TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Informaatikainstituut
Infosüsteemide õppetool

Andmebaasi loogilise ja füüsilise disaini antimustrite esinemine mõnedes vaba tarkvara poolt kasutatavates SQL-andmebaasides

Bakalaureusetöö

Üliõpilane: Arne Lapõnin
Üliõpilaskood: 120920IAPB
Juhendaja: Erki Eessaar

Tallinn
2015

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

---------- ----------

(*kuupäev*) (*allkiri*)

Annotatsioon

Andmebaasi loogilise ja füüsilise disaini antimustrite esinemine mõnedes vaba tarkvara poolt kasutatavates SQL-andmebaasides

Töö eesmärgiks on uurida, millised antimustrid esinevad tänapäeva vaba tarkvara poolt kasutatavates SQL-andmebaasides.

Tarkvara antimustrid kätkevad endas peamisi probleeme, mida tehakse tarkvara arendamisel. Selle mustrite kategooria alamkategooriaks on SQL-andmebaaside disaini antimustrid. Tänapäeva infoühiskonnas on äärmiselt oluline, et andmebaasid, millele on rajatud kõik informatsiooni töötlevad rakendused, oleks disainitud nii, et nad ei sisaldaks aastakümneid tarkvaratööstust vaevanud vigu.

Töö annab konkreetsete süsteemide näitel aimu sellest, kas SQL-andmebaasid, mida kasutab vaba tarkvara, on hästi või halvasti disainitud.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 48 leheküljel, 3 peatükki, 7 joonist, 7 tabelit.

Abstract

Appearance of Logical and Physical Database Design Antipatterns in Some SQL Databases Used by Free Software

The aim of this work is to find out which design antipatterns can be found in today's SQL databases that are used by free software.

Software antipatterns encapsulate the main problems of software design. A subcategory of the category is SQL database design antipatterns. It is vital that databases, on which almost all applications depend on, contain as little of these antipatterns as possible.

This work should provides, based on the example of concrete systems, insights as to whether SQL databases that are used by free software, have been designed adequately or not.

The thesis is in Estonian and contains 48 pages of text, 3 chapters, 7 figures, 7 tables., etc.

Lühendite ja mõistete sõnastik

EMR ***Electronic medical record***

Digilugu (Electronic health record 19.05.2015)

ERP ***Enterprise resource planning***

Ettevõtte ressursside planeerimise tarkvara. (Enterprise resource planning 19.05.2015)

FK ***Foreign key***

Välisvõti

Kandidaatvõti ***Candidate key***

Korteeži (SQL-andmebaasi korral rea) unikaalne identifikaator, millest ei saa eemaldada atribuute, kaotamata unikaalsuse omadust. (Eessaar 2008, 62)

PK ***Primary key***

Primaarvõti

SQL ***Structured Query Language***

Struktuurpäringukeel on andmebaasi päringukeel relatsiooniliste andmebaaside jaoks (Struktuurpäringukeel 10.05.2015). Lisaks päringute ehk andmete otsimiste lausetele kuuluvad sellesse ka laused andmete muutmiseks, transaktsioonide juhtimiseks, andmebaasi struktuuri ja käitumise kirjeldamiseks ning õiguste jagamiseks.

SQL-andmebaas ***SQL database***

Andmebaas, mis on loodud kasutades andmebaasisüsteemi, mis toetab SQL andmebaasikeelt.

Surrogaatvõti ***Surrogate key***

Surrogaatvõti on süsteemi poolt genereeritud identifikaator, mida

kasutatakse korteežide (SQL-andmebaaside korral ridade) eristamiseks ega oma kasutaja jaoks sisulist tähendust. (Eessaar 2008, 64)

Jooniste nimekiri

Tabelite nimekiri

Sisukord

Sissejuhatus

Andmebaasis, kui infosüsteemi tuumas, hoitakse seotud andmeid, mis on ettevõtjale või üksikisikule vajalikud, et rahuldada tema infovajadusi. Antud töös käsitleme termineid andmed ja informatsioon kui sünonüüme. Tihti määrab ettevõtte edukuse see, kuidas ta suudab kasutada enda valduses olevat informatsiooni. Saavutamaks seatud eesmärke, peavad ettevõtjad ja üksikisikud hoolitsema selle eest, et nende infosüsteemi aluseks olev andmemudel ja andmebaasi tehniline kirjeldus toetaks nende praeguseid ning tulevasi äriprotsesse. Kuna kõikidel ei ole inim- ja kapitaliressurssi, et luua infosüsteem algusest, tihtipeale ei ole see isegi vajalik, võetakse kasutusele mõni valmis tehtud lahendus. Viimase kümne aasta jooksul on Oracle'i, IBM-i, Microsofti kõrvale tõusnud terve rida väiksmaid tegijaid, nt Odoo, xTuple jpt, kes pakuvad oluliselt odavamaid lahendusi, mis on tihtipeale ka vabavaralised. Käesoleva tööga üritan saada selgust, kas avatud lähtekoodiga infosüsteemide andmebaasid on hästi ja õigesti disainitud.

2001. aastal avaldas Michael Blaha kaks artiklit (Blaha, 2001a; Blaha, 2001b), milles ta võttis kokku SQL-andmebaaside disainiprobleemid, millega ta puutus kokku eelnenud kümne aasta jooksul. Käesolevas töös üritan analüüsida, kas tänapäevastes andmebaasides ilmnevad samad vead või on need kadunud. Ühtlasi uurin andmebaaside loogilist ja füüsilist disaini, toetudes Bill Karwini tööle SQL-andmebaaside antimustrite vallas (Karwin, 2010).

Töö eesmärgid on:

1. Tutvuda peamiste andmebaasi disaini antimustritega.
2. Uurida, kas Michael Blaha välja toodud vead korduvad tänapäeva SQL andmebaasides, mida kasutavad avatud lähtekoodiga rakendused.
3. Uurida, millised disaini disainiprobleemid esinevad veel nendes andmebaasides. Selliseid probleeme saab tulevikus kirjeldada SQL antimustrite formaadis.

Antimustreid otsin xTuple avatud lähtekoodiga rakendusest Postbooks ja vabavaralisest ning avatud lähtekoodiga OpenEMR andmebaasidest. Postbooks rakenduse SQL-andmebaas valiti peamiselt lähtekoodi kättesaadavuse tõttu, lisaks figureerib xTuple paljudes soovitavate

ettevõtte ressursside planeerimise tarkvara nimekirjades (Rubens 23.05.2015). OpenEMR valiti, sest tegu on populaarse valikuga e-tervise valdkonnas (OpenEMR 19.05.2015). Postbooks kasutab PostgreSQLi ja on mõeldud ettevõtete ressursside planeerimiseks, OpenEMR kasutab MySQL andmebaasisüsteemi ning on suunatud e-tervise valdkonnale. Töö keskendub avatud lähtekoodiga (vabadele) tarkvarasüsteemidele, sest nende populaarsus ja olulisus ajas kasvab. Need süsteemid on uurimiseks vabalt kättesaadavad ning nii on võimalik käsitleda eritüübilisi süsteeme. Lisaks oli juhendaja poolne soov kaitsta seda lõputööd kindlasti avalikul kaitsmisel, et ka töö tulemused oleksid avalikud. Kui ettevõtte arendab firmasisest tarkvara, siis ei pruugi ta soovida avalikustada selles sisalduvaid puuduseid.

Töö on jagatud kahte suurde ossa. Esimene osa on teoreetiline, see annab põgusa ülevaate disainimustritest ning seejärel tutvustatakse antimustreid. Järgneb kokkuvõte Michael Blaha tööst andmebaaside vigade ja antimustrite kaardistamisel ning lõpuks ülevaade Bill Karwini kokku kogutud antimustritest. Teine suur osa on praktiline, mille käigus analüüsitakse valitud andmebaase lähtuvalt varem kirjeldatud antimustritest.

1. Antimustrid

Selles peatükis antakse teoreetiline ülevaade mustritest ja antimustritest.

1.1 Mis on disainimuster?

Sõnastamaks antimustri mõistet, on mõistlik alustada sellest, mis on muster tarkvaraarenduse kontekstis. Üldiselt on disainimuster formaalne viis dokumenteerimaks lahendust probleemile mingis valdkonnas (Design pattern 03.15.2015). Objektorienteeritud tarkvara arenduses mõistetakse disainimustrit, kui kirjeldust kogumikust üksteisega suhtlevatest objektidest ja klassidest, mis on kohandatud lahendama üldist disaini probleemi mingis kindlas kontekstis (Gamma jt. 1995, 3).

Gamma, Helm, Johnson ja Vlissides (edaspidi *The "Gang of Four"*) on kirjutanud, et disainimuster koosneb neljast osast:

1. Mustri nimi identifitseerib disaini probleemi, selle lahendusi ja tagajärgi. Nimi aitab läheneda probleemile kõrgemal abstraktsioonitasemel ja lihtsustab arendajate omavahelist suhtlust. (*Ibid*, 3)

2. Disaini probleem näitab, millal antud disainimustrit kasutada saab. See selgitab probleemi ja konteksti. Probleemi kirjeldamise käigus võib määrata eeldused, mis peavad olema täidetud, et disainimustrit saaks kasutada. (*Ibid*, 3)

3. Lahendus määrab elemendid, mis moodustavad disaini, selles olevad sõltuvus- ja vastutussuhted. Lahendus ei tohiks sisaldada liiga konkreetseid disaini- või rakenduspõhimõtteid, sest nii nagu muster kirjeldab probleemi abstraktsel tasemel, peaks ka lahendust saama kasutada šabloonina mitmete sarnaste probleemide korral. (*Ibid*, 3)

4. Tagajärjed on disainimustri kasutamise tulemused või kompromissikohad. Tagajärgi tuleb arvestada, valides alternatiivide vahel ja analüüsides mustri positiivseid ja negatiivseid tahke. (*Ibid*, 3)

Disainimustrite teadmise ja kasutamise vajalikkus seisneb selles järgnevas:

- Aitavad muuta tarkvara taaskasutatavaks ja paindlikuks.
- Lihtsustavad lahenduste dokumenteerimist, suurendades nende kättesaadavust.
- Abistavad alternatiivsete lahenduste seast parima leidmisega.
- Kiirendavad süsteemide disainimist.
- Hõlbustavad abstraktset disainimist.
- Soodustavad süsteemide hooldamist, säilitades disaini algpõhimõtteid. (Gamma jt. 1995, 1-4)

Olles sõnastanud disainimustrite mõiste, koosluse ja vajalikkuse, saab nüüd kirjeldada nende nimelisi vastandeid, ehk antimustreid.

