TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Informaatikainstituut
Infosüsteemide õppetool

Ankurmodelleerimise mudelite realisatsiooni generaator PostgreSQL jaoks

Magistritöö

Üliõpilane: Elari Saal
Üliõpilaskood: 110061IAPM
Juhendaja: dotsent 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) (lõputöö kaitsja allkiri)

Ankurmodelleerimise mudelite realisatsiooni generaator PostgreSQL jaoks

Annotatsioon

Käesoleva magistritöö eesmärk on realiseerida ankurmodelleerimise mudelitest andmebaasikeele koodi genereerimise vahend. Genereeritud kood peab realiseerima ankurmudeli PostgreSQL andmebaasisüsteemis. Koodigeneraator oleks kasutatav koos olemasoleva veebipõhise ankurmodelleerimise vahendiga ja selle loomiseks kasutatakse vahendi olemasolevat raamistikku uute koodigeneraatorite loomiseks. Nimetatud eesmärgi saavutamiseks on vaja uurida ankurmodelleerimise meetodi alusel loodud andmebaaside realiseerimise võimalikkust PostgreSQL andmebaasisüsteemi abil. Selle uurimiseks kavandatakse ja seejärel realiseeritakse PostgreSQLis konkreetne ankurmudel raadiojaama esitlusloendi kohta. Mudel pole kuigi suur, kuid selles on kasutatud kõiki ankurmodelleerimise konstruktsioone. Ankurmodelleerimise olemasolev veebipõhine tarkvara toetas töö kirjutamise ajal täielikult ainult koodi genereerimist andmebaasisüsteemi MSSQL Server jaoks. PostgreSQL tugi puudus. Seetõttu teostatakse tarkvaraarendus, lisamaks modelleerimisvahendile andmebaasisüsteemi PostgreSQL tugi. Magistritöö tulemusena selgitatakse, kas ankurmodelleerimise mudelite realiseerimine PostgreSQLis on võimalik, kaardistatakse võimalikud kitsaskohad ning koostöös ankurmodelleerimise ideoloogia rajajaga täiendatakse modelleerimistarkvara.

Magistritöö on kirjutatud eesti keeles ning sisaldab teksti 90 leheküljel, 6 peatükki, 16 joonist, 6 tabelit.

A Generator for Generating Implementation of Anchor Modelling Models in PostgreSQL

Abstract

The goal of this thesis is to implement a code generator based on anchor models. The generated code must implement anchor models in PostgreSQL database management system. The code generator would accompany an existing web-based anchor modeling system and would be created by using its existing framework for creating code generators. To achieve the goal, the possibility of implementing anchor models in PostgreSQL must firstly be studied. For this purpose a specific anchor model of a radio station's set list will be designed and then implemented in PostgreSQL. The model is quite small but uses all the anchor modeling constructs. At the time of writing the thesis, the existing web based anchor modeling system only fully supported code generation for Microsoft SQL Server database management system. PostgreSQL support was not implemented. In the course of this work the possibility of implementing anchor models in PostgreSQL is examined, the potential bottlenecks identified, and the modeling software updated in cooperation with the author of the anchor modeling approach.

This thesis is written in Estonian and consists of 90 pages of text, including 6 chapters, 16 figures, and 6 tables.

Jooniste nimekiri

Tabelite nimekiri

Sisukord

Sissejuhatus

Tehnoloogia kiire arenguga käib kaasas andmehulkade kasv, andmete struktuuri ning andmetele kehtivate reeglite keerukuse kasv. Suurenevad ka nõudmised andmebaasis toimuvate operatsioonide kiirusele. On igati loogiline, et ka meetodid andmete hoidmiseks ja otsimiseks peavad arenema. Tänapäeval on endiselt valdavad SQL andmebaasid, mis on loodud SQL andmebaasikeelt toetavates andmebaasisüsteemides. Mõnedele (kuid kindlasti mitte kõigile) nende disainimisega seotud küsimustele annab vastuse normaalkujude teooria. See teooria pole „valmis", vaid areneb koos andmebaaside nende disainimise kohta käivate teadmiste kasvuga.

Bakalaureusetöös, mille autor kaitses aastal 2010 (Saal, 2010) ning selle põhjal kirjutatud teadusartiklis (Eessaar ja Saal, 2013), uuriti puuduvate/määramata andmete esitamist SQL-andmebaasides ning nende poolt põhjustatud kitsaskohti. Üks meetod ebatäielike või ennustamatute andmetega toime tulemiseks on kasutada kuuendal normaalkujul olevaid tabeleid. Sellistes tabelites on lisaks võtmele (kuhu kuulub üks või mitu veergu) maksimaalselt veel üks veerg. Lihtsustatult seletades, kui mõelda infosüsteemi valdkonnamudelile, siis selliselt disainitud SQL-andmebaasis on iga atribuudi kohta eraldi tabel. Bakalaureusetöö käigus tehtud eksperiment demonstreeris ilmekalt, et sellise disainiga andmebaasi korral on võimalik saavutada häid tulemusi andmete terviklikkuse, usaldusväärsuse ja otsimise kiiruse osas. Lisaks võimaldab selline tehniline lahendus erilise vaevata hoida ajaloolisi andmeid ning andmebaasi väga efektiivselt evolutsioneerida, sest kui andmebaasis on vaja hakata registreerima uutele atribuutidele või seosetüüpidele vastavaid väärtuseid, siis pole vaja muuta olemasolevate tabelite struktuuri, vaid tuleb luua uued tabelid. Paraku selgus ka, et selliselt disainitud andmebaasi loomine ilma tehniliste abivahenditeta on keerulisem, ja sellest tulenevalt ka kulukam, võrreldes näiteks maksimaalselt viiendal normaalkujul olevaid tabeleid sisaldava andmebaasi loomisega. See on autori hinnangul peamine põhjus, miks üks talle tuttav ettevõte kuuendal normaalkujul olevate tabelite kasutuselevõtust loobus.

Andmete modelleerimiseks ning ajalooliste ja puuduvate andmetega toime tulemiseks on välja pakutud uus lähenemine – ankurmodelleerimine, mille kasutamise tulemusena tekivad SQL-andmebaasis enamasti kuuendal normaalkujul olevad tabelid. Ankurmodelleerimist lihtsustab see, et erinevat tüüpi mudelielementide hulk on suhteliselt

väike. Ankurmodelleerimist saab töö kirjutamise ajal (2014. aasta sügisel) teostada graafilises veebipõhises modelleerimisvahendis. Lisaks võimaldab vahend genereerida andmebaasiobjektide loomiseks vajaliku koodi.

Olemasolev ankurmodelleerimise veebirakendus lihtsustab märkimisväärselt kuuendal normaalkujul olevate tabelite kasutuselevõttu, tehes selle potentsiaalselt ahvatlevaks senisest laiemale kasutajate ringile. Puudusena peab välja tooma, et olemasolev tarkvara toetas töö kirjutamise ajal (2014. aasta sügisel) ametlikult ainult andmebaasikeele lausete genereerimist andmebaasisüsteemi MSSQL Server jaoks.

Käesoleva magistritöö eesmärk on realiseerida ankurmodelleerimise olemasolevale veebipõhisele vahendile generaatorsüsteem PostgreSQL jaoks. Selle loomiseks kasutatakse vahendile loodud üldist genereerimise lahendust, mida on võimalik kohandada erinevate andmebaasisüsteemide jaoks. Ülesande täitmisel on alameesmärgiks välja selgitada, kas ankurmodelleerimise mudeleid saaks realiseerida andmebaasisüsteemis PostgreSQL loodud andmebaasides. Lähtuvalt autori üldistest kogemustest PostgreSQLiga alustatakse tööd eeldusega, et see on võimalik. Selle täpseks väljaselgitamiseks on vaja realiseerida ankurmodelleerimist kasutades üks andmebaas. Selleks luuakse viiendal normaalkujul olev andmebaasi disain, mis baseerub bakalaureusetöös kasutatud andmebaasi disainil. Seejärel modelleeritakse antud andmebaas ankurmodelleerimise veebirakenduse abil, genereeritakse olemasolevat generaatorit kasutades kood MS SQL Server jaoks ning üritatakse tõlkida see kood PostgreSQLile sobivaks. Selle käigus tuvastatakse ja kaardistatakse võimalikud kitsaskohad ja takistused.

Töö käigus kasutatakse modelleerimistarkvara Rational Rose („Rational Rose Modeler”) ning avalikku veebipõhist ankurmodelleerimise keskkonda (http://www.anchormodeling.com/modeler/latest/). Alameesmärgi täitmise käigus kirjeldavate skeemiobjektide identifikaatorid luuakse ingliskeelsed, lihtsustamaks eksperimendi kordamist eesti keelt mittekõnelevate isikute poolt.

