kyselyn tuottaman litteän vastauksen

perusteella. Ratkaisu toimii ilman, että meidän täytyy järjestelmällisesti kulkea rakentuvaa puuta pitkin, etsien lapsien vanhempia eri oksista.

Yhden rekursiivisen kyselyn käyttäminen on tehokas tapa noutaa hierarkkista dataa relaatiotietokannasta, mutta sitä käyttäessä tulee ottaa huomioon, mitä haetaan ja missä muodossa vastaus on. Jos haluamme generoida kokonaisen termi-tesauruksen, haluamme noutaa kaikki termit. Tässä tilanteessa yksinkertaisempaa on tehdä normaali kysely kaikille termeille, ilman rekursiota. Kyselyn tuottama vastaus on joka tapauksessa saman muotoinen, joten vastaus joudutaan käsittelemään samalla tavalla. Rekursio tietokannan puolella on tällöin täysin turhaa. Käytännössä suorituskyvyssä ei välttämättä ole suurta eroa, mutta kyselyn toteuttaminen on kehittäjälle huomattavasti yksinkertaisempaa, jos kyselyyn ei tarvitse sisällyttää rekursiota. Rekursiivinen kysely on kuitenkin perusteltu silloin, kun tietokannasta ei haluta hakea aivan kaikkea dataa kerralla, koska se säästää käytettäviä resursseja tiedonsiirrossa ja potentiaalisesti datan suodattamisessa. Todelliset erot suorituskyvyssä riippuvat datan määrästä ja työkaluista.

5.5 Generoitujen SQL-kyselyjen esittely

Alkuperäinen ratkaisu perustui usean SQL-kyselyn lähettämiseen. Jokaista termiä varten tehdään useita eri kyselyä. Ajettavat kyselyt ovat: kysely termille itselleen, kysely, joka noutaa termin merkityksien tietokantatunnisteet, toinen kysely, joka noutaa termin varianttien tietokantatunnisteet, erikseen kyselyt jokaiselle merkitykselle ja variantille ja lopuksi kysely, joka hakee kaikki termin lapsitermit. Menetelmän hitaus suuremmilla termimäärillä johtuu juuri kyselyjen korkeasta määrästä, koska jokainen kysely aiheuttaa sovelluksen ja PostgreSQL-tietokannan välistä kommunikaatiota. Näitä kyselyjä ei myöskään tehdä rinnakkain, eli toteutus kuluttaa merkittävän määrän aikaa kyselyjen vastauksia odotellessa. Heikkouksistaan huolimatta menetelmä voi toimia tyydyttävällä nopeudella, jos noudettavien termien määrä on matala.

Ratkaisu ensimmäisen ratkaisun heikkoon suorituskykyyn on monimutkaisemman kyselyn käyttäminen. Yksi rekursiivinen kysely voi noutaa kerralla kaiken, mitä useat yksinkertaiset kyselyt noutaisivat. Hyvänä puolena ratkaisussa on se, että se tiputtaa sovelluksen kanssa kommunikointiin kestävää aikaa merkittävästi, koska kyselyjä on vain yksi. Menetelmä myös siirtää merkittävän osan sovelluksen logiikasta suoritettavaksi PostgreSQL-tietokannassa, jolloin saamme täyden hyödyn tietokannan tehoista. Generoitu SQL-lause on liitteessä 6. Kyseinen versio SQL-lauseesta on hieman siistitty esittelemistä varten, ja ei siksi toimisi suoraan.

Generoitu SQL-lause alkaa CTE-taulun määrittelemisellä. Tähän väliaikaiseen tietokantatauluun lapsitermit rekursiivisesti kerätään. Kyselyn tärkein osa on jaettuna kahteen erilliseen select-lauseeseen, joista ensimmäinen hakee alustavan listan termejä. Nämä termit valitaan niiden vanhemman tietokantatunnisteen perusteella, eli käyttäjä voisi antaa jonkun termin tietokantatunnisteen, jonka kaikki lapset haluavat, tai

esimerkiksi hakea termejä, joilla ei ole vanhempaa termiä. Ilman vanhempaa termiä haettaessa kysely hakee kaikki termit. Toinen select-lause taas hakee lapsiatermeille, jotka ovat jo väliaikaisessa CTE-taulussa. Koska select-lauseet yhdistetään unionilla, siirretään nämä uudet lapset myös CTE-taluun, jolloin myös niiden lapset etsitään. Molemmat select-lauseet sisältävät useita join-operaatioita, joiden avulla tämä SQL-kysely pystyy noutamaan kaikki lopulliseen tesaurukseen tarvittavat tiedot. Viimeiseksi lopulliseen vastaukseen CTE-taulusta valitaan kaikki sinne lisätyt lapset.

5.6 XML-dokumentin generointi

Molemmat tesauruksen noutomenetelmistä tuottavat samanlaisen tietorakenteen, jonka perusteella XML-dokumentti generoidaan. XML-generointivaihe on siis samanlainen esimerkki sovelluksessa molempien tiedonhaku menetelmien kanssa. XML-muotoisen tesauruksen generointiin Python-natiiveista tyypeistä käytetään lxml-nimistä kirjastoa. Tämän kirjaston avulla pystymme määrittelemään XML-muodon, ja sitten syöttämään sille sopivaa dataa. Lopputuloksena on validi XML-dokumentti, määrittelemässämme rakenteessa. Lxml mahdollistaa nopeiden XML-kirjastojen yksinkertaisen käyttämisen Python-ohjelmointikielellä [lxml, 2021].