1.2 Mis on antimuster?

Töö tarkvaraarenduse disainimustritega algas aastal 1987, kui Ward Cunningham ja Kent Beck hakkasid tuntud arhitekti Christopher Alexanderi õppetunde rakendama tarkvaraarenduses. Disainimustrid hakkasid saama laiemat tähelepanu 1994. aastal, kulmineerudes *The "Gang of Four"* kuulsa raamatuga "*Design Patterns: Elements of Reusable Object-Oriented Software*". Tarkvaraarenduse kogukond oli nendest arengutest vaimustuses, ent leidus ka inimesi, kes suutsid häid disainimustreid kasutada vales kontekstis või ei osanud disainimustrite abil probleeme lahendada. Üpris pea saadi aru, et disainimuster muutub antimustriks, kui see tekitab rohkem probleeme, kui see lahendab. (Brown jt. 1998, 9-11)

Antimustrite üle hakati arutlema peaaegu samal ajal kui disainimustrite üle ja laiema avalikkuse tähelepanu osaliseks said need 1996. aastal, kui Michael Akroyd pidas Object World West konverentsil nende teemalist esitlust. Kuid antimustrite kirjeldamisesse on panustanud paljud inimesed, seega konkreetne alustaja puudub, pigem peetakse neid disainimustrite edasiarenguteks. (*Ibid*, 12)

Disaini- ja antimustri struktuurne erinevus seisneb selles, et disainimuster koosneb probleemist ja lahendusest. Probleem koosneb kontekstist ja disaini mõjutanud teguritest. Tänu lahendusele ilmnevad tagajärjed, millest saadakse kasu, aga võivad ilmneda ka uued

probleemid, mida hakatakse järgmiste mustritega lahendama. Antimuster koosneb aga kahest lahendusest, millest esimene on vigane ja tekitab rohkem probleeme, kui see lahendab. Teine lahendus, mida nimetatakse refaktoreeritud lahenduseks, peab tagama, et antimustri tekitatud probleemid saaksid lahendatud. (Brown jt. 1998, 15-16)

Tarkvaraarenduse antimustrite tundmise vajalikkus seisneb järgnevas:

1. Juhivad tähelepanu tüüpilistele probleemide tarkvaraarenduses.
2. Pakuvad sõnavara, tuvastamaks probleeme ja nende lahendusi.
3. Soodustavad disainimustrite kasutamist õiges kontekstis. (Brown jt. 1998, 8)

Tarkvaraarendaja peab tundma disainimustreid, et disainida paindliku ja taaskasutatavat tarkvara, ent sama tähtis on mõista antimustreid, et vältida vigu, mis võivad tekkida disainimustrite kasutamisel või siis hoopis kogenematusest või ignorantsusest tingitud kasutamata jätmisest. Käesolevas töös võetakse fookusesse SQL-andmebaaside (edaspidi lihtsalt *andmebaaside*) disainimisel tehtavad vead ja antimustrid.

1.3 Michael Blaha töö

2001. aastal esitles Michael Blaha Saksamaal toimunud kaheksandal pöördprojekteerimise konverentsil andmebaaside pöördprojekteerimise tulemusi. Kümne aasta jooksul töötas ta üle 50 andmebaasiga, millest oma artiklites kirjeldas 35, jättes välja mõned esimesed katsed, mille kohta tal puudus piisavalt märkmeid. Ainult üks uuritavatest andmebaasidest ei olnud relatsiooniline. Blaha tõi välja korduvad vead andmebaaside disainis ja analüüsis iga andmebaasi ja andis disainile hinnangu viie palli skaalal. Tabelis 1. on välja toodud uuritavate andmebaaside disaini detailid (B1 – B5) ja antimustrid (B6 – B9). Märgistasin iga vea koodiga B1 – B9, et neile oleks hiljem tekstis mugavam viidata. (Blaha 2001a, 136) Kasutan siin ja edaspidi sõna „tabel“, et viidata baastabelite ning sõna „vaade“, et viidata virtuaalsetele tabelitele.

Tabel 1. Blaha disaini detailid ja antimustrid

Kood	Nimi	Kirjeldus
B1	Primaarvõtme deklareerimine	Primaarvõti on kombinatsioon tabeli veergudest (sellesse kuulub üks või rohkem veergu), millele vastav väärtuste kombinatsioon identifitseerib tabeli rida.
B2	Välisvõtmete deklareerimine	Välisvõti on viide mõne teise (või sama) tabeli primaarvõtmele või primaarvõtmeks mitte valitud kandidaatvõtmele. Välisvõtmed seovad tabeleid ja nende ridu.
B3	Primaarvõtme välja piirang	Kas väärtuse puudumise marker NULL (tuntud ka kui NULL väärtus) on lubatud või mitte? Andmebaaside teooria järgi on kõik primaarvõtme veerud kohustuslikud ega tohi sisaldada NULLe.
B4	Andmetüüpide järjekindlus	Kas sarnased veerud kasutavad samu andmetüüpe ja väljade pikkusi?
B5	Indeksid	Kas primaar-, kandidaat- ja välisvõtmetele on tehtud indeksid? Indeksite kasutamine parandab jõudlust.
B6	Paralleelsed veerud	Kas atribuudi erinevate väärtuste hoidmiseks on kasutatud mitut samatüübilist ja sarnase nimega veergu?

Kood	Nimi	Kirjeldus
B7	Tugevalt denormaliseeritud tabelid	Kas tabelid on viiendal normaalkujul? Mida kõrgemal normaalkujul on tabelid, seda vähem korduvat ja võimalik et vastuolulist informatsiooni neis leidub.
B8	Veergude arv tabelis	Suur veergude arv tabelis osutab võimalikule tugevale denormaliseerimisele (kuid ei pruugi seda tähendada).
B9	Klassifikaatorid	Kas klassifikaatorid on deklareeritud ehk neid hoitakse stringina samas või eraldi tabelis, või on nad kodeeritud kujul?

Analüüsides andmebaasi disaini detaile leidis Blaha, et umbes 75% andmebaasides jõustati primaarvõtme ning kõigest 10% välisvõtme kitsendust. 40% andmebaasidest lubas NULLe primaarvõtme veergudes, minnes vastuollu andmebaaside teooriaga. SQLis tähendab PRIMARY KEY kitsendus automaatselt ka vastavate veergude kohustuslikkust ning seega pole rangelt võttes vaja primaarvõtme veerule NOT NULL kitsendust määrata. Samas skeemi evolutsiooni käigus võib tabeli võti muutuda ja siis on hea kui NOT NULL kitsendused on juba olemas. Enamikus andmebaasides (80%) kasutati andmetüüpe järjekindlalt. Umbes veerandis andmebaasides kasutati indekseid rahuldavalt ja umbes sama paljud kasutasid indekseid minimaalselt või üldse mitte. (Blaha 2001b, 147-150)

Antimustrite kasutust analüüsides, täheldas Blaha, et 20% andmebaasides kasutati paralleelseid veerge laialdaselt. Paralleelsete veergude kasutamine raskendab päringute tegemist ja vähendab andmebaasi paindlikust, raskendades uute ärireeglite rakendamist. (*Ibid*, 150) Normaliseerimise eesmärgiks on vähendada andmete liiasust ja selle tulemusel vähendada andmete muutmise anomaaliaid, muuta andmebaasi struktuur kasutajale arusaadavaks ning lihtsustada teatud kitsenduste jõustamist (Eessaar 2008, 237). Normaliseerimise pöördprotsessi, ehk tabelite normaliseerituse astme vähendamist, kutsutakse denormaliseerimiseks. Denormaliseerimist võib kaaluda eesmärgiga parandada mõningate

andmebaasis toimuvate päringute töökiirust. (*Ibid*, 241) Denormaliseerimist tohib teha kaalutletult ja eesmärgipäraselt ja pigem analüütilistes rakendustes, kus kasutakse andmebaase, milles lugemisoperatsioone on tunduvalt rohkem kui kirjutamisoperatsioone. 33 andmebaasi 35-st, mida Blaha uuris, kuulusid operatsiooniliste (tehingutöötlusele orienteeritud) rakenduste koosseisu. Nendes on lugemis- ja kirjutamisoperatsioonide hulk enamvähem võrdne. Blaha leidis, et 50% operatsioonilistes andmebaasides leidus vähemalt üks tugevalt denormaliseeritud tabel. Pooltes andmebaasides hoiti klassifikaatoreid kodeeritud kujul ja need deklareeriti ainult kaheksas. (Blaha 2001b, 152)

2001. aastal kümne aasta pikkust tööd kokku võttes, tõdes Blaha, et tollased tarkvara loomise tavad olid puudulikud ja arendajatel oli palju ruumi oma teadmiste ja oskuste taseme tõstmiseks. Käesoleva tööga üritan saada selgust, kas 15 aastaga on midagi muutunud.

1.4 Bill Karwini kirja pandud antimustrid

Bill Karwin kogus raamatusse "*SQL Antipatters*" kõige levinumad vead, mida inimesed, aga ka tema, on teinud, kasutades SQLi. Ta on jaganud antimustrid nelja kategooriasse.

1. Andmebaasi loogilise disaini antimustrid

   Loogiline disain tegeleb konkreetsetest realisatsiooni- ja rakenduskeskkonnast sõltumatute lahenduste loomisega. Loogilise andmebaasi disaini moodustavad loogilise andmemudeli koostamine ja andmebaasioperatsioonide loogiline kavandamine. SQL-andmebaasi loogilise disaini tulemusel pannakse paika andmebaasi tabelid, veerud ja tabelite vahelised suhted ning andmete kasutamise elementaarteenused. (Eessaar 2008, 279 - 281)

2. Andmebaasi füüsilise disaini antimustrid

   Füüsiline disain häälestab loogilise disaini lahendusi konkreetsete riist- ja tarkvara produktide jaoks (Eessaar 2008, 280). Andmebaasi füüsiline disain tähendab andmebaasi kavandamist konkreetset andmebaasisüsteemi silmas pidades. Sinna hulka kuulub ka meetodite valimine ja rakendamine jõudluse ning turvalisuse tagamiseks, sest need on andmebaasisüsteemide spetsiifilised. Andmebaasi füüsilise disaini tegevuste hulka kuuluvad näiteks identifikaatorite ja andmetüüpide määramine, indeksite valimine, SQL-andmebaaside korral ka denormaliseerimine.

3. Päringute ja andmemuudatuste antimustrid

   Päringute ja andmemuudatuste tegemise eesmärgiks on andmebaasi lisada või seal saada andmeid. SQL-andmebaasides kasutatakse selle jaoks SQL andmekäitluskeele lauseid SELECT, UPDATE, INSERT, DELETE ja MERGE.