Alameesmärgi täitmiseks realiseeritav andmebaasidisain on oluliselt lihtsustatud ja mõeldud ainult käesoleva töö raames planeeritud uuringu läbiviimiseks. Disaini alusel ei ole plaanis luua reaalset andmebaasilahendust reaalsele ettevõttele.

Töös tehtavad järeldused võiksid huvi pakkuda üksikisikutele, ettevõtetele ja organisatsioonidele, kes soovivad luua senisest efektiivsemaid lahendusi infosüsteemide (sealhulgas andmebaaside) evolutsioneerimiseks, ajalooliste andmete säilitamiseks ning puuduvate andmetega toime tulemiseks. Samuti on töö abiks nendele, kes üritavad realiseerida ankurmodelleerimise andmebaase mõnes teises andmebaasisüsteemis.

1. Taust

Antud peatükis antakse lühiülevaade normaliseerimisest, normaalkujudest, ankurmodelleerimisest ja tabeli elimineerimise teisendusest. Jutt käib SQL-andmebaaside kohta. SQL andmebaasid põhinevad SQL andmebaasikeele aluseks oleval andmemudelil, mis omakorda on väljatöötatud relatsioonilise andmemudeli põhimõtteid (aga mitte kõiki põhimõtteid) järgides. SQLi kirjeldab rahvusvaheline standard, mille viimane redaktsioon töö kirjutamise ajal oli SQL:2011. Järgnevas kirjelduses kasutatakse SQLi mõisteid (tabel, rida, veerg), mitte relatsioonilise andmemudeli mõisteid (relatsiooniline muutuja, relatsioon, atribuut, korteež). Tabelite all peetakse silmas baastabeleid (tabelid, mis pole defineeritud teiste tabelite põhjal).

Oskamatu andmebaasi disain võib põhjustada ootamatuid probleeme, mis mõjutavad olulisel määral andmebaasi kasutavate programmide tööd. Eriti problemaatilisteks võib pidada juhtumeid, kus peame kahtlema andmete õigsuses või peame silmitsi seisma dubleeritud andmetega. Mõlemal juhul satub andmete usaldusväärsus küsimärgi alla. Nimetatud, ja ka mitmeid teisi probleeme aitab vähendada (kuid kindlasti mitte lõpuni vältida) tabelite normaliseerimine.

Andmete õigsuse ja terviklikkuse paremaks kindlustamiseks on relatsioonilise ja SQL-andmemudeli jaoks välja töötatud normaliseerimise protsess („Database normalization”), mille käigus viiakse tabelid üha kõrgematele normaalkujudele (*normal form*, *NF*). Selle protsessi pakkus esimesena välja relatsioonilise mudeli autor E. F. Codd (1972) ning seda on hiljem väga palju edasi arendatud (Eessaar, 2014).

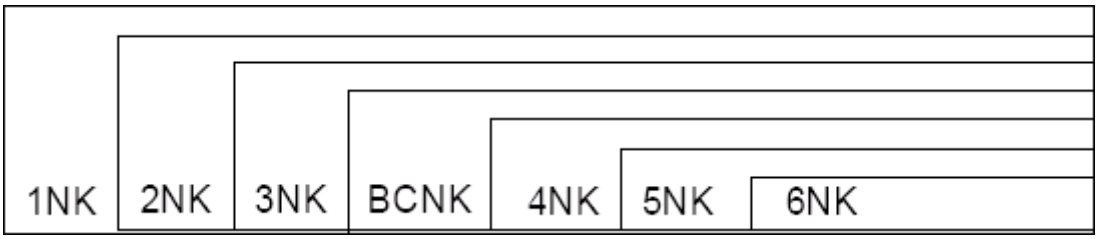
Normaliseerimine on protsess, mille igal sammul kontrollitakse normaliseeritava tabeli vastavust mingile reeglite hulgale. Kui tabel ei vasta reeglitele, tuleb see dekomponeerida (jagada, lagundada, tükeldada) välisvõtmete kaudu seotud tabeliteks, millest igaüks vastab nimetatud reeglitele. Dekomponeerimiseks rakendatakse tabelile projektisooni operatsioon (*projection*) ning algse tabeli taastamiseks on vajalik ühendamise operatsioon (*join*). Iga normaliseerimise sammu tulemusel viiakse tabelid uuele, kõrgemale normaalkujule. Põhimõtteliselt on tabelite dekomponeerimiseks võimalik kasutada ka mõnda muud operatsiooni kui projektisoon (nt piirang). Kuid sellisel viisil normaliseerimine pole käesoleva töö teemaga otseselt seotud ja seda rohkem ei käsitleta.

Tabel, mille väljades olevad väärtused kujutavad endast andmete kasutaja jaoks atomaarseid andmeelemente, kusjuures igas väljas on maksimaalselt üks selline andmeelement, on normaliseeritud ehk esimesel normaalkujul. Atomaarsus andmete kasutaja

jaoks tähendab, et väärtus on kasutajale „suupärane" „kängar", mille väiksemateks osadeks lahutamist ei peeta vajalikuks. Samas on „atomaarsus" suhteline (ja seega mitte täpselt defineeritav) mõiste, sest näiteks kuupäeva, mida ilmselt vaadeldakse atomaarse elemendina, saab lahutada kuu, päeva ja aasta komponentideks. Samuti saab vabatekstilist väärtust, mida ilmselt ka käsitletakse atomaarse elemendina, lahutada üksikuteks märkideks.

Atomaarseks andmeelemendiks olev väärtus ei pea olema „lihtne" väärtus nagu kuupäev või tekst, vaid võib vastavalt oma andmetüübile olla keeruka sisemise struktuuriga (näiteks sõrmejälge esitav väärtus). Täiendava normaliseerimise (sageli öeldakse lihtsalt normaliseerimise) üks eesmärk on andmete liiasuse vähendamine. Ühtlasi esineb vähem andmete muutmise anomaaliaid ning andmebaasis olevate andmete struktuur muutub kasutajale arusaadavamaks (Eessaar, 2014).

Täiendava normaliseerimise veel üks eesmärk on parandada loogiliselt ebakorrektset disaini, mis tähendab et täiendava normaliseerimise tulemusena saab andmebaasis registreerida tõeseid väiteid, mida seal ilma täiendava normaliseerimiseta ei olnud võimalik registreerida. Andmebaasi struktuur muutub kergemini mõistetavamaks ja paremini „reaalset maailma" peegeldavaks. Sõltuvalt päringust võib paraneda ka andmete otsimise kiirus. Lisaks muutub lihtsamaks mitmete kitsenduste jõustamine (nende jõustamiseks hakkab piisama tabelites võtmete deklareerimisest).



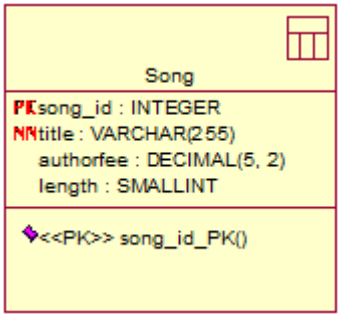
Joonis 1. Enamlevinud normaalkujud (Eessaar, 2008)

On kirjeldatud palju erinevaid normaalkujusid. Mida kõrgem on vaadeldav normaalkuju, seda rangemad reeglid kehtivad sellele vastava tabeli struktuurile. Sealjuures sisaldab iga järgmine normaalkuju astmele viimise reeglistik madalamate astmete dekompositsiooni reegleid (vt joonis 1). Tabel, mis on normaalkujul N, on kindlasti normaalkujul N-1. See tabel võib, kuid ei pruugi, olla normaalkujul N+1.

1.1. Kuues normaalkuju

Öeldakse, et tabel on normaliseeritud, kui antud tabel on vähemalt esimesel normaalkujul. Lisaks öeldakse, et tabel on **täielikult** normaliseeritud, kui tabel on viidud viiendale normaalkujule. Andmete liiasuse puudumise mõttes on olukord sellise tabeli puhul parim, mis selle lähenemisega võimalik. Samas ei taga ka kõigi tabelite viiendal normaalkujul olemine, et andmebaasist on andmete liiasus täielikult kadunud. Vahemärkusena olgu öeldud, et andmete liiasuse kaotamine ei ole eesmärk omaette. Andmebaasis võib teatud pragmaatilistest kaalutlustest lähtuvalt olla andmete liiasus, kuid see peab olema kontrollitud (st andmebaasi kasutaja peab olema sellest teadlik ja andmebaasisüsteem hoolitseb, et liiasusest tulenevalt ei jõua andmebaasi vastuolulised andmed).