Käytännössä tesauruksen elementtien tiedot täytyy vain siirtää vastaavien lxml-elementtien sisälle, koska tietorakenne on jo XML-dokumenttia vastaava. Tässä vaiheessa myös varmistetaan, että dokumentissa olevilla termeillä, merkityksillä ja varianteilla on kaikilla uniikit tunnisteet. Tietokannassa riittää, että niiden tunniste on uniikki oman taulunsa sisällä, mutta XML-dokumenttia varten niiden tunnisteisiin lisätään tieto siitä, minkä tyyppinen elementti on kyseessä. Liitteessä 7 on esimerkki sovelluksen generoimasta XML-dokumentista.

Pseudokoodissa esitellään aikaisemmin generoidun tietorakenteen muuntaminen XML-muotoon. Kuten varsinaisessa koodissa, etree-moduulin kautta kutsutaan lxml kirjaston osia. Heti ensimmäisenä etree-moduulia käytetään XML-elementin luomiseen. Tämä elementti on koko XML-muotoisen tesauruksen juurielementti. Tämän jälkeen create_term_thesaurus-funktiossa käydään läpi kaikki tietokannasta noudetut termit, ja niillä kutsutaan term_maker-funktiota. Funktio loppuu siihen, kun thesaurus-elementti muunnetaan tekstimuotoiseksi, varsinaiseksi XML-dokumentiksi. Pseudokoodin toisessa osassa, term_maker-funktiossa, vastaan otetaan create_term_thesaurus_xml-funktiossa läpi käydyt termit. Kaikille lapsitermin elementit luodaan, merkitys- ja variantti-listat käydään läpi luoden niille elementit ja lopuksi termin kutsutaan samaa term_maker-funktiota termin omille lapsille. Pseudokoodi on nähtävissä liitteessä 8.

6 HTML-muotoisen tesauruksen toteutuksen esittely

XML-dokumentin lisäksi sovellukseen toteutettiin myös verkkosivuversio tesauruksesta. Tämä versio on myös tarjolla termeille ja merkityksille. Verkkosivu toteutetaan samaan tyyliin kuin XML-dokumentti, mutta lopputuloksena on HTML-dokumentti, joka verkkoselaimet osaavat esittää käyttäjälle. Kuten XML, myös HTML on merkintäkieli, jolla voimme esittää hierarkkista dataa, kuten tesauruksen. Suurimpana etuna HTML-versiossa on se, että toisin kuin XML-versio, se toimii käyttöliittymänä tesaurukseen. Hyperlinkkien ansiosta käyttäjä voi vaivattomasti selata tesaurusta, ja jopa siirtyä merkitys-tesauruksesta termi-tesaurukseen ja toisin päin. Jokaisella tesauruksen sivulla on linkit kaikkiin sivulla tarkasteltavaan termiin tai merkitykseen liittyviin kohteisiin. Termin sivulla on linkit termin ylätermeihin, alatermeihin ja termiin liittyviin merkityksiin. Merkityksien sivuilla on vastaavat linkin merkityksille.

6.1 Sivukohtaisen datan noutaminen

Koska verkkosivua rakentaessa tietokannasta halutaan noutaa vain hyvin tarkasti rajattu tesaurukseen liittyvä tieto, täytyi sen noutamisen menetelmää muokata. Esimerkiksi termin XML-dokumenttia rakentaessa haettiin kaikki määritellyn termin lapsitermit ja niihin liittyvät tiedot, verkkosivulla taas halutaan esittää vain rajallinen määrä tietoa. Tämän takia verkkosivujen tietoja noutaessa haettaville lapsitermeille on määritelty maksimi syvyys. Verkkosivuilla lapsitermit myös sivutetaan. Tästä seuraa se, että vain verkkosivulle sopiva määrä lapsitermejä noudetaan kerralla. Riippuen termistä, tämä voi olla vain murto-osa kaikista termeistä, jotka olisi voitu kerralla noutaa. Näin saavutetaan sekä käyttäjälle selkeämmät verkkosivut, että nopeammat verkkosivujen latausajat.

6.2 HTML-dokumentin generointi

Käyttäjälle esitettävä HTML-muotoinen tesaurus generoidaan tietokannasta noudetun tiedon perusteella käyttäen Tornado Web Frameworkin verkkosivupohjia. Se on Python-pohjainen työkalu, joka on tarkoitettu verkkosovelluksien kehittämiseen. Hyödynnämme Tornadon HTTP-palvelinta ja sen kutsujen hallinnoijaa [Tornado, 2021]. Niiden avulla pystymme määrittelemään palvelimen, joka vastaa kutsuihin ennalta määriteltyihin polkuihin. Toisin sanottuna, HTTP-palvelin palauttaa käyttäjälle joko XML- tai HTML-version tesauruksesta, riippuen siitä kuinka palvelimelle tehty kutsu on määritelty. Periaatteessa myös lxml olisi pystynyt generoimaan HTML-muotoisen tesauruksen.