4. Rakenduste arendamise antimustrid

   SQLi tuleb väga sageli kasutada ka rakendustes. Need on kirjutatud mõnes teises keeles, nt Java, C++, Ruby, PHP.

Antud töös kontsentreerun esimesele kahele kategooriale ehk andmebaasi loogilise ja füüsilise disaini antimustritele. Karwini antimustritele ei ole vaja koode lisada, sest nende nimed on piisavalt unikaalsed ning neile on mugav tekstis viidata. Kasutan viitena ingliskeelseid nimesid, kuid käesoleva töö ühe kõrvaltulemusena pakun iga mustri jaoks välja ka võimaliku eestikeelse nime, mille esitan Tabel 2 ingliskeelse nime järel sulgudes.

Tabel 2. Karwini antimustrite kirjeldused

Nimi	Kirjeldus
Jaywalking (Ristmiku vältimine)	Väärtuste hoidmiseks kasutatakse nimekirja, kus väärtused on eraldatud komadega. Nii üritatakse näiteks vältida vahetabeli loomist binaarsele mitu-mitmele seosetüübile.
Naive trees (Naiivne puu)	Antimuster puudutab hierarhilisi andmeid. Ilmneb siis, kui tabeli reas on viide sama tabeli mõnele muule reale. Tekib puude esitamiseks mõeldud andmestruktuur, kus iga laps on seotud ainult oma vanemaga (vt. Joonis 1.). Sellist disaini nimetatakse ka lähedusnimekirjaks või külgnevusnimistuks (ingl *adjacency list*)
ID Required (ID, palun)	Surrogaatvõtme kasutamine sisulise tähendusega võtme või liitvõtme asemel.
Keyless Entry (Võtmevaba sisestus)	Välisvõtme piirangud jäetakse realiseerimata, mille tõttu süsteem ei kontrolli viidete terviklikkust.

Nimi	Kirjeldus
Entity-Attribute-Value (Olem-Tunnus-Väärtus)	Objektorienteeritud programmeerimisest tuntud pärimist realiseeritakse baastabeli ja atribuutide tabeli abil. (vt Joonis 2. ja Joonis 3.)
Polymorphic Associations (Polümorfne ühendus)	Antimuster tekib, kui tabeli veerg võib viidata rohkem kui ühele tabelile. Sel juhul loobutakse andmebaasi tasemel deklaratiivsest välisvõtme piirangu jõustamisest ja viidatava tabeli nime hoitakse eraldi veerus.
Multicolumn Attributes (Mitmeveerulised tunnused)	Sarnaneb *Jaywalking* antimustrile. Kui tabeli atribuudil võib olla mitu väärtust, siis luuakse mitu veergu, millest iga üks hoiab erinevat väärtust.
Metadata Tribbles (Metaandmeküülikud)	Kui andmebaasi tabelites on palju andmeid, võib tekkida kiustatus tekitada juurde tabeleid või veerge vastavalt tabeli mingi veeru unikaalsetele väärtustele.
Rounding Errors (Ümmardamisvead)	Paljud arendajad kasutavad SQLis harjumusest FLOAT andmetüüpi, mis võib oma sisemise loogika tõttu tekitada probleeme.
31 Flavours (31 eri maitset)	Veeru väärtuste piiramine, kasutades kitsendusi, kus võimalikud väärtused mis veerus võivad olla (ja mis võivad aja jooksul muutuda) on kitsenduse spetsifikatsioonis ülesloetud.
Phantom Files (Illusioonfailid)	Faile hoitakse väljaspool andmebaasi ja andmebaasis hoitakse teekonda nendeni.
Index Shotgun (Indeksitega laetud püss)	Indeksite kasutamine, mõtlemata läbi, kuidas neid peab kasutama parima tulemuse saavutamiseks.

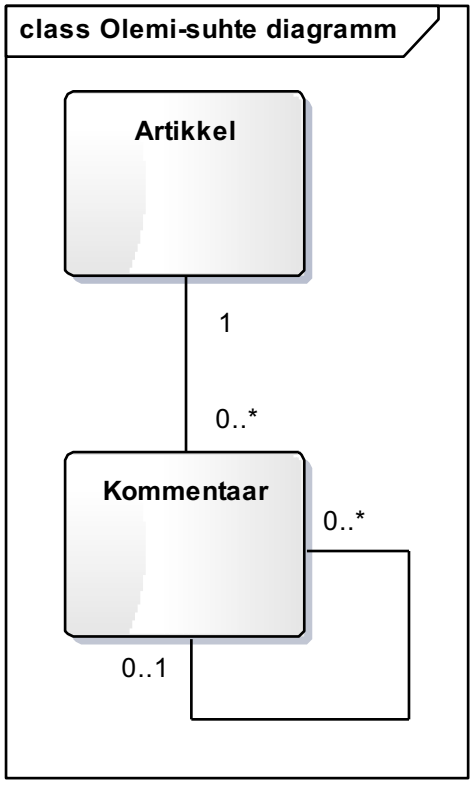
1.4.1 Jaywalking

Jaywalking antimustri peamised probleemid:

- Päringute tegemine on keeruline. Kui sellises väljas hoitakse välisvõtme väärtuseid, siis on näiteks keeruline päringuga tabeleid ühendada.
- Uue väärtuse lisamine nimekirja on lihtne, aga tõenäoliselt tuleb mõelda nimekirja sorteerimise peale ja see on juba keerulisem. Nimekirjast kustutamine nõuab juba mitut sammu.
- Keeruline on kontrollida nimekirja lisatud andmete õigsust.
- Nimekirja pikkusele seab piirangud andmevälja pikkus. (Karwin 2010,15 - 17)

Antimustri laheduseks on tabeli loomine, kus iga nimekirjas oleva väärtuse kohta on eraldi rida. Näiteks, kui sellist veergu kasutatakse mitu-mitu seose realiseerimiseks, siis on lahenduseks vahetabeli loomine, mille primaarvõtme moodustavad esimese ja teise tabeli välisvõtmed. Vahetabel lihtsustab esimese ja teise tabeli ühendamisoperatsioone ning lisamisi ja kustutamisi. Kui süsteem teab välisvõtmeid, siis oskab see kontrollida viidete terviklikkuse reegli täidetust. Välisvõtmete abil saab kontrollida andmete õigsust. Vahetabel võimaldab ka veergude indekseerimist, mis parandab jõudlust. (*Ibid*, 19 - 21)

1.4.2 Naive trees



Joonis 1. Lähedusnimekirja näite olemi-suhte diagramm

Väga sagedane operatsioon puustruktuuride põhjal on sõlme kõikide otseste ja kaudsete järglaste leidmine. See disain on problemaatiline siis, kui andmebaasisüsteemis kasutatav SQL mägimurrak ei paku lausekonstruktsioone sellise struktuuri põhjal rekursiivete päringute tegemiseks. Sellises süsteemis saab järglaste taseme leidmiseks tabelit ühendada iseendaga, kuid see võimaldab päringuga liikuda vaid kindla tasemini, mitte rekursiivselt kõige sügavama tasemeni.

Antimustrit saab lahendada:

- Kui lisada tabelisse veerg, milles hoitakse teekonda juurest kuni antud sõlmeni.
- Kui hoida informatsiooni iga sõlme parema ja vasaku järglase kohta.
- Teha tabel, milles hoida informatsiooni iga vanem-järglane paari kohta, isegi, kui nende vahele jääb mõni sõlm. (Karwin 2010, 40)

1.4.3 ID Required

ID Required antimustri probleem ilmneb hästi vahetabelite puhul, mis tekivad M:N suhetest. Selliste vahetabelite korral on mõistlik kasutada liitvõtit, millest üks pool viitab esimesele,

teine teisele tabelile. Liitvõti garanteerib võtmete kombinatsioonide unikaalsuse. Kui kasutada surrogaatvõtit, ei ole selline unikaalsus kindlustatud, mille tõttu võivad osad kombinatsioonid korduda. Kui jõustada unikaalsuse kitsendus, siis on surrogaatvõti üleliigne. (Karwin 2010, 46 - 47)

Antud antimustril on lihtsad lahendused. Kui tabelis on veerg, mille väärtus on igas reas unikaalne (st see identifitseerib rida) ja kuhu ei saa lisada NULLe, siis ei ole tarvidust tekitada surrogaatvõtit. Sama põhimõte kehtib tabelis, mille ridu on võimalik identifitseerida mitme atribuudi abil. (*Ibid*, 51 - 52)

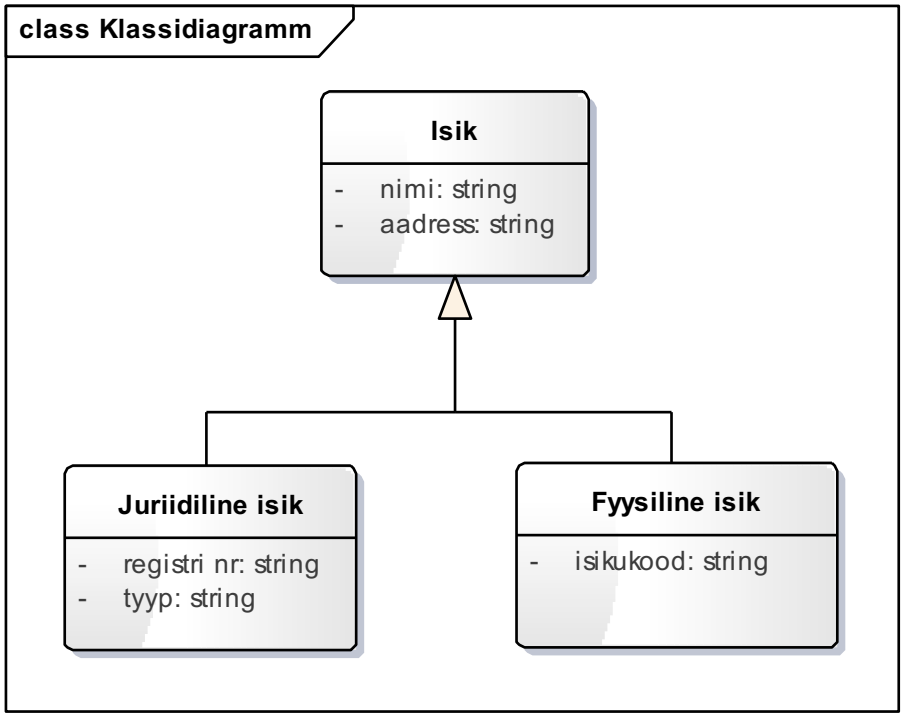
1.4.4 Keyless Entry

Välisvõti on tabeli veergude hulk (kuhu kuulub üks või mitu veergu), mis viitab teise või sama tabeli kandidaatvõtmele, milleks on enamasti primaarvõti. Viidete terviklikkuse reegel ütleb, et andmebaasis ei tohi olla ühtegi välisvõtme väärtust, millele ei leidu vastavat kandidaatvõtme väärtust. (Eessaar 2008, 68) Kuigi SQLi puhul võib välisvõtme väärtus ka puududa (olla NULL), kui vastav veerg NULLe lubab.