Eksisteerib veelgi kõrgem ning võib-olla vähem tuntud normaalkuju – kuues normaalkuju, mis viib dekomponeerimise uuele tasemele. Tabelite sellisel viisil dekomponeerimine ei aita enam vähendada andmete liiasust (liiasusest vabanemise mõttes oli viies normaalkuju kõrgeim), kuid selle kasutamisel võib olla mitmeid muid eeliseid.



Joonis 2. Viiendal normaalkujul olev tabel

Joonisel 2 on ära toodud viiendal normaalkujul olev muusikapalade tabel *Song*, mis koosneb primaarvõtme veerust *song_id*, milles on süsteemi-genereeritud unikaalsed täisarvud, ja kolmest olemitüübi *Song* sisulistele atribuutidele vastavast veerust: pealkiri (*title*), autoritasu määr (*authorfee*) ja loo pikkus (*length*). Antud tabelit kasutatakse ka käesoleva töö raames läbi viidavas eksperimendis.

song_id	title	authorfee	length
1	Sinuta	2,00 €	3:35
2	Hopp Johanna	2,00 €	3:11
3	Candle In The Wind	3,00 €	4:12
4	Laika	2,00 €	*NULL*
5	Kiigelaul	*NULL*	*NULL*

Joonis 3. Viiendal normaalkujul oleva tabeli võimalik sisu

Tabeli struktuuri (joonis 2) ja võimalikku sisu (joonis 3) lähemalt uurides näeme, et autoritasul ja loo pikkusel ei ole NOT NULL kitsendust, mis osundab, et antud väljad võivad ükskõik millise muusikapala puhul tühjad olla (neis on sellisel juhul NULL-marker). Andmete kvaliteedi seisukohast on tegemist tõsise probleemiga, sest lisaks ebatäielikele andmetele andmebaasis puudub meil ka arusaam, miks mingi väärtus sisestamata jäi. Põhjuseid võib olla erinevaid:

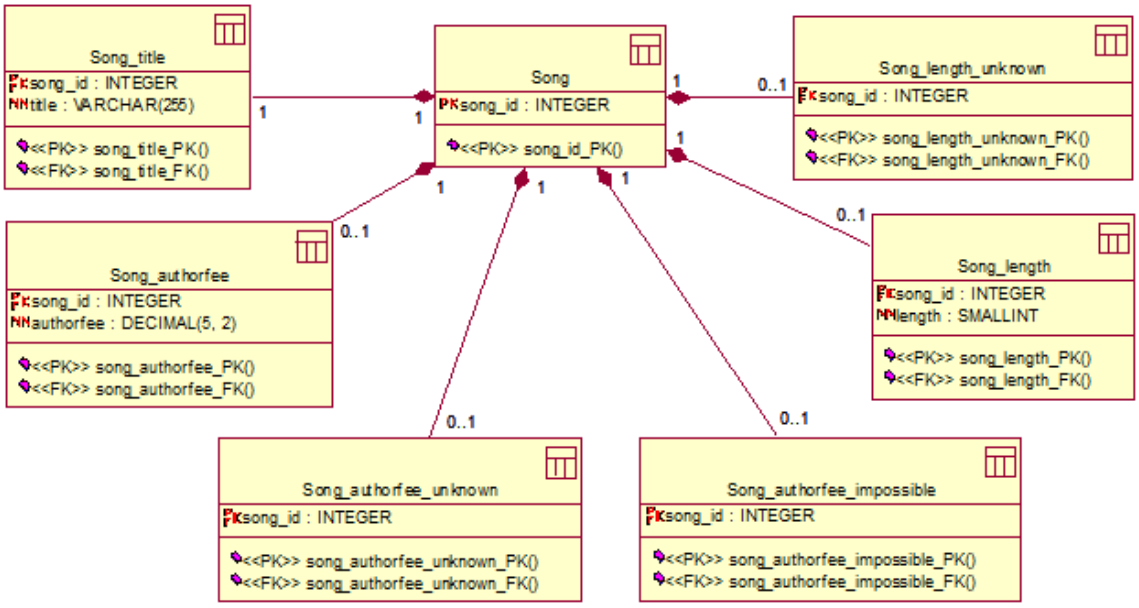
- andmebaas on halvasti kavandatud ning nõuab andmete registreerimist, mida kunagi ei teki,
- andmete sisestaja ei omanud piisavalt informatsiooni,
- andmete sisestaja tegi vea,
- viga tarkvaras või riistvaras,
- küberrünnak,
- tegemist oli arvutatava väärtusega, mille tulemus pole nulliga jagamise tõttu määratud,
- muu mitteloetletud põhjus.

On tõenäoline, et valdav enamus SQL-andmebaaside disaineritest kasutab rohkem või vähem teadlikult mingil viisil normaliseerimise teooriat. Paraku aga näeme, et isegi viiendat normaalkuju kasutades ei leia me paljudele olulistele probleemidele lahendust.

Viiendal normaalkujul olevad tabelid saab jagada veelgi väiksemateks osadeks, luues iga kontseptuaalse andmemudeli atribuudi jaoks eraldi tabeli, seostades selle välisvõtme abil olemitüübile vastava esialgse tabeliga. Sellised tabelid on kuuendal normaalkujul. Tabelite kokku ühendamise tulemusena taastatakse esialgne tabel, sealjuures kõiki tükeldatud osi läheb taastamisel vaja.

Kuues normaalkuju (6NK) tähendab, et tabelit ei ole võimalik projektsiooni kasutades rohkem dekomponeerida, põhjustamata andmekadusid. Definitsioonist tulenevalt – kui tabelis T on ainult üks kandidaatvõti V ja lisaks maksimaalselt üks veerg, mida V ei hõlma, siis on T

kuuendal normaalkujul (Saal, 2010). Teisisõnu, selliselt disainitud SQL-andmebaasis on iga kontseptuaalse andmemudeli atribuudi kohta eraldi tabel.



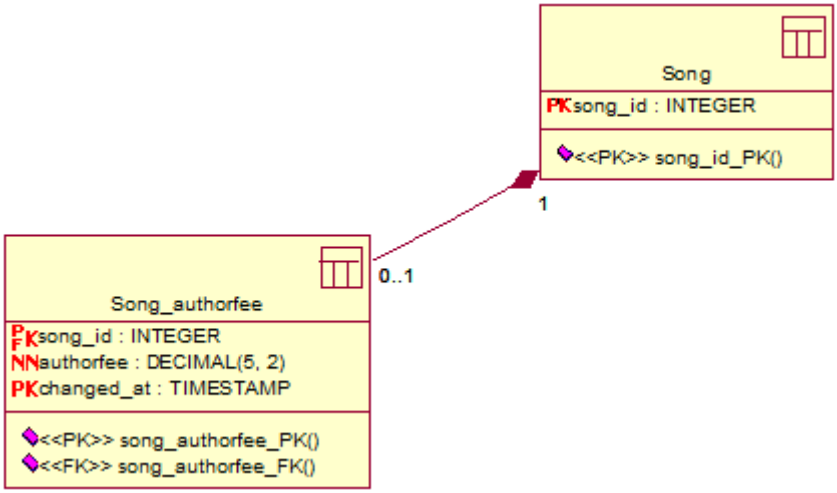
Joonis 4. Kuuendal normaalkujul olevad tabelid

Sõltuvalt tabelite kuuendale normaalkujule viimise eesmärkidest on tulemuseks erinev tabelite hulk. Joonisel 4 esitatakse joonisel 2 kujutatud tabeli *Song* kuuendale normaalkujule viimise üks võimalik tulemus, mis võimaldab pidada arvet erinevate põhjuste kohta, miks mõni andmeväärtus andmebaasist puudub. Näeme, et NOT NULL-kitsenduseta veerge andmebaasis enam ei leidu. Tabel *Song* sisaldab ainult ühte veergu (*song_id*), millega on ka seotud arvujada generaator (*sequence*). Muusikapala iga kontseptuaalne atribuut on realiseeritud vähemalt ühe eraldi tabeliga. Näeme, et on tekkinud tabelid *Song_authorfee_unknown* ja *Song_authorfee_impossible*. Antud tabelid sisaldavad nende muusikapalade primaarvõtme väärtusi, millede puhul saame vastavalt öelda, et autoritasu määr pole teada või pole autoritasu määr võimalik (näiteks autoriõiguse seaduse poolt kaitsmata rahvalaulude puhul). Tabel *Song_length_unknown* määrab lood, millede puhul kestus on teadmata. Antud meetod tabelite tükeldamiseks demonstreerib ilmekalt, et kuues normaalkuju võimaldab lahendada puuduvate andmete probleemi, spetsifitseerides, millised andmed puuduvad, ning võimaldades registreerida ka puudumise põhjuse.