Tornado-työkalu valittiin, koska sitä käytettiin jo valmiiksi sivujen tarjoilemiseen verkon yli, joten se oli luonteva valinta. Tämä ei myöskään lisännyt projektin riippuvuuksien määrää, koska sivun generointiin käytetyt työkalut olivat jo osa Tornadoa. Lisäksi sivujen renderöinti HTML-muotoon palvelimella on nopeaa ja se vapauttaa käyttäjän selaimen vain esittämään sivun sen sijaan, että sen tulisi myös generoida lopullinen sivu paikallisesti.

Koska yhdelle verkkosivulle haetaan kerralla vain rajallinen määrä tietoja, on myös HTML-dokumentin generointi nopeaa. Esimerkiksi termi-tesauruksessa ei haeta tai näytetä kaikkia termin lapsitermejä kerralla, koska niille on määritelty maksimimäärä, joka voidaan kerralla näyttää. Käyttäjälle tämä esitetään sivutuksena, jossa käyttäjä voi liikkua eri lapsitermi sivujen välillä.

Tornado Web Frameworkin verkkosivupohjat mahdollistavat melko yksinkertaisen verkkosivujen generoinnin, jos kehittäjällä on jo aikaisempaa kokemusta HTML-merkintäkielestä ja edes jostain yksinkertaisemmasta ohjelmointikielestä. Verkkosivupohjat toimivat siten, että kehittäjä kirjoittaa verkkosivun rungon HTML-merkintäkielellä, ja tämän jälkeen lisää HTML-tagien väliin verkkosivupohjalle spesifiä syntaksia. Tällä syntaksilla määritellään, missä on verkkosivugeneraattorin suorittama sovelluskoodi. Tämä sovelluskoodi on hyvin lähellä esimerkiksi Python-ohjelmointikieltä, ja kun verkkosivugeneraattori suorittaa sen, on sovelluskoodin tarkoitus korvata HTML-tiedostossa itsensä suorittamisen tuloksena muodostuneella uudella HTLM-pätkällä. Lopputulokseen vaikuttaa sovelluskoodin lisäksi tietysti myös sovelluskoodille annettu mahdollinen syöte. Sovelluskoodi määrittelee siis sivun rakenteen, kun taas syöte määrittelee sisällön. Jos syöte ei vaihtelisi, voisi sivun generoida staattisella sivugeneraattorilla, jonka tulos tallennettaisiin palvelimelle etukäteen. Jos taas syötettä ei annettaisi ollenkaan, olisi järkevintä vain kirjoittaa sivu kokonaisuudessaan suoraan HTML-merkintäkielellä. Tämän tesauruksen tapauksessa kumpikaan näistä tilanteista ei päde, joten järkevintä on generoida sivu lennossa syötteen perusteella.

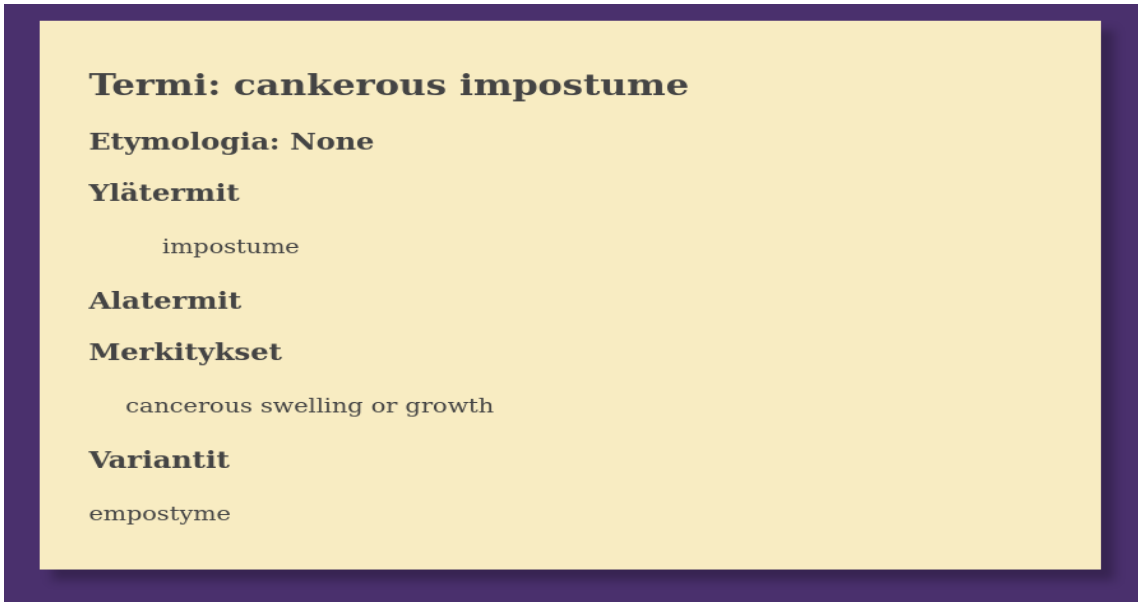
Koodiesimerkissä 1 näemme esimerkin verkkosivupohjasta, jossa generoidaan lista. Sovelluskoodia ympäröi HTML-tagit “
” ja “”, jotka määrittelevät listan alun ja lopun. Lista ei kuitenkaan sisällä varsinaisia listan elementtejä, jotka merkitään tageilla “
” ja “”. Lista sisältää kuitenkin sisältää HTML-merkintäkielen ulkopuolisilla “{%” ja “%}” tageilla merkittyjä osioita. Nämä osiot sisältävät verkkosivun generointia varten suoritettavan sovelluskoodin. Tämä esimerkkipätkä sisältää yksinkertaisen for-silmukan, joka lisää listaan linkkejä termien sivuille.

```
<ol>
  {% for parent in parents %}
    <li>
      <div>
        <a href={{ term_path + str(parent['identifier']) }}>
                {{ parent['term_text'] }}
        </a>
      </div>
    </li>
  {% end %}
</ol>
```