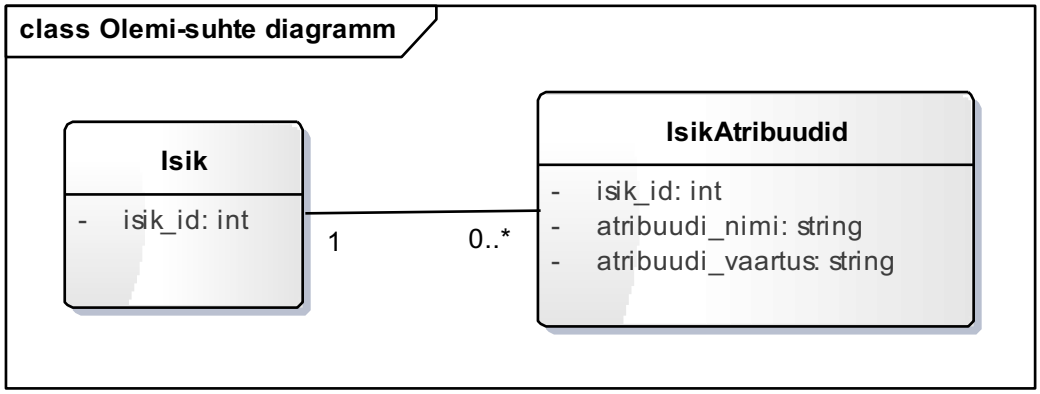
Ometi eksisteerib andmebaase, kus seda fundamentaalset reeglit rikutakse, jättes jõustamata välisvõtmed. See võib juhtuda, sest andmebaasisüsteemid ei toeta sellist võimalust või arendaja laiskusest. Tagajärjeks on, et andmebaasiga seotud rakendused toimivad ainult siis, kui kõik sisestatavad andmed on koguaeg perfektsed, mis on tegelikkuses võimatu. Et parandada tekkivaid andmete konflikte, peavad arendajad kirjutama programme, mis otsivad vigu, mis on aga kohmakas meetod ega laiene hästi suurtele süsteemidele. (Karwin 2010, 54 - 55) Lisaks annavad välisvõtmed andmebaaside kasutajatele (arendajad, rakendused, andmebaasisüsteem ise) infot, mida need oma huvides saavad ära kasutada. Välisvõtmed muudavad andmebaasi skeemi arusaadavamaks ning võimaldavad ka andmebaasisüsteemidel sisemiselt oma tööd optimeerida.

Mida varem andmebaasisüsteem avastab vead, seda lihtsam on neid parandada, seega on välisvõtmete deklareerimine kriitilise tähtsusega. Tänapäevastes andmebaasisüsteemis on olemas kompenseerivad tegevused, mis aitavad tagada viidete terviklikkuse reegli kehtivust teatud andmemuudatuste olukorras (ON UPDATE/DELETE CASCADE/RESTRICT). Seega saab arendaja ära hoida palju probleeme, kasutades välisvõtme kitsendust ja kompenseerivaid tegevusi.

1.4.5 Entity-Attribute-Value



Joonis 2. Isiku klassidiagramm



Joonis 3. *Entity-Attribute-Value* näite olemi-suhte diagramm

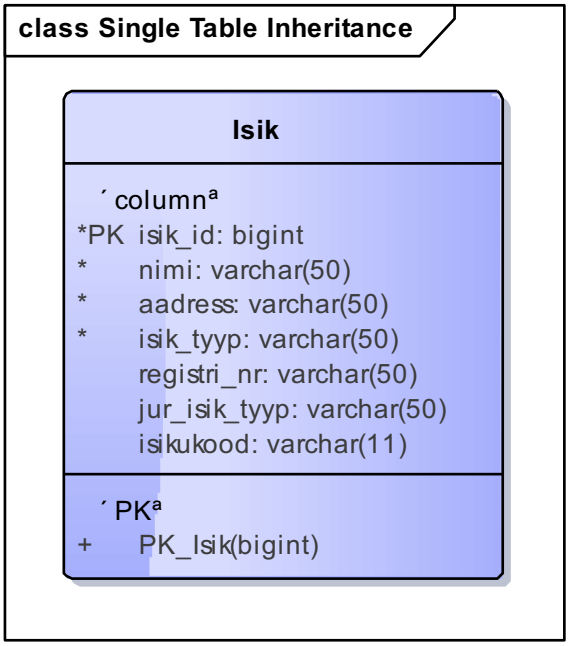
Entity-Attribute-Value (EAV) antimustri järgi tehakse näiteks uus tabel, veergudega: *id*, mis viitab baastabelile, *atribuudi_nimi* ja *atribuudi_väärtus*. Selline disain lubab lisada dünaamiliselt uusi atribuute, samas tekitab see hulga probleeme. Sellist EAV disaini järgides ei ole võimalik muuta atribuute kohustuslikeks (sest samas veerus on koos nii kohustuslike kui ka mittekohustuslike atribuutide väärtused) ega kasutada erinevate atribuutide puhul erinevaid andmetüüpe. Süsteemis ei saa deklareerida üle nende väärtuste välisvõtmeid ja seega ei jõusta süsteem viidete terviklikkuse reeglit. Viidete terviklikkuse kontrolli tuleb teha

programmi järgi andmeid haldavates rakendustes. Et saada tavalisele tabelile iseloomulikku tulemuste rida, tuleb iga atribuudi kohta teha tabelite ühendamisoperatsioon, mis atribuutude arvu suurenedes muutub väga kulukaks. (Karwin 2010, 62 - 67)

EAV antimustri kasutamisel üldistuste puhul on mitu lahendust, mis töötavad väga hästi, kui baastüübil on lõplik arv alamtüüpe. Rõzova (2011) kirjutab selle ülesande lahendamiseks koguni 14 disainimustrit.

- Ühe tabeli pärimine (*Single Table Inheritance*)

  Üla- ja alamtüüpide atribuutide väärtuseid hoitakse ühes tabelis. Alamtüüpide eristamiseks luuakse spetsiaalne veerg, milles hoitakse tüübi identifikaatorit. (vt. Joonis 4.) Kasutades seda mustrit luuakse hierarhia kohta ainult üks tabel, andmete saamiseks ei pea tegema ühendamisoperatsioone ja hierarhiat saab lihtsalt muuta. Mustri nõrkusteks peetakse keerulisust, sest on raske aru saada, millised väljad kuuluvad millisele alamtüübile. Lisaks tekib tabelisse palju mittekohustuslikke veerge (kus on lubatud NULLid) ning tabel võib paisuda väga suureks. (Fowler 2002, 279) Kõik alamtüüpide atribuutidele vastavad veerud peavad olema mittekohustuslikud.

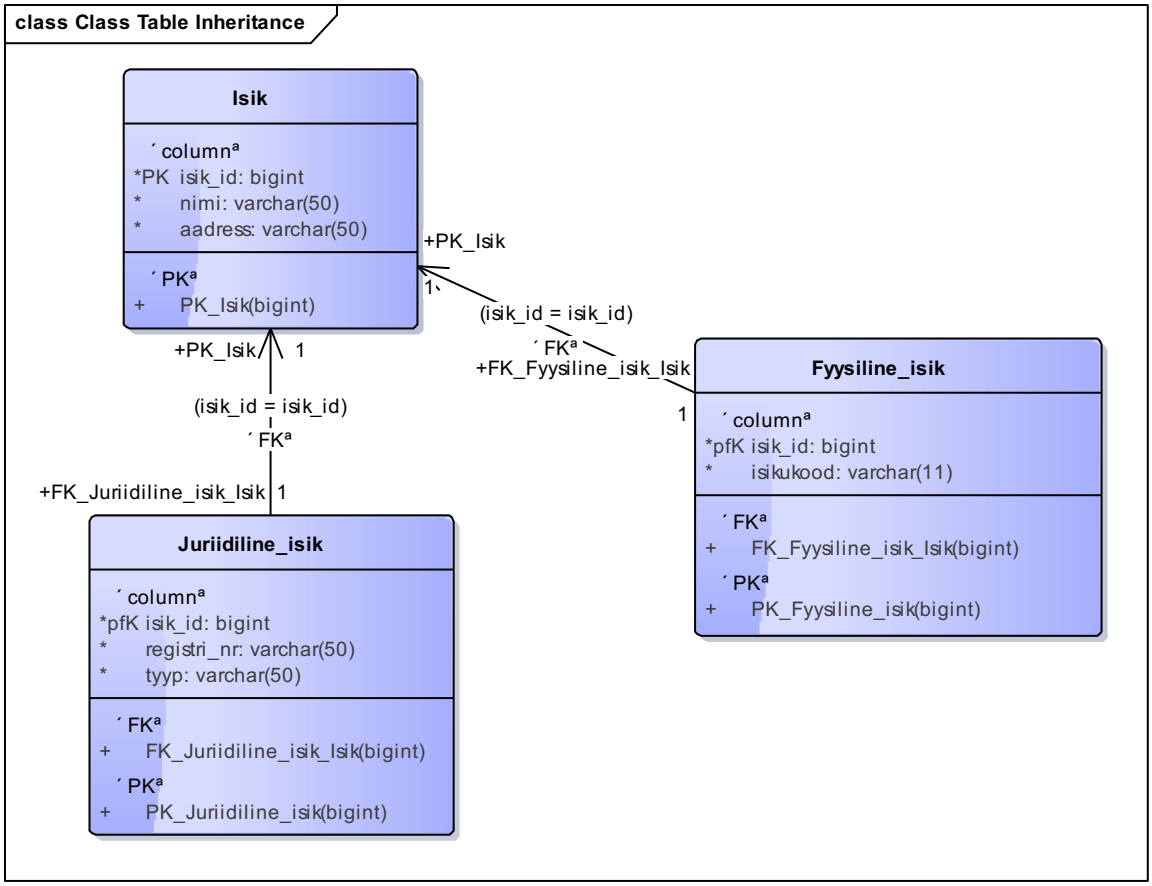


Joonis 4. ***Single Table Inheritance***

- Klassi tabeli pärimine (*Class Table Inheritance*)