Algse tabeli *Song* (vt joonis 2) oleks saanud viia kuuendale normaalkujule ka nii, et iga kontseptuaalse atribuudi kohta tekibki ainult üks eraldi tabel. Sellisel juhul poleks võimalik pidada arvestust selle üle, mis *põhjusel* mõni andmeväärtus puudub. Ühtlasi näitab

see, et tabeli dekomponeerimisel (nagu üldse disainimisel) on sageli võimalik teha erinevaid valikuid ning parim valik sõltub osaliselt ka süsteemile esitatud nõudmistest.

Kuuenda normaalkuju eelisena tuleb kindlasti välja tuua, et kui tekib vajadus hakata registreerima uuele kontseptuaalsele atribuudile vastavaid andmeid, siis kaob vajadus muuta olemasolevate tabelite struktuuri. Näiteks, kui muusikapalade korral tekib vajadus hakata registreerima atribuudile „loomise aasta" (*creation year*) vastavaid andmeid, siis andmebaasi tekiks uus tabel *Song_creation_year*. Olemasolevate tabelite struktuuri ei ole vaja selleks muuta. Andmebaasi kasutava rakenduse töö, mis ei vaja aasta andmeid, pole sellisest muudatusest mõjutatud.



Joonis 5. Ajaloolised andmed ja kuues normaalkuju

Infosüsteemide puhul peame rääkima ka ajaloolistest andmetest. Falkenberg (1992) kirjeldab nõudmisi evolutsioneeruvale infosüsteemile. Üheks nõudmiseks on, et süsteem peab pidama meeles kõik selles registreeritud andmed, välja arvatud juhul kui andmete mitte meeles pidamiseks antakse ilmutatud kujul korraldus (nt seadustest tulenevatel põhjustel või soovist tagada osapoolte privaatsust). Oletame, et tahaksime talletada muusikapalade autoritasude muutumise ajalugu. On loogiline oletada, et ajas tasud muutuvad (ilmselt kasvavad). Joonisel 5 on toodud tabel *Song_authorfee*. Näeme, et autoritasu muutuste jälgimiseks piisab, kui tabelile lisada autoritasu muutmise aegasid talletav veerg *changed_at*. Andmete haldamise süsteemi jõudlust parandab see, et ajalooliste atribuutide ja sidemete puhul pole andmete muutmisel vaja olemasolevaid ridu lukustada, sest tabelisse lisandub uus rida, uuema ajaga. Kui mõne reegli kontrollimiseks on vaja (trigeris, protseduuris või

rakenduses) mõnda rida ilmutatud kujul eksklusiivselt lukustada, siis sellise disaini korral mõjutab see väiksemat hulka andmeid.

Kehtiva autoritasu leidmiseks saaks kasutada SQL-lauset:

```
SELECT
      authorfee
FROM Song_authorfee
WHERE song_id = 3
ORDER BY changed_at DESC
FETCH FIRST 1 ROWS ONLY;
```

Vahemärkusena olgu öeldud, et autoritasude muutus võis tulla ka riigis kehtiva valuuta muutusest (nagu Eestis üleminek kroonidelt eurodele). Sellisel juhul saame samadel põhimõtetel lisada andmebaasi tabeli *Song_autorfree_currency* ja registreerida seal, millistel ajaperioodidel oli kasutusel millised valuutad.

Minnes jälgimisega detailsemaks, peame eristama alljärgnevaid ajahetki:

- millal tehti muutmise otsus,
- millal muudatus hakkas kehtima,
- millal registreeriti muudatus andmebaasis.

Viiendal normaalkujul olevate tabelite korral on ajalooliste andmete talletamine keerukam. Üks võimalik variant on, et autoritasude veerg tuleks muusikapalade tabelist „välja kolida" ning kasutada eraldiseisvat tabelit, analoogselt joonisel 5 kujutatud meetodil. Sisuliselt on tegemist hübriidse disainiga, kus teatud andmed on kuni viiendal normaalkujul olevates tabelites ja teatud andmed kuuendal normaalkujul olevates tabelites. Veel üks variant on, et rea mingis väljas muudatuste tegemine tingib rea muudatuse eelse versiooni registreerimise eraldi tabelis koos muutmise metaandmetega. Selle lähenemise probleem on, et kuigi muudeti vaid ühe atribuudi väärtust, tehti koopia tervest reast. See suurendab kindlasti märkimisväärselt andmemahte. Sellel lähenemisel põhineb nt *Oracle Workspace Manager*, mis võimaldab teostada andmebaasis olevate andmete versioonihaldust (Potter, 2013).

Veel üks Falkenbergi (1992) nõudmistest evolutsioneeruvale infosüsteemile seisneb selles, et muudatus infosüsteemis ei tohi katkestada seda süsteemi kasutava organisatsiooni tööd. Kui andmebaasis on vaja hakata registreerima uutele atribuutidele või seosetüüpidele vastavaid väärtuseid, siis kuuendal normaalkujul tabelite disaini korral on vaja luua uued tabelid, mitte pole vaja muuta olemasolevate tabelite struktuuri. See tähendab vähem

muudatusi infosüsteemi teistes osades ja vähem tõrkeid infosüsteemi kasutava organisatsiooni töös.

Eelnevast saab järeldada, et kuuendal normaalkujul olevatel tabelite kasutamisel on mitmeid olulisi eeliseid madalamatel normaalkujudel olevate tabelite kasutamise ees: parem puuduvate andmetega toimetulemine, ajalooliste andmete hoidmise võimaluse lisamise lihtsus ning andmebaasi evolutsioneerimise lihtsus ja madalam risk.

Rääkides kuuenda normaalkuju disaini puudustest peab aga esile tooma, et tabelite arv seitsmekordistus (joonis 4), võrreldes viiendal normaalkujul oleva tabeliga *Song* (joonis 2). Välisvõtmeid, mida meie näites viienda normaalkuju puhul kasutada polnud vaja, on nüüd kuus. Võib järeldada, et kuues normaalkuju lahendab probleeme, tuues kaasa uue probleemi – andmebaasi struktuuri keerukuse kasvu. Veel üheks probleemiks on see, et kuuenda normaalkuju disaini korral muutub palju keerulisemaks andmebaasi tasemel erinevate kitsenduste jõustamine, mis peavad tagama andmete (ja selle kaudu ka neid andmeid kasutavate protsesside) vastavuse ärireeglitele. Kujutame näiteks ette olemitüüpi *Performing* (laulu esitamine), millel on atribuudid *start_time* ja *end_time* (alguse aeg ja lõpu aeg). Kontrollimaks, et lõpu aeg pole väiksem kui alguse aeg, saab viienda normaalkuju disaini korral deklareerida lihtsa tabelitaseme kitsenduse *end_time* >= *start_time*. Kuid kuuenda normaalkuju disaini korral oleks vaja selle kitsenduse kontrollimiseks lugeda andmeid kahest erinevast tabelist. Seda saaks SQLis teha deklaratiivselt, kui SQL-andmebaasisüsteemid toetaksid üldiseid deklaratiivseid kitsendusi (*Assertion*), mida need paraku ei tee. Seega ei jää üle muud kui kirjutada keerulist ja veaohtlikku trigeri protseduuride koodi või loobuda kitsenduste jõustamisest andmebaasi tasemel ning kirjutada koodi kõikide andmebaasi kasutavate rakenduste tasemel või seda kontrolli üldse mitte jõustada, lootes kasutajate tähelepanelikkusele ja arukusele (mis on loomulikult väga naiivne lootus).

Bakalaureusetöö (Saal, 2010) eksperiment näitas, et isegi lihtsa andmebaasiskeemi korral kasvab tabelite arv vähemalt viis korda, veergude arv kaks korda, välisvõtmete arv kuus korda ja uue muusikapala lisamiseks vajaliku SQL-lause pikkus kuus korda.