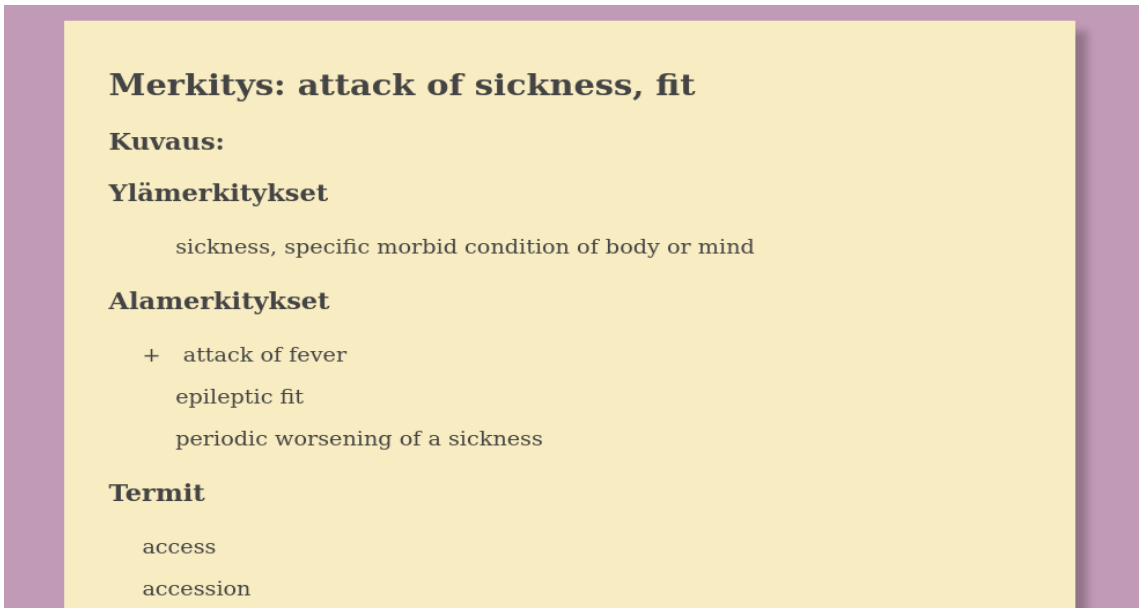
Koodiesimerkki 1. Listan generointi

6.3 Käyttöliittymän käyttäminen

Käyttöliittymä on jaettu kahteen eri osioon. Toista niistä käytetään termien selaamiseen ja toista merkityksien. Käyttäjän saapuessa jollekin sanakirjan sivuista, hän voi jatkaa sanakirjan selaamista muutamalla eri tavalla. Jos käytämme esimerkkinä kuvitteellista sanakirjan sivua, joka sisältää ainakin yhden kaiken tyyppisiä linkkejä, voi käyttäjä tehdä useita eri päätöksiä. Käyttäjä voi esimerkiksi vaihtaa näkymänsä sanakirjassa termistä merkitykseen ja toisin päin. Kuten näemme kuvassa 8 termien ja merkityksien välillä voi olla linkki, joka näytetään käyttäjälle käyttöliittymässä. Tätä linkkiä klikkaamalla käyttäjä pystyy siirtymään merkityksen sivulle, jossa on vastaava linkki takaisin termin sivulle. Kuvassa 9 näemme esimerkin merkitys-tesauruksen käyttöliittymästä, jossa on linkki takaisin merkitykseen liittyvien termien sivuille. Työn pohjana toimivan tietokannan rakentamisen yhteydessä sille luotiin myös käyttöliittymä. Myös tämä käyttöliittymä toteutettiin verkkosivuna, joten nämä kaksi eri käyttöliittymää on teoriassa mahdollista liittää toisiinsa. Käyttöliittymien yhdistäminen rajattiin kuitenkin tämän työn ulkopuolelle. Tämä jatkokehitysidea on kuitenkin mahdollista toteuttaa HTML-merkintäkielen hypertekstuaalisuuden ansiosta.