  Iga tüübi kohta luuakse eraldi tabel. Alamtüübi tabeli primaarvõti viitab vanema primaarvõtmele ja on seega ühtlasi välisvõti. (vt. Joonis 5.) Mustri tugevuseks peetakse

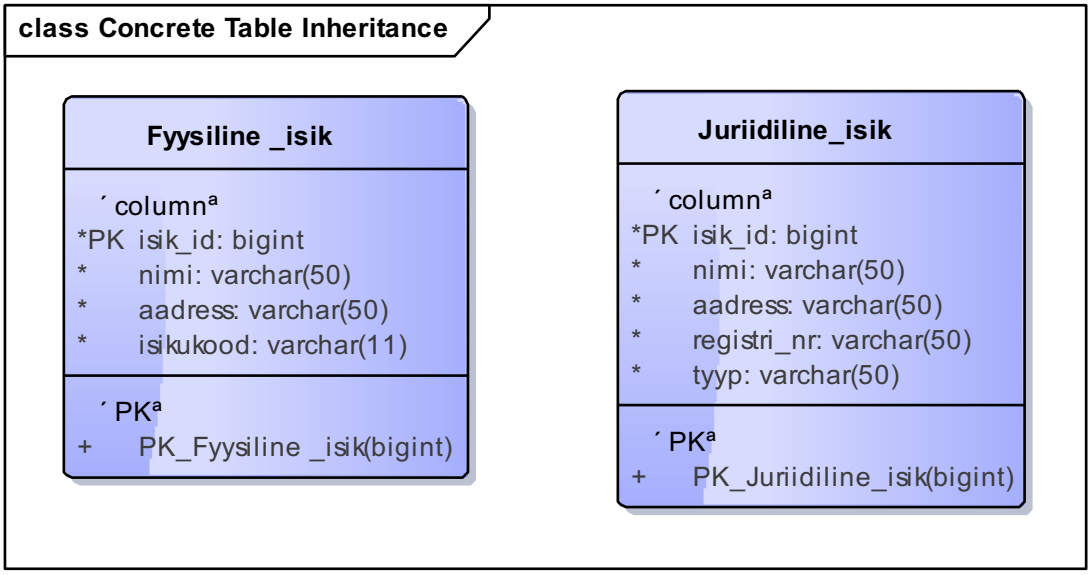
seda, et tabelite ja veergude tähendused ning domeenimudeli ja andmebaasi loogilise disaini seosed on arusaadavad. Samas on andmete lugemiseks vaja teha mitu ühendamisoperatsiooni ja hierarhia muutmine ei ole lihtne. (Fowler 2002, 286)



Joonis 5. ***Class Table Inheritance***

- Konkreetse tabeli pärimine (*Concrete Table Inheritance*)

Iga alamtüübi kohta luuakse eraldi tabel, mis hoiab lisaks tüübispetsiifilistele atribuutidele ka kõiki vanema atribuute. (vt. Joonis 6.) Muster on hea, sest iga tabel on iseseisev ja andmete lugemiseks ei pea tabeleid ühendama. Nõrkusteks peetakse seda, et ridade unikaalsust on keeruline tagada. Samuti tuleb muutuste korral ülatüübi struktuuris, muuta kõiki alamtüüpidele vastavaid tabeleid. (Fowler 2002, 294 - 296) Lisaks tekitab selline tabel andmete dubleerimist, kui ülatüübi eksemplar võib kuuluda mitmesse alamtüüpi.



Joonis 6. ***Concrete Table Inheritance***

1.4.6 Polymorphic Associations

Polymorphic Associations antimuster tekib, kui tabeli veerg tahetakse panna viitama rohkem kui ühele tabelile. SQL−andmebaasides ei ole sellisel juhul võimalik deklareerida välisvõtme kitsendust, mistõttu nendest loobutakse. Viidete terviklikkuse kontrollimiseks tuleks programmeerida andmebaasis trigereid või realiseerida see rakenduse koodis. Päringute tegemiseks tuleb kasutada LEFT OUTER ühendamisoperatsiooni kõikide potentsiaalsete tabelitega, mis võib osutuda andmebaasisüsteemi jaoks väga kulukaks. (Karwin 2010, 78 - 79) Antimustri kasutamise asemel võib:

- iga seose kohta luua vahetabeli, jõustada välisvõtmed ja sellega ka viidete terviklikkuse,
- mitme viidatavate tabelite asemel luua üks tabel (nt kasutades Klassi tabeli pärimise mustrit, vt. Joonis 5.), millele saab välisvõtit kasutades viidata. (*Ibid*, 83 - 88)

1.4.7 Multicolumn Attributes

Mitme veeru kasutamine muudab päringud keerukamaks, sest tuleb kontrollida kõiki veergusid, kus otsitav väärtus olla võib. Ka andmete unikaalsuse tagamine muutub keerukaks. Kui peaks juhtuma, et olemasolevatest veergudest ei piisa andmete hoidmiseks, siis peab hakkama tabelit ümber tegema ja muutma andmebaasi kasutava rakenduse koodi. (*Ibid*, 90 - 93)

Selle asemel, et atribuudi mitut erinevat väärtust hoida mitmes veerus, tuleks nende väärtuste hoidmiseks luua eraldi tabel. (Karwin 2010, 95)

1.4.8 Metadata Tribbles

Antimustri heaks näiteks on erinevate tabelite tekitamine vastavalt aastale, nt Tulud_2010, Tulud_2011, Tulud_2012 jne. Selline lahendus on kohmakas ja muudab edasise töö keeruliseks. Tuleb rakendada reeglid, et andmed sisestataks õigesse tabelisse. Kui andmed muutuvad, siis need muutused tuleb sünkroniseerida üle kõikide tabelite. Päringute tegemine, et saada andmed, mis asuvad erinevates tabelites muutub, tabelite arvu suurenedes, aeglaseks. Selliste päringute keerukus ja suure tõenäosusega ka täitmiseks kuluv aeg kasvab. Ühtlasi on raske viidata tabelile, mis on jaotatud mitmeks erinevaks tabeliks. (*Ibid*, 98 - 102)

Mahukate andmete haldamiseks on olemas palju paremaid viise. Üheks nendest on tabeli horisontaalne eraldamine (kasutaja eest peidetud sektsioonideks jagamine), mille teeb kasutaja eest ära andmebaasisüsteem. Sellist meedet kasutades, saab kasutaja töötada tabeliga nagu see oleks üks tabel, kuid andmebaasis on sisemiselt väiksemateks osadeks jaotatud. Mõned andmebaasisüsteemid nagu Oracle Database toetavad seda valmis kujul, mõned andmebaasisüsteemid nagu PostgreSQL saab seda reeglite abil programmeerida ning mõnes andmebaasisüsteemis nagu MS Access pole see võimalik. Teiseks meetmeks on vertikaalne eraldamine. Sel juhul tükeldab kasutaja tabeli kaheks – ühes on veerud, mille andmetüüpidesse kuuluvad väärtused on kindla suurusega, teises veerud, mille andmetüüpidesse kuuluvad väärtused ei ole kindla suurusega, praktikas nt VARCHAR, BLOB ja TEXT. Nii saab päringuid kiirendada, eraldades vähem kasutatavad andmed nendest, mida tõenäoliselt läheb rohkem vaja. (*Ibid*, 105 - 106)

1.4.9 Rounding Errors

Kuna enamikes programmeerimiskeeltes on olemas float andmetüüp, võib arendaja harjumusest ka SQLis kasutada FLOAT andmetüüpi. FLOAT andmetüüp kasutab IEEE 754 standardit, mis kodeerib reaalarvud kahendsüsteemis. Mõned arvud vajavad lõpmatu arv komakohti, FLOAT andmetüüp ei toeta seda, nii et ta valib salvestamiseks lähima kahendarvu. Kümnendarv 59.95 salvestatakse FLOAT andmetüüpi kasutades kahendaarvuna, mida teisendades saame kümnendarvu 59.95000076293945. SQLis muudab see arvude võrdlemise keeruliseks ja aritmeetilisis tehteid tehes võivad ümmardamisest tekkinud vead muutuda päris suureks. (*Ibid*, 112 - 115)

FLOAT andmetüübi asemel tuleks kasutada NUMERIC või DECIMAL andmetüüpi. NUMERIC ja DECIMAL andmetüüpidel ei ole vahet. Mõlemad salvestavad ratsionaalarve ümmardamata ning mõlemad lubavad täpsustada numbrite koguarvu ja numbrite arvu paremal pool koma. NUMERIC ja DECIMAL andmetüübid võimaldavad ohutut arvude võrdlemist. (Karwin 2010, 117)

1.4.10 31 Flavours

31 Flavours antimuster tekib, kui arendaja piirab veeru väärtused, kasutades CHECK kitsendust, ENUM andmetüüpi, triggereid. Selline lahendus piirab klassifikaatorite väärtuste hulka, kuid muudab keeruliseks uute väärtuste lisamise, kõikide väärtuste kasutamise rakenduses, mingite väärtuste kasutamise lõpetamise. Lisaks on sellise lahenduse teisaldavatus halb, sest iga andmebaasisüsteemi võimalused selliste piirangute realiseerimiseks varieeruvad mingil määral. (*Ibid*, 120 - 123)

Palju parem lahendus veeru väärtuste piiramiseks on luua eraldi klassifikaatorite tabel ning tekitada sellele viitav välisvõti. Klassifikaatorite tabel võimaldab väärtuste lugemist, muutmist, kasutamise lõpetamist ja teisaldamist. (*Ibid*, 125)

1.4.11 Phantom Files

Antud antimustri sisuks on, et suuri faile, nt pilte, hoitakse väljaspool andmebaasi (nt failisüsteemi kataloogis) ja andmebaasis hoitakse stringi kujul teekonda nendeni. Selline lahendus võib tekitada mitmeid probleeme. Nimelt, kui andmebaasis kustutatakse rida, mis viitab suurele failile, siis see fail ei kustu automaatselt ja failisüsteemi võib jääda suur kogus faile, mida keegi ei kasuta, ent mis võtavad palju ruumi. Samuti ei allu väljaspool andmebaasi asuvad failid transaktsioonide halduse (tehingutöötluse) mehhanismile, st andmebaasi transaktsiooni tühistamine ei võta tagasi väljaspool andmebaasi oleva faili kustutamise operatsiooni. Andmebaasist varukoopiat tehes, tuleb mõelda väljaspool andmebaasi paiknevatele ressurssidele, sest automaatselt neid varukoopiatesse ei lisata. Ühtlasi on raske kontrollida, kas andmebaasis salvestatud teekond failideni ikka on õige. (*Ibid*, 128 - 131)

Antimustri tekitatavaid probleeme saab lahendada hoides faile andmebaasis nt BLOB tüüpi veerus (või mõnda muud sellist tüüpi veerus, mida andmebaasisüsteem taoliseks puhuks pakub). Nii on võimalik käsitleda neid faile ülejäänud andmebaasi kontrolli all olevate andmetega samaväärselt ning järgida nende puhul viidete terviklikkuse reeglit, transaktsioonide atomaarsust, terviklikkust, isoleeritust ja püsivust. Failid lisatakse

automaatselt andmebaasi varukoopiatesse. BLOB andmetüüpide kasutamise miinuseks on, et andmebaasi maht võib minna päris suureks. (Karwin 2010, 133)

1.4.12 Index Shotgun

Andmebaasi indeks on andmestruktuur, mis kiirendab andmete lugemisoperatsioone tabelitest (Database index 17.05.2015). Kui tekitada indekseid mõtlemata läbi, kuidas need aitavad, võib tekkida kolme tüüpi probleeme:

- Tekitatakse liiga vähe indekseid või üldse mitte.