Bakalaureusetöö käigus selgus ka, et andmete otsimine (SELECT operatsioon) kuuendal normaalkujul olevatest tabelitest käib väga kiiresti. Samal ajal on andmete sisestamine (INSERT) aeglasem, võrreldes viiendal normaalkujul olevate tabelitega. Sellest võib järeldada, et infosüsteemis, kus teostatakse vähe INSERT-operatsioone ning palju SELECT-operatsioone, oleks kuuenda normaalkuju kasutamine õigustatud, hoolimata selle keerukusest. Seda asjaolu silmas pidades tutvustas autor kuuenda normaalkuju eeliseid enda töökohas, proovimaks selle kasutamist reaalses projektis, kus andmeid tuli tihti otsida ning

harva sisestada või muuta. Kahjuks keeldusid projekti asjaosalised idee proovimisest, nähes slaididelt andmebaasi struktuuri ja SQL-lauseid pärast struktuuri viimist kuuendale normaalkujule.

Kuuendal normaalkujul on mitmeid eeliseid, aga selle kasutuselevõtt tähendab ettevõtte jaoks lühemas perspektiivis selgelt rohkem tööks kulutatud inimtunde, raha ja koolitusvajadust. Probleemi aitaks leevendada mingi modelleerimisvahendi olemasolu, mis andmebaasi loomist ja evolutsioneerimist automatiseerida või lihtsustada aitaks. See viib meid ankurmodelleerimise juurde, mis on just selleks mõeldud.

1.2. Ankurmodelleerimine

Rääkides ankurmodelleerimisest, tasub korraks peatuda andmeaitadel e andmeladudel (*data warehouse*). Andmeaidal on tüüpiliselt alljärgnevad omadused:

- kasutusel on SQL-andmebaasisüsteem,
- andmeid on palju (gigabaite kuni petabaite),
- andmeid on lisatud pika perioodi vältel (üldjuhul rohkem kui aasta),
- andmed pärinevad erinevatest süsteemidest (kus tihti kasutatakse erinevaid kodeeringuid ja formaate),
- andmed on osaliselt või täielikult ajaloolised (säilib muudatuste ajalugu),
- andmeaita administreerivad ning haldavad isikud võivad muutuda,
- andmete põhjal teostatakse analüüsi, kusjuures analüüsi meetod võib ajas muutuda,
- iseloomulikud operatsioonid on andmete andmebaasi laadimine (INSERT laused) ja suurt hulka andmeid hõlmavad keerukad koondandmete leidmise päringud (SELECT laused).

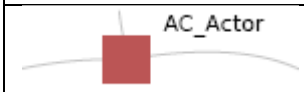
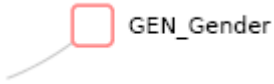
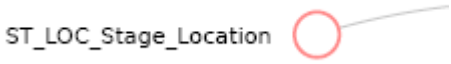
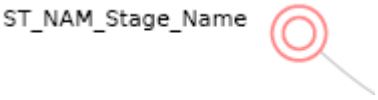
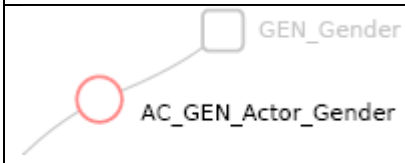
Andmeaida loomine ja haldamine on vastutusrikas ning keeruline töö. Võib oletada, et andmete hulk kasvab, andmete struktuur muutub ajapikku järjest keerulisemaks ning nõudmised jõudlusele, andmeformaatide ning operatsioonide mitmekesisusele, aga ka andmete terviklikkusele kasvavad. Vigu, mida tehakse andmeaitu luues, on hiljem äärmiselt keeruline ning kulukas parandada. Seetõttu peab andmeaida kontseptsioon olema esimesest päevast alates selline, mis arvestaks tuleviku, muutuste ning võimalike probleemidega.

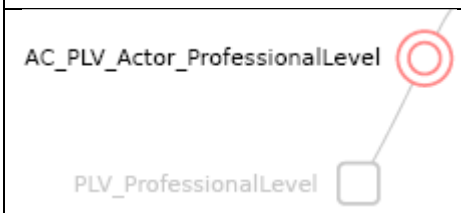
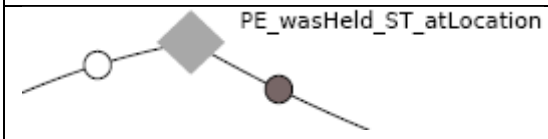
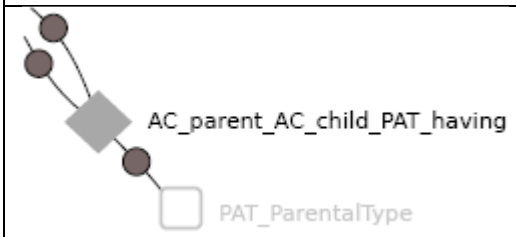
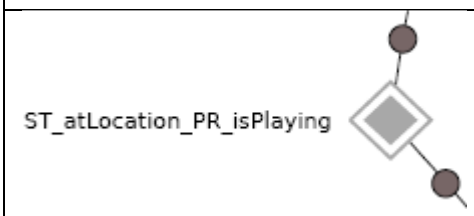
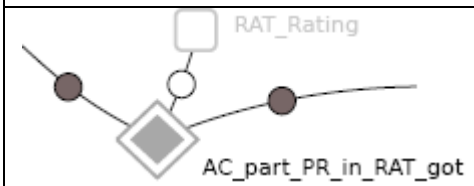
Ankurmodelleerimine on agiilne (paindlik, väle) andmebaasi projekteerimise modelleerimistehnika (Rönnbäck et al., 2010), mis sai inspiratsiooni just andmeaitade loomisel ja kasutamisel tehtud vigadest. Ankurmodelleerimise abil loodud andmebaas tuleb

nimelt äärmiselt hästi toime ajas nii struktuuri kui ka sisu poolest muutuvate andmetega. See sobib loomulikult kasutamiseks ka teistsugustes süsteemides (ka näiteks operatiivandmete süsteemides, kus põhilised operatsioonid on üksikute olemite andmete otsimine ja muutmine).

Ankurmodelleerimise modelleerimiskeel võimaldab kirjeldada olemitüüpe (ankur), atribuute, seosetüüpe (side) ja klassifikaatoreid (sõlm) (vt Tabel 1). Atribuutidele, seosetüüpidele ja klassifikaatoritele saab lisada võimaluse väärtuste ajaloo säilitamiseks (Potter, 2013).

Tabel 1. Ankurmodelleerimises kasutuselolevad elemendid

Ikoon	**Nimi**
	Ankur (*anchor*) Ankur esindab olemitüüpe (nagu artist).
	Sõlm (*knot*) Sõlm on olemitüüp, mis esitab klassifikaatorit (nagu stiiliroll). See esitab andmeid, mida saab kasutada teiste andmete liigitamiseks.
	Staatiline attribuut (*static attribute*) Olemitüüpi kuuluvate olemite omadus, mille väärtuste ajalugu ei soovita andmebaasis säilitada (nagu muusikapala kestus). „Staatiline” tähendab, siin ja edaspidi, et andmebaas ei pea säilitama vastavate andmete ajalugu.
	Ajalooline attribuut (*historized attribute*) Olemitüüpi kuuluvate olemite omadus, mis võib ajas muutuda (nagu autoritasu määr). „Ajalooline” tähendab, siin ja edaspidi, et andmebaas peab säilitama vastavate andmete ajalugu.
	Sõlmitud staatiline attribuut (*knotted static attribute*) Ühendab sõlme ja ankru, et kirjeldada

Ikoon	Nimi
	omadusi, millel pole palju erinevaid võimalikke väärtusi, st on võimalik moodustada nende väärtuste nimekiri (nagu isiku sugu).
	Sõlmitud ajalooline attribuut (*knotted historized attribute*) Ühendab sõlme ja olemi, et kirjeldada omadusi, millel pole palju erinevaid väärtusi, kuid mis võivad ajas muutuda (nagu isiku sugu).
	Staatiline side (*static tie*) Seos vähemalt kahe ankru vahel.
	Sõlmitud staatiline side (*knotted static tie*) Seos vähemalt kahe ankru ja ühe sõlme vahel.
	Ajalooline side (*historized tie*) Ajalooline seos vähemalt kahe ankru vahel.
	Sõlmitud ajalooline side (*knotted historized tie*) Ajalooline seos vähemalt kahe ankru ja ühe sõlme olemi vahel.

Juhin tähelepanu ühele probleemile seoses pakutavate mudelielementidega. Sõlme puhul ei saa säilitada ajaloolisi andmeid (puudub ajalooline sõlm e *historized knot*). Samas klassifikaatorite väärtused muutuvad ajas. Mõelge näiteks, kui palju on maailma riikide hulk või Eesti haldusterritoriaalne jaotus viimase 25 aasta jooksul muutunud. Ka olemite seisundid (seisundi klassifikaatorid) muutuvad ajas seoses äriprotsesside muutumisega. Jääb väga vähe asju, mida modelleerida sõlmena. Isegi „sugu” ei sobi, kui mõelda kolmanda soo võimalusele

(„Australian Passport Office”). Kui soovitakse säilitada klassifikaatorite väärtuste ajalugu, siis on need parem modelleerida ankrutena.