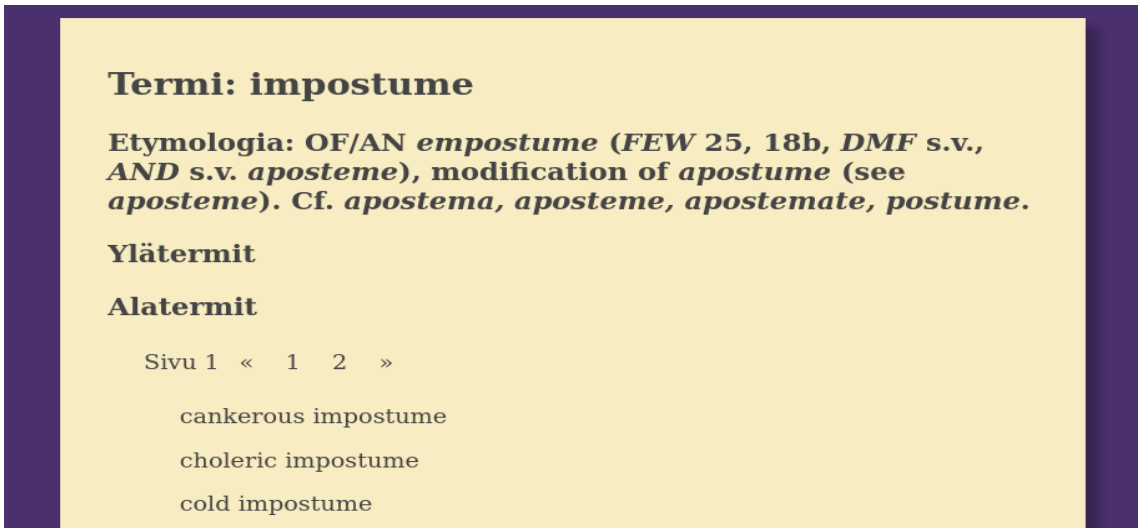
Kuva 8. Termi, jolla ei ole lapsia. Kuvassa näkyvät myös termin merkitykset ja variantit.



Kuva 9. Merkitys-tesauruksen käyttöliittymä.

Termin sivulla on siihen liittyvien merkityksien lisäksi listattu kaikki sen ylätermit ja alatermit. Nämä termit on listattu aakkosjärjestykseen ja sivutettu, että niiden selaaminen olisi käyttäjälle mahdollisimman vaivatonta. Mikäli myös termin alatermillä on alatermejä, on alatermin eteen lisätty +-merkki. Klikkaamalla tämän merkityn alatermin linkkiä, käyttäjä pääsee tarkastelemaan myös näytä syvemmällä sijaitsevia alatermejä. Tämä lista on nähtävissä kuvassa 10, missä näemme juuritermin sivun. Merkityksen sivulla on vastaavat ominaisuudet alamerkityksille. Tämän lisäksi termien sivuilla on listattuna myös termin etymologia ja merkityksille on listattu niiden kuvaukset. Termi- ja merkityssivujen välille on pyritty luomaan selkeä ero sivujen värivalinnoilla, koska

sivujen rakenne ja sisältö muistuttavat muuten visuaalisesti toisiaan. Näin käyttäjä pystyy tunnistamaan mistä sivusta on kyse, ilman että hänen on pakko lukea mitä sivulla lukee. Termien sivuilla on myös listattuna termin eri variantit, jotka tesaurus sisältää. Näille varianteille ei ole omia sivuja, joten ne ovat yksinkertaisesti osana termin oletusmuodon sivua.



Kuva 10. Termin sivu ja lista sen lapsista.

7 Yhteenveto

Toteutettu työ tukee [Norri et al., 2020] näkemystä siitä, että sanakirjojen tallentaminen relaatiotietokantaan on toimiva ratkaisu, toisin kuin historiallisesti on ajateltu. Hierarkkista dataa voi generoida relaatiotietokannasta melko nopeasti, hyödyntäen rekursiivisia SQL-kyselyjä ja tarkasti suunniteltua relaatiotietokantaa. Suunnitteluvaiheessa täytyy varmistaa, että relaatiotietokantaan saadaan tallennettua kaikki hierarkiaan liittyvä informaatio. Jos kaikki suunnitteilla olevan relaatiotietokannan käyttökohteet ovat valmiiksi tiedossa, on mahdollista valita niihin parhaiten sopiva hierarkkisen datan säilytysmenetelmä. Tavallinen vieruslistakin mahdollistaa kuitenkin rekursiivisien kyselyiden käyttämisen. Tämä yhdistelmä välttää muihin vaihtoehtoihin liittyvät monimutkaisuudet, mutta mahdollistaa tehokkaiden SQL-kyselyiden toteuttamisen.

Vaarana hierarkkisen datan noutamisessa tietokannasta on lähinnä useiden pienien SQL-kyselyjen käyttäminen koko näkymän rakentamiseen. Suurien hierarkkisien näkymien generointi sadoilla tai tuhansilla kyselyillä on erittäin hidasta, koska järjestelmän komponentit joutuvat jokaisen kyselyn kohdalla toistamaan kommunikaation liittyvät vaiheet. Molemmat toteutukset olivat kuitenkin lopulta hyödyllisiä, koska niillä oli eri vahvuudet ja heikkoudet. Ensimmäinen toteutus on hidas suurien XML-tiedostojen generointiin. Se soveltuu silti paremmin reaaliaikaiseen verkkosivujen generointiin, koska se mahdollistaa erittäin tarkasti määriteltävän tiedon noutamisen, ollessaan edelleen tarpeeksi nopea. Toinen toteutus taas sopii hyvin eräajoihin ja on huomattavasti nopeampi suurien tietomäärien kanssa. Se ei kuitenkaan ole tarpeeksi joustava pienen datamäärän noutamiseen tietokannasta, koska se ei mahdollista generoitavan puurakenteen koon rajoittamista. Näiden toteutuksien suurin heikkous on kuitenkin niiden tiukka riippuvuus tietystä tietokannasta. Vaikka toteutuksen voikin kohdetietokannasta riippuen toistaa uudestaan, täytyy potentiaalisen kehittäjän kuitenkin toteuttaa käytännössä koko sovellus uudelleen. Lopulta kohdetietokannan tapa säilyttää hierarkkinen data vaikuttaa myös siihen onko toteutuksesta edes mahdollista ottaa mallia, koska se on myös riippuvainen tietystä tavasta tallentaa hierarkkista dataa.