  Indeksid võtavad andmemahust umbes 10 – 20% (Eessaar 2014a). Mitmed arendajad arvavad, et indeksite tekitatav lisaandmemaht on raiskamine, seega loobuvad nad üleüldse indeksite tegemisest. (*Ibid*, 137)

- Tekitatakse liiga palju indekseid.

  SQL−andmebaasisüsteemid loovad primaarvõtmetele tavaliselt automaatselt indeksid, seega pole arendajal mõtet seda teha. Veergudele, kus on pikad stringid ei ole samuti mõtet luua indekseid, sest nende alusel tavaliselt päringuid ei tehta. Võib juhtuda, et arendaja loob indeksid kõikidele veergudele ja kõikidele veergude kombinatsioonidele, sest ta ei tea, mida tuleb indeksite loomisel arvestada. Sel juhul raiskab ta lihtsalt andmebaasis ruumi. (*Ibid*, 138) Lisaks salvestusruumi raiskamisele raisatakse ka andmebaasisüsteemi ressursse, sest indeksid tuleb hoida indekseeritavate tabelitega kooskõlas.

- Päringute tegemine viisil, mida ükski indeks ei abista.

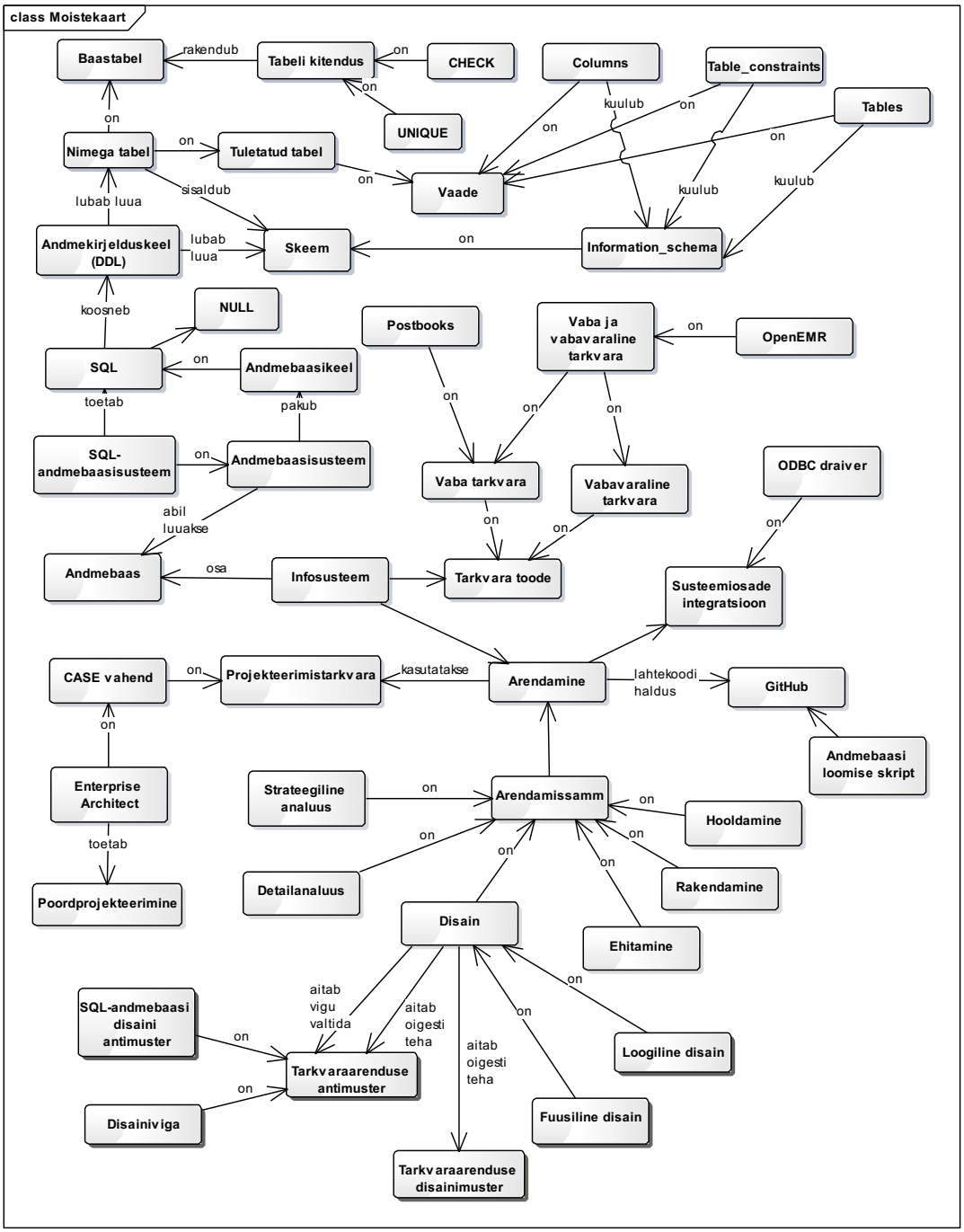
  Kui luua tabelis indeks, mis katab inimese perekonnanime ja eesnime, aga andmeid otsida eesnime järgi, siis ei ole sellest indeksist abi. (*Ibid*, 139) Samuti ei kasutata tavalist indeksit, siis kui päringu tingimuses rakendatakse indekseeritud veerule mõnd funktsiooni või operaatorit. Selliseks puhuks oleks vaja funktsioonil põhinevat indeksit.

Enne indeksite loomist tuleks analüüsida, millised on andmebaasis kõige enam kasutatavad päringud ja siis hakata vaatama, kas indeksite loomine parandab nende päringute täitmise kiirust. Andmebaasisüsteemides on olemas tööriistad, mis toetavad sellist testimist. See tähendab, et arendaja ei pea indekseid looma pelgalt sisetunde järgi. Üldiselt tasuks pöörata tähelepanu veergudele, mida loetakse kõige rohkem, nt välisvõtmed. Kui vahetabeli

primaarvõti koosneb mitmest veerust, siis tasuks võtme teisele veerule teha eraldi indeks, sest see veerg pole võtme alusel automaatselt loodud liitindeksis kõige esimesel kohal. (Karwin 2010, 141 - 146)

2. Eksperiment

Selles peatükis kirjeldatakse töö praktilist osa, mis seisnes olemasolevate andmebaaside uurimises. Joonis 7 kirjeldab eksperimendi läbiviimisega seotud mõisteid mõistekaartina, mis on ülesjoonistatud UML klassidiagrammina.



Joonis 7. Lõputöö teema põhimõistete kaart

2.1 Eksperimendi kirjeldus

Eksperimendi aluseks võtsin xTuple rakenduse Postbooks ning vaba ja avatud lähtekoodiga OpenEMR andmebaasid. Postbooks andmebaas on tehtud kasutades PostgreSQL andmebaasisüsteemi. OpenEMR-i andmebaas on realiseeritud kasutades MySQL andmebaasisüsteemi. Postbooks on avatud lähtekoodiga aruandlus ja ERP süsteem, mis on suunatud väike- ja keskmise suurusega ettevõtjatele (Postbooks 19.05.2015). OpenEMR on vaba ja avatud lähtekoodiga arstipraksise haldustarkvara, mis toetab ka EMR ehk digiloo funktsionaalsust (OpenEMR 19.05.2015). Mõlema süsteemi andmebaasi genereerimise lähtekood on üles laetud GitHubi. (OpenEMR source code 19.05.2015, xTuple source code 19.05.2015)

Andmebaaside lähtekood võeti GitHubist ja nende abil loodi andmebaasid serveril. Andmebaaside disainist parema ülevaate saamiseks kasutati modelleerimistarkvara Enterprise Architect pöörprojekteerimise funktsionaalust. Enterprise Architect ühendub ODBC draiveri abil serveriga ja andmebaasi skeem laetakse UML mudelisse. Õpetus võeti Enterprise Architect dokumentatsioonist. (Sparx Systems Documentation 24.05.2015) Andmebaasi tabelite suure hulga tõttu ei olnud mõtet teha ühte suurt diagrammi. Seega loodi diagramme vastavalt vajadusele, selleks, et mõista tabelite ja ridade vahelisi seoseid.

Uurin, mitu vaadet loodi andmebaasis. Vaated võimaldavad andmebaasis luua virtuaalse andmete kihi, mis kapseldab andmebaasi ja serveerib erinevat tüüpi kasutajatele just neile huvipakkuvad andmed just neile vajalikus formaadis (Burns 2010). Uurin, kui palju lubatakse tabelites NULLe, mis võivad viia loogiliselt ebakorrektsete päringute tulemusteni. Uurin, kui palju luuakse PostgreSQL andmebaasis CHECK kitsendusi, mis aitavad välistada ebakorrektsete andmete andmebaasi sattumist. MySQL andmebaasis ei olnud põhjust CHECK kitsendusi otsida, sest MySQL (5.7) ei jõusta CHECK kitsendusi (MySQL 5.7 Reference Manual 24.05.2015). Lisaks uurin, kui mitu UNIQUE kitsendust on kokku andmebaasis ja kui mitmel protsendil tabelitest on vähemalt üks UNIQUE kitsendus. On levinud viga määrata tabelile küll primaarvõti, kuid unustada võimalus, et tabelis on mitu võtit. Primaarvõtmeks mitte valitud võtmeid nimetatakse alternatiivvõtmeteks ja need tuleks UNIQUE kitsenduste abil andmebaasis jõustada.

Eksperimendi käigus uurin andmebaase lähtudes Michael Blaha andmebaasi disaini detailidest ja antimustritest (vt. Tabel 1) ja Bill Karwini kõikidest antimustritest välja arvatud

Keyless Entry, *Multicolumn Attributes* ning *Index Shotgun*, sest need antimusterid kattuvad vastavalt Blaha B2, B6 ja B5 antimustriga.