Modelleerimist saab teostada graafilises veebipõhises rakenduses (http://www.anchormodeling.com/modeler/latest/). Rakendus on tasuta ja avatud lähtekoodiga. Loodud mudeli baasil saab genereerida andmebaasiobjektide loomiseks vajaliku koodi. SQL-lausete genereerimiseks kasutatakse teisendusreegleid, mis loovad tabelid, trigerid, kitsendused, vaated, funktsioonid ja kommentaarid. Selleks kasutatakse mallimootorit Sisula. Loodud tabelid on üldjuhul kuuendal normaalkujul.

```
-- Knots are used to store finite sets of values, normally used to describe states
-- of entities (through knotted attributes) or relationships (through knotted ties).
-- Knots have their own surrogate identities and are therefore immutable.
-- Values can be added to the set over time though.
-- Knots should have values that are mutually exclusive and exhaustive.
-- Knots are unfolded when using equivalence.
--
-- Knot table -----------------------------------------------------------------
-- SRO_StyleRole table
-------------------------------------------------------------------------------
IF Object_ID('dbo.SRO_StyleRole', 'U') IS NULL
CREATE TABLE [dbo].[SRO_StyleRole] (
    SRO_ID smallint IDENTITY(1,1) not null,
    SRO_StyleRole varchar(100) not null,
    Metadata_SRO int not null,
    constraint pkSRO_StyleRole primary key (
        SRO_ID asc
    ),
    constraint uqSRO_StyleRole unique (
        SRO_StyleRole
    )
);
GO
```

Joonis 6. Genereeritud koodi näide

Töö kirjutamise ajal toetas koodigeneraator andmebaasiobjektide loomiseks vajalike SQL-lausete genereerimist ainult andmebaasisüsteemi MS SQL Server jaoks (joonis 6). Muude levinud SQL-andmebaasisüsteemide (PostgreSQL, MySQL, Oracle jne) toe lisamine aitaks kaasa ankurmodelleerimise kasutamise levikule. Seetõttu on käesoleva töö eesmärgiks lisada ankurmodelleerimise vahendile võimalus genereerida koodi PostgreSQLi jaoks. Lähtuvalt autori üldistest kogemustest alustatakse tööd eeldusega, et ankurmodelleerimise mudeli realiseerimine PostgreSQLis on võimalik.

Ankurmodelleerimise teooria ühe autori, Lars Rönnbäcki sõnul on maailmas hinnanguliselt 50-100 andmeaita ja andmebaasi, mille loomiseks kasutati ankurmodelleerimist. Esimene neist (https://www.lansforsakringar.se/privat/) loodi Rootsis juba aastal 2004 (käsitsi, sest generaatorit veel ei eksisteerinud). Samuti on teada, et ettevõte Teracom (http://www.teracom.se) on käivitanud vähemalt kolm ankurmodelleeritud rakendusandmebaasi ja neli andmeaita, ning Hollandi ettevõte m-wise (http://m-wise.eu/en/) vähemalt 14 andmeaita. Suurimas teadaolevas ankurmodelleeritud andmebaasis hoitakse üle 40 TB andmeid. („Privaatne kommunikatsioon“)

1.3. PostgreSQL

PostgreSQL on populaarne, tasuta ning vabatarkvarana pakutav objekt-relatsiooniline andmebaasisüsteem („DB-Engines Ranking”). Eestis kasutavad seda näiteks Skype ja riigiportaal eesti.ee („Riigiportaali „eesti.ee” tehnoloogilise raamistiku versioonimise tarkvara tellimine”). PostgreSQL on hästi dokumenteeritud ning kasutajasõbralik vahend andmete ladustamiseks ning töötlemiseks, mis toetab kaasaegseid klasterdamise ning optimeerimise lahendusi. 2015. aasta jaanuarikuu seisuga on PostgreSQL andmebaasisüsteemide populaarsuse indeksis neljandal kohal („DB-Engines Ranking”).

Üks põhjus, miks käesolev töö uurib ankurmodelleerimise generaatori realiseerimist andmebaasisüsteemis PostgreSQLis, seisneb selles, et PostgreSQLi peetakse tänapäeval sobivaks andmebaasisüsteemiks andmeaitades kasutamiseks (Bartolini, 2009). Samuti tasub esile tuua, et tegemist on tasuta tarkvaraga, mis arendajatele pakutavate võimaluste poolest konkureerib parimate kommertsandmebaasisüsteemidega ning seetõttu võib nii majanduslikus kui tehnilises plaanis selle andmebaasisüsteemi kasutamine olla mõistlik. Kindlasti on oma osa ka selles, et autor on selle andmebaasisüsteemiga nii ülikoolis kui tööelus kokku puutunud. Samuti soovitas seda andmebaasisüsteemi juhendaja, kuna PostgreSQL on kõigele lisaks väga hästi laiendatav ning seega hea platvorm infosüsteemide evolutsioneerimise tehnikate ja meetodite tundmaõppimiseks ning evolutsioneeruvate infosüsteemide realiseerimiseks.

1.4. Tabeli elimineerimise teisendus

Ankurmodelleerimise tehnikat kasutades tekib andmebaas, kus üldjuhul on väga palju tabeleid (joonisel 4 toodud ning kuuendale normaalkujule viidud tabel *Song* demonstreeris seda

ilmekalt). Mõistlik oleks luua vaated e virtuaalsed tabelid (*views*), mis tabelid suuremaks loogiliseks tervikuks kokku tagasi ühendavad. Nii toimides lihtsustub andmebaasi kasutus ning andmebaasi „peale" rakenduskihti loovad isikud ei ole kohustatud end kuuenda normaalkuju teooriaga kurssi viima.

Võttes aluseks joonisel 4 toodud kuuendal normaalkujul olevad muusikapala kirjeldavad tabelid, saaks luua järgmise vaate, mis oleks väga sarnane joonisel 2 toodud viiendal normaalkujul oleva tabeliga:

```
CREATE OR REPLACE VIEW Songs AS
SELECT
      S.song_id,
      ST.title,
      SA.authorfee,
      SL.length
FROM Song AS S
LEFT JOIN Song_title AS ST
      ON ST.song_id = S.song_id
LEFT JOIN Song_authorfee AS SA
      ON SA.song_id = S.song_id
LEFT JOIN Song_length AS SL
      ON SL.song_id = S.song_id
ORDER BY S.song_id ASC;
```

Näeme, et tabelite ühendamiseks tuleb teostada kolm ühendamise (*join*) operatsiooni. Päring osutab, et ühendamisoperatsioonide arv on vähemalt võrdne ankru atribuutide arvuga, mis keerukama infosüsteemi puhul tähendab, et vaate alampäringus võib olla kümneid ühendamisoperatsioone. Pange tähele, et vaate alampäringus kasutatakse välisühendamise (*outer join*) operatsioone, kuna atribuutidele vastavates tabelites võivad seotud read puududa, kuna olemi atribuudi väärtused puuduvad.

Oletame, et soovime leida selliste muusikapalade autoritasu määra, mille nimes sisaldub sõna „*Wind*". Päring vaatest oleks alljärgnev:

```
SELECT
      song_id,
      authorfee
FROM Songs
WHERE title LIKE '%Wind%';
```

Näeme, et andmete lugemine tabelist *Song_length* on sellise päringu puhul ebavajalik, mistõttu oleks mõistlik lause täitmisel vastav ühendamisoperatsioon tegemata jätta.

Ühendamisoperatsioonide läbiviimine muudab lause täitmise aeglasemaks. Siinkohal tulevad appi andmebaasisüsteemidesse sisse ehitatud optimeerimismoodulid, mis võivad

andmekäitluskeele lause töötlemise käigus rakendada tabelite elimineerimise tehnikat („What is Table Elimination? ”).