Tietokanta soveltui erittäin hyvin toteutuksessa generoitavien näkymien datan noutamiseen, vaikka sitä ei ollut suunniteltu tämä käyttötarkoitus mielessä. Vieruslistan ja sulkeumataulukon käyttäminen arvioitiin sopivimmaksi tavaksi tallentaa hierarkkinen sanakirjadata, kun tietokantaa kehitettiin ja tämä päätös mahdollisti tässä tutkimuksessa tehdyn toteutuksen. Jos tietokanta olisi suunniteltu esimerkiksi polkulaskennan ympärille, olisi toteutus voinut olla huomattavasti erilaisempi sen vaatiman merkkijonojen käsittelyn takia. Tietokannan päälle rakennettavaa sovellusta kehittäessä on aina etu, jos voi vaikuttaa tietokannan rakenteeseen, mutta sopivan tietokannan päälle uusien sovelluksien rakentaminen on aina mahdollista. Toteutus ei juuri hyötyisi

tietokannan rakenteen muuttamisesta. Heikkoutena toteutuksessa on osittaisen näkymän generointi, koska nykyinen rekursiivinen SQL-kysely ei arvioi onko se löytänyt oikean määrän entiteettejä näkymää varten. On mahdollista, että esimerkiksi polkulaskenta tai sisäkkäiset joukot mahdollistaisivat osittaisen näkymän tehokkaan generoinnin, mutta niiden tuomat ongelmat tietokannan tilan ylläpitämissä eivät joka tapauksessa vaikuta sen arvoisilta.

Jatkokehitysideaksi jää universaalimpi työkalu, joka pystyisi hyödyntämään sqlcodegenin muodostamia tietokantamalleja ja SQLAlchemyn joustavaa tietokantakyselyjen generointia. Toisaalta esimerkiksi PostgreSQL ja muut tietokannat usein tarjoavat jo työkalun koko tietokannan tulostamiseen XML-muodossa. Tämän takia hedelmällisempi lähestymistapa universaalimpaa hierarkkisien näkymien generointityökalua kohti voi olla XML-muotoisen tietokannan suora käsittely tavallisilla XML-työkaluilla.

8 Viiteluettelo

Ala-Fossi, Jukka. 2021. Toteutus: Hierarkkisien näkymien generointi relaatiotietokannan sanakirjadatasta. Haettu osoitteesta: *https://gitlab.com/jukka_tutkielma/hierarkkisien_nakymien_generointi_toteutus* (16.11.2021).

Ben-Gan, Itzik and Tom Moreau. 2008. *Advanced Transact-SQL for SQL Server 2000. 1st ed*. Apress.

Bouvier, Dennis J. 1995. The state of HTML. *ACM Special Interest Group on Microarchitecture Bulletin*. 21, 2, 8-13.

Celko, Joe. 2012. *Joe Celko's Trees and Hierarchies in SQL for Smarties. 2nd ed*. Morgan Kaufman.

Chernysh, B.A, A.S Kartamyshev and A.V Murygin. 2020. Hierarchical Data Model Choosing in the Information Systems Design in Relational DBMS. *2020 International Multi-Conference on Industrial Engineering and Modern Technologies (FarEastCon)*. 1-5.

Cormen, Thomas H, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. 2009. *Introduction to Algorithms, 3rd Edition.* MIT Press.

Cover, Robin, Nicholas Duncan, and David T Barnard. 1991. The Progress of SGML (Standard Generalized Markup Language): Extracts from a Comprehensive Bibliography. *Literary and linguistic computing*, 6, 3, 197–209.

Fidalgo, Robson do Nascimento, Elvis Maranhão De Souza, Sergio España, Jaelson Brelaz de Castro and Oscar Pastor. 2012. EERMM: A metamodel for the enhanced entity-relationship model. *31st International Conference on Conceptual Modeling 2012*. 515-524.

Fong, Joseph. 2006. *Information Systems Reengineering and Integration. 2nd ed*. Springer.

Fong, Joseph and Herbert Shiu. 2012. An Interpreter Approach for Exporting Relational Data into XML Documents with Structured Export Markup Language. *Journal of Database Management*. 23, 1, 49-77.

Fong, Joseph, Francis Pang and Chris Bloor. 2001. Converting relational database into XML document. *DEXA Workshop 2001*. 61-65.

Gyorodi, Cornelia, Robert Gyorodi, Romulus-Radu Moldovan-Duşe and George Pecherle. 2016. Improve query performance on hierarchical data. adjacency list model

vs. nested set model. *International Journal of Advanced Computer Science & Applications*. 7, 4, 272-278.