Andmebaasi tabelite ja veergude arv, ilma primaarvõtmeteta tabelite arv leiti kasutades päringuid süsteemikataloogi põhjal loodud vaadetest:

- *information_schema.columns,*
- *information_schema.tables,*
- *information_schema.table_constraints*

Päringud on esitatud lisas Lisa 1.

Andmebaaside tabelite normaliseeritust hindasin kaudselt, üleliigsete veergude olemasolu järgi ja uurides sõltuvusi veergude vahel.

2.2 Eksperimendi tulemused

xTuple Postbooks andmebaasis on 272 tabelit ja 31 vaadet. OpenEMR andmebaasis on 159 tabelit, vaateid ei loodud.

Postbooks andmebaasis on deklareeritud 121 UNIQUE kitsendust. 114 (42%) tabelis on deklareeritud vähemalt üks UNIQUE kitsendust. Nende kitsenduste sekka ei loetud PostgreSQL andmebaasisüsteemi automaatselt loodavad primaarvõtme veeru indeksid. OpenEMR andmebaasis loodi 24 UNIQUE kitsendust 24 (15%) tabelile.

Postbooks andmebaasis loodi 268 tabelile vähemalt üks CHECK kitsendus ehk peaaegu igale tabelile. 268. tabelis oli vähemalt üks kitsendus, mis välistas NULLide sisestamise. Selliseid kitsendusi, mis välistasid NULLide sisestamise oli kokku 1171 ehk iga tabeli kohta keskmiselt 4,4. Loodi 145 CHECK kitsendust 114. tabelile, mis pidid tagama, et ei oleks võimalik sisestada tühikutest koosnevat stringi või et sisestataks õige klassifikaator.

OpenEMR andmebaasis on 1859 veergu, millest 863 ehk 46% veergudest ei ole võimalik sisestada NULLi. Postbooks andmebaasis on 3817 veergu, millest 1171 ehk 31% veergudest ei ole võimalik sisestada NULLi.

Tabelid 3–7 annavad ülevaate Michael Blaha ja Bill Karwini antimustrite esinemisest Postbooks ja OpenEMR andmebaasides.

Tabel 3. Blaha disainivigade B1 - B5 tulemused

Nimi	Primaarvõtme deklareerimine	Välisvõtmete deklareerimine	NULLi lubamine primaar-võtme väljas	Andmetüüpide järjekindlus	Indeksid
xTuple Postbooks	Jah, 97% tabelitel. Puuduvate primaarvõtmetega tabelid on abitabelid	Välisvõtmed on üldiselt deklareeritud, samas on palju kohti, kus oleks võinud kasutada välisvõtit. 272 tabeli kohta on viiel korral kasutatud ON UPDATE CASCADE kompenseerivat tegevust. ON DELETE CASCADE kompenseerivat tegevust on kasutatud palju rohkem.	Ei	Enamus stringiväärtusi hoitakse andmetüüpi TEXT omavates veergudes, ujukomakohtade andmetüübi täpsus varieerub. Kuupäevi hoitakse date/timestamp andmetüübiga veergudes.	Kõikidel primaarvõtmetel on indeksid. Välisvõtmetel on vähe indekseid. Indekseid luuakse valdavalt siis, kui tahetakse tagada väärtuse unikaalsus.

OpenEMR	79% tabelitest on primaarvõti olemas.	Välisvõtmed puuduvad.	Ei	Teatud järjepidevus on nähtav, aga on mitmeid juhtumeid, kus stringipikkused varieeruvad. Primaarvõtme andmetüüp varieerub INT ja BIGINT vahel. Kuupäevi hoitakse date/timestamp andmetüübiga veergudes.	Osadele primaavõtme ja kandiaatvõtme väljadele on loodud indeksid.

Tabel 4. Blaha antimustrite B6 – B9 tulemused

Nimi	Paralleelsed veerud	Tabelite denormaliseeritus	Veergude arv tabelis	Klassifikaatorid
xTuple Postbooks	Suurtes denormaliseeritud tabelites esineb väga palju.	Esinevad suured denormaliseeritud tabelid. Suurtes tabelites esineb ülekanduvaid sõltuvusi.	Keskmiselt on (baas)tabelites ja vaadetes 13 veergu. Maksimaalne veergude arv (baas)tabelis on 79, vaates 93.	Üksikute klassifikaatorite jaoks on tehtud omaette tabel, nt. rahaühik, aga enamasti on need stringi vabas vormis tekstina samas tabelis.
OpenEMR	Mõnedes tabelites leidub.	Mitmed tabelid on tugevalt denormaliseeritud. Suurtes tabelites esineb ülekanduvaid sõltuvusi.	Keskmiselt on (baas)tabelis 12 veergu. Maksimaalne veergude arv (baas)tabelis on 142.	Hoitakse üldiselt stringide kujul samas tabelis.

Tabel 5. Karwini antimustrite tulemused I osa

Nimi	*Jaywalking*	*Naive Trees*	*ID Required*
xTuple Postbooks	Antimustri kasutamine puudub.	Hierarhilised andmed puuduvad	Peaaegu igas tabelis genereerib andmebaasisüsteem primaarvõtme väärtuse. Sisulisi kandidaatvõtmeid on loodud, aga neid ei kasutata primaarvõtmetena.
OpenEMR	Antimustri kasutamist ei leidu.	Andmebaasis on üksikud tabelid, mille puhul võib arvata, et viidatakse iseendale, aga välisvõtmete puudumise tõttu ei ole võimalik kindel olla.	Enamik tabelites on kasutatud surrogaatvõtit primaarvõtmena. Vahetabelites kasutatakse liitvõtmeid.

Tabel 6. Karwini antimustrite tulemused II osa

Nimi	*Entity-Attribute-Value*	*Polymorphic Associations*	*Metadata Tribbles*
xTuple Postbooks	Antimustri kasutamine puudub.	Sarnased objektid on iseseisvad tabelid. Pärimist ei ole üritatud realiseerida.	Antimustrit ei esine.
OpenEMR	Antimustri esinemist ei toimu.	Sarnased objektid on iseseisvad tabelid. Pärimist ei ole üritatud realiseerida.	Mitmes tabelis on veerud, mida eristatakse aastate järgi.

Tabel 7. Karwini antimustrite tulemused III osa

Nimi	*Rounding Errors*	*31 Flavours*	*Phantom Files*
xTuple Postbooks	Antimustrit ei esine.	Ühetähelised sümbolid piiratakse 14 tabelis piiranguga, kontrollides, et väärtus oleks massiivis.	Antimustrit ei esine. Viiteid piltidele hoitakse TEXT andmetüüpi veerus.
OpenEMR	FLOAT andmetüüpi kasutati 7 tabelis ehk mitte eriti tihti.	10 tabelis piiratakse veeru väärtusi, kasutades ENUM andmetüüpi. Mõnede klassifikaatorite väärtuste võimalikkus selgub kommentaaridest.	Suuri faile ei hoita antud andmebaasis, vaid kettal, millele viitavat teekonda ja teekonna sügavust salvestatakse, või välises andmebaasis.

Toetudes tabelitele 3–7 võib väita, et xTuple Postbooks andmebaas on disainitud palju paremini kui OpenEMR-i andmebaas. Postbooks andmebaasis on korralikult deklareeritud primaarvõtmed ja enamik välisvõtmeid. Andmetüüpide kasutus on suuresti järjekindel ja ka indeksite kasutamisest võib välja lugeda, et arendajad on töötanud välja strateegia.

OpenEMR andmebaasi puhul on märgata, et seda on arendanud lõdvalt seotud arendajad. Seda väljendavad ilmekalt sarnaste tähendustega veergude erineva väljapikkusega andmetüübid või lausa erinevad andmetüübid. Samuti erineb veergude nimetamise stiil. Näiteks osad primaarvõtme veerud kasutavad nime jaoks stiili *tabeli_nimi_id*, osad nimetatakse lihtsalt *id*. Postbooks andmebaas oli veergu andmetüüpide ja nimetamise osas ühtlasem.

Postbooks andmebaasi kitsaskohad ilmenvad selles, et paljud tabelid ei ole piisavalt kõrgel normaalkujul. Selle tõttu esineb andmete dubleerimist ja kannatab andmebaasi loetavus.

xTuple arendajad võiksid pöörata tähelepanu andmebaasi suuremale normaliseerimisele, klassifikaatorite tabelite loomisele, mis lihtsustaks pikas perspektiivis andmebaasi hooldamist, ja sisuliste primaarvõtmete suuremale kasutamisele.

OpenEMR andmebaasis esinevad puudused primaarvõtmete deklareerimises, välisvõtmeid ei ole üldse deklareeritud. Välisvõtme kitsenduste puudumine muudab peaaegu võimatuks viidete terviklikkuse reegli järgimist, mis on hästi toimiva andmebaasi üks nurgakivi. Lisaks muudab see andmebaasi analüüsimise tunduvalt raskemaks. Kuigi tagantjärgi on välisvõtmete lisamine keeruline, tuleks sellega tegeleda, et andmebaas oleks ka tulevikus hallatav. Ainuüksi USA-s teenindatakse OpenEMR-i tarkvara abil ligi 30 miljonit patsienti. Samuti tasuks arendajatel vaeva näha andmebaasi normaliseerimise, klassifikaatorite tabelite loomisega ja andmetüüpide ühtlustamisega.

Siit võib järeldada, et kuigi ühe andmebaasi disain on õnnestunud paremini kui teise oma, siis mõlemas ilmnevad samad probleemid – liigne denormaliseerimine, sarnaste väärtuste hoidmine paralleelsetes veergudes, ebapiisav välisvõtmete deklareerimine ning klassifikaatorite tabelite puudumine. Tulemused on võrdlemisi sarnased nendele, mida järeldas Michael Blaha peaaegu 15 aastat tagasi.

3. Kokkuvõte

Käesoleva töö peamisteks eesmärkideks oli teada saada, kas andmebaaside disaini antimustrid, mida Michael Blaha peaaegu 15 aastat tagasi kaardistas, korduvad ka tänapäeva vaba tarkvara poolt kasutatavates SQL−andmebaasides, ja millised antimustrid neis veel leiduvad. Töö tegemine eeldas põhjalikku tutvust antimustritega.