Tabeli elimineerimise teisenduse abil saab optimeerija päringust elimineerida ühendused tabelitesse, kust infot ei pärita või millede kasutamata jätmine tulemust ei mõjuta. Nii välditakse asjatut andmete seostamist ning hoitakse kokku arvutusressurssi. Teiste sõnadega, andmebaasisüsteem kirjutab kasutaja poolt esitatud lause ümber, et see oleks loogiliselt samaväärne, kuid lihtsam ning ühtlasi kiiremalt täidetav. Tabeli elimineerimine on võimalik vaid siis, kui andmebaasis on deklareeritud kandidaatvõtmed ja välisvõtmed. See annab andmebaasisüsteemile infot tabelite vaheliste seoste olemasolust ning võimaldab andmebaasisüsteemil eeldada, et teatud andmeid andmebaasis kindlasti ei ole (sest need andmed oleksid vastuolus deklareeritud kitsendustega).

Antud näide demonstreerib ühtlasi, miks ei tasu kuuendal normaalkujul olevate tabelite peale ehitatud vaadetest pärida infot alljärgnevas formaadis (kui see just teostatava päringu eesmärk pole):

```
SELECT
      *
FROM view ...
WHERE ...
ORDER BY ...
```

Pärides infot SQL-markeri tärni (*) abil, ei heideta tabelite ühendamisoperatsioone kõrvale ning päring on tõenäoliselt väga ressursinõudlik.

1.5. Sisula

Sisula (*simple substitution language*) on mallimootor, mis loodi ankurmodelleerimise projekti raames, lihtsustamaks SQL-koodigeneraatori loomist või tõlkimist („sisula”). Mootor on kirjutatud, kasutades skriptimiskeelt Javascript. Analoogselt ankurmodelleerimise rakenduse endaga on tegemist vabatarkvaraga.

Ankurmodelleerimise mudeli objekte hoitakse brauseri mälus XML-formaadis (*Extensible Markup Language*). Sisula tõlgib objektid JSON-formaati (*JavaScript Object Notation*), avab mallifailid ning seejärel rakendab defineeritud hulga reegleid. Pärast reeglite rakendamist valmib soovitud kood.

Joonisel 7 on toodud fragment Sisula lähtekoodist. Koodi lähemalt sirvides on näha, et lõpliku genereerimistulemuse koostamiseks teostatakse iteratsioon üle mallifailide (*for-*

tsükkel) ning iga faili iga rea kohta rakendatakse teatud hulk reegleid (*replace*-käsk), mis mallides olevad muutujad reaalsete objektinimedega asendavad.

```
sisulate: function(xml, map, directive) {
    // objectify the xml
    var schema = Sisulator.objectify(xml, map).schema;
    // this variable holds the result
    var _sisula_ = '';
    // process and evaluate all sisulas in the directive
    var reader = new ActiveXObject("Scripting.FileSystemObject");
    var file = reader.OpenTextFile(directive);
    var scripts = file.ReadAll();
    file.Close();
    scripts = scripts.replace(/\r/g, ''); // unify line breaks
    // only non-empty lines that are not comments (starts with #)
    scripts = scripts.match(/^[^#].+/gm);
    var script;
    var splitter = /\/\*~|~\*\//g; // split JS /*~ sisula template ~*/ JS
    // process every sisula
    for(var s = 0; script = scripts[s]; s++) {
        script = script.replace(/^\s+/,'').replace(/\s+$/,''); // trim
        file = reader.OpenTextFile(script);
        var sisula = file.ReadAll();
        file.Close();
        // make sure everything starts with JavaScript (empty row)
        sisula = '\n' + sisula;
        // split language into JavaScript and SQL template components
        var sisulets = sisula.split(splitter);
        // substitute from SQL template to JavaScript
        for(var i = 1; i < sisulets.length; i+=2) {
            // honor escaped dollar signs
            sisulets[i] = sisulets[i].replace(/[$]{2}/g, 'Â§DOLLARÂ§'); //
            sisulets[i] = sisulets[i].replace(/["]{2}/g, 'Â§QUOTEDÂ§'); //
            sisulets[i] = sisulets[i].replace(/[$]{([\S\s]*?)}[$]/g, '" + $
            sisulets[i] = sisulets[i].replace(/[$]\(([\S\s]*?)\)\?[^\S\n]*(
            sisulets[i] = sisulets[i].replace(/[\$]([\w.]*?)(?:([\$])|([^\w
            sisulets[i] = sisulets[i].replace(/(\r\n|\n|\r)/g, '\\n" +\n');
            sisulets[i] = sisulets[i].replace(/^/gm, '"'); // start of line
            sisulets[i] = '_sisula_+=' + sisulets[i] + '";'; // variable as
        }
```

Joonis 7. Sisula on kirjutatud Javascriptis

Andmebaasiobjektide loomiseks vajaliku SQL-koodi genereerimiseks saab koostada malle. Igas mallis on muutumatu osa ja mudelist sõltuv muutuv osa. Järgnevas mallis on muutuv osa tähistatud rasvase fondiga.

```
-- Knot identity table ------------------------------------
-- $knot.identityName table
-------------------------------------------------------------
IF Object_ID('$knot.capsule$.$knot.identityName', 'U') IS NULL
CREATE TABLE [$knot.capsule].[$knot.identityName] (
    $knot.identityColumnName $knot.identity $knot.identityGenerator not
null,
    $(schema.METADATA)? $knot.metadataColumnName
$schema.metadata.metadataType not null, : $knot.dummyColumnName bit null,
    constraint pk$knot.identityName primary key (
        $knot.identityColumnName asc
    )
);
GO
```

Antud koodinäite baasil saab Sisula teisenduste abil genereerida sõlme (*knot*) loomiseks vajaliku SQL-koodi andmebaasisüsteemi MS SQL Server jaoks. Genereeritud kood on malliga väga sarnane ning näeb välja alljärgnev:

```
-- Knot table ---------------------------------------------
-- SRO_StyleRole table
-------------------------------------------------------------
IF Object_ID('dbo.SRO_StyleRole', 'U') IS NULL
CREATE TABLE [dbo].[SRO_StyleRole] (
    SRO_ID smallint IDENTITY(1,1) not null,
    SRO_StyleRole varchar(100) not null,
    Metadata_SRO int not null,
    constraint pkSRO_StyleRole primary key (
        SRO_ID asc
    ),
    constraint uqSRO_StyleRole unique (
        SRO_StyleRole
    )
);
GO
```

Võrreldes malli ning genereeritud koodi, on näha, et muutujad tähistatakse mallis dollarimärgiga ($). Fraasist