Harrington, J. L. 2016. *Relational Database Design and Implementation (4th ed.)*. Morgan Kaufmann.

Ide, Nancy, Jacques Le Maitre and Jean Véronis. 1993. Outline of a model for lexical databases. *Information Processing and Management*. 29, 2, 159-186.

Junkkari, Marko. 2005. PSE: An object-oriented representation for modeling and managing part-of relationships. *Journal of Intelligent Information Systems.* 25, 2, 131-157.

Lemnitzer, Lothar, Laurent Romary and Andreas Witt. 2009. *Representing Human and Machine Dictionaries in Markup Languages*. Computing Research Repository, December 2009.

lxml. 2021, lxml - XML and HTML with Python, Haettu osoitteesta: *https://lxml.de/* (17.4.2021).

Maler, Eve and Jeanne El Andaloussi. 1995. *Developing Sgml Dtds: From Text to Model to Markup*. Prentice Hall.

Murthy, C. S. V. 2008. *Data Base Management Design.* Global Media.

Norri, Juhani, Marko Junkkari and Timo Poranen. 2020. Digitization of data for a historical medical dictionary. *Lang. Resour. Evaluation,* 54, 3, 615-643.

Oppel, Andy. 2015. *SQL: A Beginner's Guide*. McGraw-Hill.

SQLAlchemy. 2021. Key Features of SQLAlchemy. Haettu osoitteesta: *https://www.sqlalchemy.org/features.html* (17.4.2021).

sqlcodegen. 2021. README.rst. Haettu osoitteesta: *https://github.com/agronholm/sqlacodegen* (24.10.2021).

Tornado. 2021. Introduction. Haettu osoitteesta: https://www.tornadoweb.org/en/stable/guide/intro.html (17.4.2021).

Tsichritzis, Dennis and Frederick H. Lochovsky. 1976. Hierarchical Data-Base Management: A Survey. *ACM Computing Surveys*. 8, 1, 105-123.

Vaquero, Antonio, Francisco José Álvarez-Montero and Fernando Sáenz-Pérez. 2014. Representing computational dictionaries in relational databases. In: Rufus, Gouws, Heid Ulrich, Schweickard Wolfang, Wiegand Herbert (eds.), *Dictionaries. An In-*

ternational Encyclopedia of Lexicography. Supplementary Volume: Recent Developments with Focus on Electronic and Computational Lexicography. De Gruyter Mouton, 1209-1227.

W3C. 2021. XML schema. Haettu osoitteesta: *https://www.w3.org/standards/xml/schema* (29.9.2021).

W3C. 2021. Extensible Markup Language (XML) 1.0 (Fifth Edition). Haettu osoitteesta: *https://www.w3.org/TR/REC-xml* (15.4.2021).

W3C. 2021. HTML 5.2 W3C Recommendation. Haettu osoitteesta: *https://www.w3.org/TR/html52* (15.4.2021).

Liite 1: Kuvaa 2 vastaava XML-pätkä

```
<kirja>
    <luku tunnus=”1”>
        <kappale>Ensimmäisen kappaleen sisältö</kappale>
        <kappale>Toisen kappaleen sisältö</kappale>
    </luku>
    <luku tunnus=”2”>
        <kappale>Kolmannen kappaleen sisältö</kappale>
    </luku>
</kirja>
```

Liite 2: Esimerkki XML-skeema

```
<?xml version="1.0"?>
<xs:schema elementFormDefault="qualified"
           xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="yritys">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="osasto">
        <xs:complexType>
          <xs:sequence>

            <xs:element name="tunniste" type="xs:integer"/>
            <xs:element name="kaupunki" type="xs:string"/>

            <xs:element name="tyontekija">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="tunniste"
                              type="xs:integer"/>
                  <xs:element name="nimi" type="xs:string"/>
                  <xs:element name="kaupunki" type="xs:string"/>
                  <xs:element name="osasto" type="xs:integer"/>
                </xs:sequence>
```

```
              </xs:complexType>
            </xs:element>

          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Liite 3: Esimerkki XML

```
<?xml version="1.0"?>
<yritys>
  <osasto>
    <tunniste>1</tunniste>
    <kaupunki>Tampere</kaupunki>

    <tyontekija>
      <tunniste>1</tunniste>
      <nimi>Artola</nimi>
      <kaupunki>Tampere</kaupunki>
      <osasto>1</osasto>
    </tyontekija>

  </osasto>
</yritys>
```

Liite 4: Ensimmäisen toteutuksen pseudokoodi

```
def get_term_thesaurus(self):
    no_parents = query(Term).filter(Term.parent_id == None).all()
    thesaurus = list()
    for term in no_parents:
```

```
        thesaurus.append(self.get_all_term_children(term))
    return thesaurus

def get_all_term_children(self, parent_term):
    sense_list = list()
    variant_list = list()
    for sense in parent.senses:
        sense_list.append(dict(sense.identifier,sense.sense,
                               sense.description))
        for variant in sense.variants:
            variant_list.append(dict(variant.identifier,
                                     variant.name))
    for child in parent.children:
        child_list.append(get_all_term_children(child))
    return dict(parent.identifier, parent.term_txt,
                parent.etymology, parent.additional,
                child_list, sense_list, variant_list)
```

Liite 5: Rekursiivisen SQL-kyselyn määrittely SQLAlchemyllä