Töö käigus kaardistati kirjanduse põhjal 12 antimustrit ja mitmed andmebaasi disaini detailid, millele tuleks tähelepanu pöörata. Antimustrite sisalduvuse osas analüüsiti pöördprojekteerimist kasutades kahte andmebaasi, kumbki erinevast valdkonnast. Analüüsi tulemusel jõuti järeldusele, et kuigi kaks väljavalitud andmebaasi erinevad üksteisest sellepoolest, kui suure sagedusega antimustrid neis esinevad, siis sügavamad probleemid on neil sarnased. Mõlema andmebaasi disain kannatab liigse denormaliseerimise, paralleelsete veergude, välisvõtmete ebapiisava deklareerimise ja klassifikaatorite tabelite puudumise tõttu. Võib väita, et alates Blaha uurimuste avaldamisest on olukord mingil määral paranenud, aga mitte palju. Põhjapanevate üldistuste tegemiseks oli uuritavate andmebaaside hulk siiski liiga väike. Ometi võib tõdeda, et ERP (ettevõtte ressursside haldamise) andmebaas oli parema disainiga kui EMR (arstipraksise halduse süsteemi andmebaas. Põhjus võib seisneda selles, et ERP süsteemid on oluliselt populaarsemad ja konkurents turul on suurem. Üldiselt tulemused saavutati, ent need oleksid olnud märgatavalt paremad, kui analüüsitavate andmebaaside hulk oleks olnud suurem.

Tulevikus tasuks antud teemat käsitleda, analüüsides suuremat hulka andmebaase. Ühtlasi tasuks rohkem uurida EMR süsteeme, et aru saada, kas siin töös kajastatud tulemused on EMR andmebaaside korral süstemaatilised. Tuleviku töö võiks üritada taoliste andmebaaside normaliseeritusse astet veergude sõltuvuste alusel põhjalikumalt analüüsida ning ühtlasi hinnata sellises andmebaasis olevat andmete liiasust üldiselt – on andmete liiasusi, mida tabelite täielik normaliseerimine ei eemalda.

Autor tänab töö juhendajat, Erki Eessaart, abi ja mõistva suhtumise eest.

Summary

The goal of this thesis was to learn as to whether the antipatterns that Michael Blaha reported almost 15 years ago still exist in today's SQL databases, which are used by free software as well as which other antipatterns can be found in these databases.

Twelve antipatterns and multiple other database design flaws were studied. Two databases were analysed by using reverse engineering to find out if antipatterns exist in them. As a result of the analysis it was concluded that even though not all antipatterns can be found in both of them and the frequency with which they occur may wary, both databases suffer from the same underlying problems. These are heavy denormalization, the use of parallel columns, underuse of foreign key enforcement, and the lack of enumeration tables (reference data tables). It can be said that since Michael Blaha wrote his study, software development practices have improved but not by much. However, for wide-ranging conclusions the number of analysed databases was too small. It was also concluded that the design of the analysed ERP (Enterprise Resource Planning) database was better than that of the analysed database of healthcare management software. This may be because ERP systems are more popular and competition is tougher. Results might have been more conclusive, had the author used more databases to analyse.

It might be wise to conduct more studies about antipatterns used in healthcare systems to find out if mistakes in database design are systematic or not. Future studies should analyse database normalization more thoroughly based on dependencies of columns. It should also be evaluated how much redundant data is in these databases since normalization process does not eliminate all data redundancy.

The author wishes to extend his gratitude to Erki Eessaar, the supervisor of this thesis.

Kasutatud kirjandus

1. Blaha, M. (2001a). A Retrospective on Industrial Database Reverse Engineering Projects – Part 1. - *Eighth Working Conference On Reverse Engineering 2-5th October 2001, Stuttgart, Germany*, 136-146

2. Blaha, M. (2001b). A Retrospective on Industrial Database Reverse Engineering Projects – Part 2. - *Eighth Working Conference On Reverse Engineering 2-5th October 2001, Stuttgart, Germany*, 147-153

3. Brown, W. J., Malveau, R. C., McCormick III, H. W., Mowbray, T. J. (1998). AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. New York: Wiley

4. Burns, L. (2011). Building the agile database: how to build a successful application using agile without sacrificing data management. Technics Publications. 276 p.

5. Database index – Wikipedia, the free encyclopedia. [WWW] http://en.wikipedia.org/wiki/ Database_index (17.05.2015)

6. Design Pattern – Wikipedia, the free encyclopedia. [WWW] http://en.wikipedia.org/wiki/Design_pattern (03.05.2015)

7. Eessaar, E. (2008). Andmebaaside projekteerimine. Tallinn: Tallinna Tehnikaülikooli Kirjastus.

8. Eessaar, E. (2014a). Andmebaasi disain, ehitamine ja andmesiire. Tallinna Tehnikaülikooli õppeaine "Andmebaasid II" teema 4 konspekt.

9. Eessaar, E. (2014b). Mõistekaardid. Tallinna Tehnikaülikooli õppeaine "Andmebaasid I" konspekt.

10. Eessaar, E. (2014c). Mõistekaardid. Tallinna Tehnikaülikooli õppeaine "Andmebaasid II" konspekt.

11. Electronic health record – Wikipedia, the free encyclopedia. [WWW] http://en.wikipedia.org/wiki/Electronic_health_record (19.05.2015)

12. Enterprise resource planning – Wikipedia, the free encyclopedia. [WWW] http://en.wikipedia.org/wiki/ Enterprise_resource_planning (19.05.2015)

13. Fowler, M. (2002). Patterns of Enterprise Application Architecture. USA: Addison-Wesley

14. Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. USA: Addison-Wesley

15. Karwin, B. (2010). SQL Antipatterns: Avoiding the Pitfalls of Database Programming. Raleigh (N.C.), Dallas (Tex.): The Pragmatic Bookshelf.

16. MySQL 5.7 Reference Manual:13 SQL Statement Syntax:13.1 Data Definition Statements:13.1.14 CREATE TABLE Syntax (2015)/Oracle Corporation. [WWW] https://dev.mysql.com/doc/refman/5.7/en/create-table.html (24.05.2015)

17. OpenEMR – Wikipedia, the free encyclopedia. [WWW] http://en.wikipedia.org/wiki/OpenEMR (19.05.2015)

18. OpenEMR source code – Github. [WWW] https://github.com/openemr/openemr/blob/master/sql/database.sql (19.05.2015)

19. Postbooks – Wikipedia, the free encyclopedia. [WWW] http://en.wikipedia.org/wiki/Postbooks (19.05.2015)

20. Rõzova, N. (2011). Disainimustrid üldistusseoste realiseerimiseks SQL-andmebaasides : bakalaureusetöö. Tallinna Tehnikaülikool, Tallinn.

21. Rubens, P. 10 Open Source ERP Options. [WWW] http://www.enterpriseappstoday.com/erp/10-open-source-erp-options.html (23.05.2015)

22. Sparx Systems Documentation:Index:Database Engineering:Import Database Schema. (2011)/Sparx Systems [WWW] http://www.sparxsystems.com/enterprise_architect_user_guide/9.0/database_engineering/importdatabaseschemafromod.html (24.05.2015).

23. Struktuurpäringukeel – Vikipeedia, vaba entsüklopeedia. [WWW] http://et.wikipedia.org/wiki/Struktuurpäringukeel (10.05.2015)

24. xTuple source code – Github. [WWW] https://github.com/xtuple/xtuple/blob/4_9_x/foundation-database/440_schema.sql (17.05.2015)

Lisa 1. SQL laused

- SQL lause tabelite ja vaadete arvu leidmiseks:

```
SELECT count(*), table_type FROM (SELECT table_catalog,
table_schema, table_name, table_type
    FROM information_schema.tables where table_schema = 'public' and
table_type = 'BASE TABLE') AS a GROUP BY table_type UNION ALL
SELECT count(*), table_type FROM (SELECT table_catalog,
table_schema, table_name, table_type
    FROM information_schema.tables where table_schema = 'public' and
table_type = 'VIEW') AS b GROUP BY table_type;
```

- SQL lause tabelite ja vaadete veergude aru leidmiseks:

```
SELECT table_schema, table_name, count(*)
  FROM information_schema.columns WHERE table_schema = 'public'
  GROUP BY table_schema, table_name ORDER BY table_name;
```

- SQL lause Postbooks andmebaasi tabelite leidmiseks, milles puuduvad primaarvõtmed:

```
SELECT table_catalog, table_schema, table_name
    FROM information_schema.tables
    WHERE (table_catalog, table_schema, table_name) NOT IN
          (SELECT table_catalog, table_schema, table_name
               FROM information_schema.table_constraints
               WHERE constraint_type = 'PRIMARY KEY')
      AND table_schema NOT IN ('information_schema', 'pg_catalog')
AND table_type = 'BASE TABLE';
```

- SQL lause OpenEMR andmebaasi tabelite leidmiseks, milles puuduvad primaarvõtmed (PostgreSQL andmebaasi süsteemikataloogi jaoks):

```
SELECT table_catalog, table_schema, table_name
    FROM information_schema.tables
    WHERE (table_catalog, table_schema, table_name) NOT IN
          (SELECT table_catalog, table_schema, table_name
               FROM information_schema.table_constraints
               WHERE constraint_type = 'PRIMARY KEY')
      AND table_schema IN ('openerm') AND table_type = 'BASE TABLE';
```

- SQL lause, mille abil saab leida tabelid, milles on unikaalsuse kitsendus:

```
SELECT table_catalog, table_schema, table_name
    FROM information_schema.tables
    WHERE (table_catalog, table_schema, table_name) IN
          (SELECT table_catalog, table_schema, table_name
FROM information_schema.table_constraints
WHERE table_schema='public' AND constraint_type='UNIQUE')
      AND table_schema NOT IN ('information_schema', 'pg_catalog')
AND table_type = 'BASE TABLE';
```

- SQL lause, mille abil saab leida tabelid, milles on CHECK kitsendus:

```
SELECT table_catalog, table_schema, table_name
    FROM information_schema.tables
    WHERE (table_catalog, table_schema, table_name) IN
          (SELECT table_catalog, table_schema, table_name
FROM information_schema.table_constraints
WHERE table_schema='public' AND constraint_type='CHECK')
      AND table_schema NOT IN ('information_schema', 'pg_catalog')
AND table_type = 'BASE TABLE';
```

- SQL lause, mille abil saab leida Postbooks andmebaasi tabelid, milles on NOT NULL kitsendus (PostgreSQL andmebaasi süsteemikataloogi jaoks):

```
SELECT table_catalog, table_schema, table_name
    FROM information_schema.tables
    WHERE (table_catalog, table_schema, table_name) IN
          (select table_catalog, table_schema, table_name from
information_schema.table_constraints where table_schema='public' and
constraint_type='CHECK' and constraint_name LIKE '%not_null%')
      AND table_schema NOT IN ('information_schema', 'pg_catalog')
AND table_type = 'BASE TABLE';
```