```
def get_term_thesaurus_recursive(self, parent_id):

    # Rekursiivinen SQL kysely

    parent_q = query(Term, Variant, Sense)

    parent_q = parent_q.join(Sense).join(Variant)

    parent_q = parent_q.filter(Term.parent_id == parent_id)

    parent_q = parent_q.cte(Recursive)

    child_q = query(Term, Variant, Sense)

    child_q = child_q.join(Sense).join(Variant)

    child_q = child_q.join(parent_q,

                           child_q.parent_id = parent_q.id)

    recursive_union = parent_q.union(child_q)
```

```
flat_result = query(recursive_union).all()
filtered_results = list()
r_d = dict()
# Kerätään lapsien merkitykset ja variantit,
# sitten poistetaan ylimääräiset rivit
for child in flat_result:
    filtered_results.append(if_unique(child))
    r_d[child.id][senses].append(if_unique(child.sense))
    r_d[child.id][variants].append(if_unique(child.variant))

# laitetaan kaikki kohteet oman parent_id alle
sorted_results = dict()
for child in filtered_results:
    if child.parent_id not in sorted_results:
        t_list = [dict(child.id, child.term, child.etymology,
                       child.additional, children=[],
                       sensest=r_d[child.id]['senses'],
                       variants=r_d[child.id]['variants'])]
        sorted_results[child.parent_id] = t_list

# siirretään lapset vanhempiensa alle
for parent_key, parent_value in sorted_results:
    for child in parent_value:
        if child.id in sorted_results:
            child['children'] = sorted_results[child.id]
# palutetaan vanhempi, jonka alle lapset on siirretty
return sorted_results[parent_id]
```

Liite 6: Rekursiivinen SQL-kysely

```
WITH RECURSIVE cte AS (

     (SELECT

        terms.id, terms.parent_id, terms.term, terms.etymology,
terms.additional, sense_terms.id, variant_usages.id, variants.id,
variants.name, senses.id, senses.sense, senses.description

    FROM

        terms

    LEFT OUTER JOIN

        sense_terms ON terms.id = sense_terms.term_id

    LEFT OUTER JOIN

        senses ON senses.id = sense_terms.sense_id

    LEFT OUTER JOIN

  variant_usages ON sense_terms.id = variant_usages.sense_term_id

    LEFT OUTER JOIN

        variants ON variants.id = variant_usages.variant_id

    WHERE

        terms.parent_id = %(parent_id_1)s ORDER BY terms.id

    )

    UNION

    (SELECT

        terms.id, terms.parent_id, terms.term, terms.etymology,
terms.additional, sense_terms.id, variant_usages.id, variants.id,
variants.name, senses.id, senses.sense, senses.description

    FROM

        terms

    JOIN
```

```
        sense_terms ON terms.id = sense_terms.term_id
    JOIN
        senses ON senses.id = sense_terms.sense_id
    JOIN
variant_usages ON sense_terms.id = variant_usages.sense_term_id
    JOIN
        variants ON variants.id = variant_usages.variant_id
    JOIN
        cte ON terms.parent_id = cte.id ORDER BY terms.id
    )
)
SELECT
    *
FROM cte
```

Liite 7: Lmxl generoima XML-dokumentti

```
<thesaurus>
  <term identifier="term41">
    <termText>access of fever</termText>
    <etymology/>
    <additional/>
    <senses>
      <sense identifier="sense39">
        <senseText>attack of fever</senseText>
        <description>esp. periodic</description>
      </sense>
    </senses>
    <variants>
      <variant identifier="variant127">
        <variantText>acces</variantText>
      </variant>
```

```
    </variants>
    <narrowerTerms/>
  </term>
</thesaurus>
```

Liite 8: XML-dokumentin generoiva pseudokoodi

```
def create_term_thesaurus_xml(terms):
    thesaurus = etree.Element('thesaurus')
    for term in terms:
        term_xml_element = term_maker(terms)
        thesaurus.append(term_xml_element)
    return etree.tostring(thesaurus, pretty_print=True)

def term_maker(term):
    term = etree.Element('term', id='term' + term.id)
    term_text_elem = etree.SubElement(term, 'termText')
    term_text_elem.text = term.term_text
    etymology_elem = etree.SubElement(term, 'etymology')
    etymology_elem.text = term.etymology
    additional_elem = etree.SubElement(term, 'additional')
    additional_elem.text = term.additional
    if 'sense_list' in term:
        senses = etree.Element('senses')
        for sense in term.sense_list:
            senses.apend(sense_maker(sense.id,
                                     sense.sense_text,
                                     sense.desc))
            term.append(senses)
    if 'variant_list' in term:
        variants = etree.Element('variants')
        for variant in term.variant_list:
            var_elem=etree.Element('variant',
                                   id='variant'+variant.id)
            variant_text = etree.SubElement(var_elem,
                                            'variantText')
            variant_text.text = variant.name
            variants.append(var_elem)
```

```
    term.append(variants)
children = call_term_maker_for_children(term.children)
narrower_terms = etree.Element('narrowerTerms')
for child in children:
    narrower_terms.append(child)
term.append(narrower_terms)
return term
```