`CREATE TABLE [$knot.capsule].[$knot.identityName]`

saab fraas

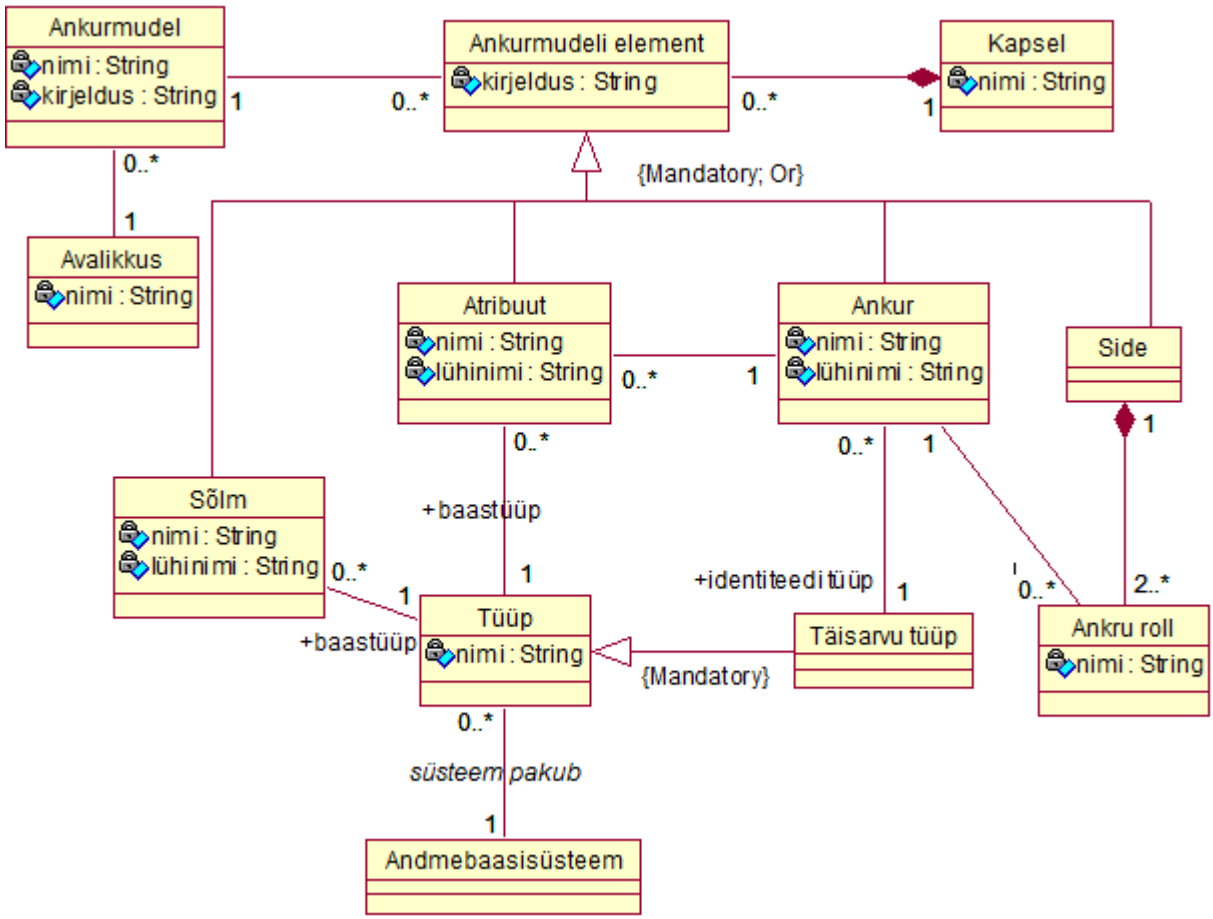
`CREATE TABLE [dbo].[SRO_StyleRole].`

Samuti saab kasutada mallis kõiki Javascripti konstruktsioone (*if*, *while* jne).

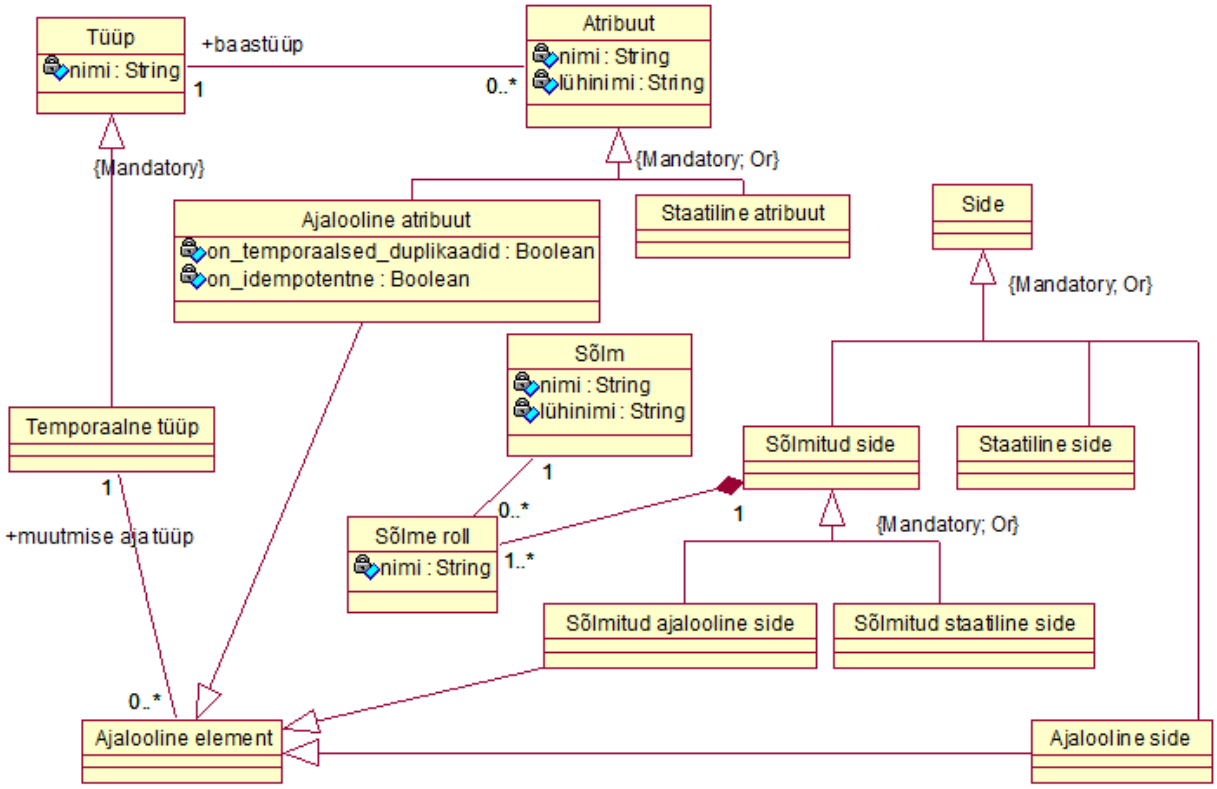
Sisula lihtsustab märgatavalt ankurmodelleerimise rakendusele teiste andmebaasisüsteemide toe lisamist, sest koodi konverteerimisega tegelev isik saab mugavalt koostada malle, mis on ka teistele võimalikele kasutajatele lihtsasti mõistetavad.

1.6. Ankurmodelleerimise põhimõisted

Järgnevalt esitatakse ankurmodelleerimise valdkonnamudelid (joonised 8 ja 9), mis kirjeldavad selle põhimõisteid ja mõistete vahelisi seoseid.



Joonis 8. Ankurmodelleerimise valdkonnamudel



Joonis 9. Ankurmodelleerimise valdkonnamudel (osa 2)

2. Koodigeneraatori loomine PostgreSQL jaoks

Selles peatükis kirjutatakse koodigeneraatori loomise protsessist ning selle käigus tehtud tähelepanekutest.

2.1. Taust

Käesoleva magistritöö eesmärgiks on lisada ankurmodelleerimise veebipõhisele vahendile (http://www.anchormodeling.com/modeler/latest/) võimalus SQL-koodi genereerimiseks PostgreSQL jaoks. Lähtuvalt autori üldistest kogemustest PostgreSQLiga alustatakse tööd eeldusega, et see on võimalik („PostgreSQL 9.3.5 Documentation").

Püüdes tuvastada ja kaardistada, milliseid tegevusi on PostgreSQLi toe lisamiseks vaja läbi viia, avastas autor, et dokumentatsiooni, mis üheselt kirjeldaks, kuidas ankurmudeli elementidest luua SQL-andmebaasi objekte, põhimõtteliselt ei eksisteeri, kui välja arvata üks lühiartikkel (Rönnbäck et al., 2010c). Samuti puudus info selle kohta, kas ja milliseid kasutajaliidese muudatusi oleks vaja teostada. Autor leidis mõned teadusartiklid (Rönnbäck et al., 2010a), mis kirjeldavad ankurmodelleerimise printsiipe ning demonstreerivad näitemudeli baasil mõningaid baastabeleid, ühte vaadet ja ühte perspektiivi. Puudub aga selge määratlus, millised tabelid, võtmed, vaated, trigerid, kitsendused ja funktsioonid tuleb luua, ning mis on nende eesmärk. Seega on parimaks lähenemiseks MS SQL Serveri jaoks genereeritud koodi uurimine ning üritada sealtkaudu mõista, milline on loodavate andmebaasiobjektide semantika ning kuidas tuleks need PostgreSQLis realiseerida.

PostgreSQLi toe lisamisel võttis autor aluseks MS SQL Serveri („Microsoft SQL Server") jaoks loodud mallid, püüdes mallikoodi ja nende abil genereeritut lugedes mõista, mis on koodilõikude eesmärk ning millised andmebaasiobjektid peaksid tekkima. Samuti sai autor privaatvestluse vormis küsida selgitavaid küsimusi Lars Rönnbäckilt, kes on ankurmodelleerimise teooria üks rajajatest.

2.2. Loomise protsessi kirjeldus

Generaatori täiendamine eeldab autorilt alljärgnevaid oskuseid (tärniga on märgitud oskused, mida on vaja ankurmodelleerimise veebirakendusele mõne teise andmebaasisüsteemi toe lisamiseks):

- andmebaasisüsteemi (nt PostgreSQL) põhjalik tundmine, sealhulgas ka protseduuride kirjutamiseks mõeldud keele (nt PL/pgSQL) tundmine,

- * andmebaasipäringute kirjutamiseks mõeldud SQL andmekäitluskeele tundmine,
- * programmeerimisoskus HTML keeles,
- * programmeerimisoskus Javascript keeles,
- * objektorienteeritud programmeerimisvõtete tundmine,
- programmeerimisoskus PHP keeles,
- Apache HTTP serveri haldamise oskus,
- * versioonihaldussüsteemi Git tundmine,
- * inglise keele oskus.

Eesmärgi täitmiseks ja täitmise võimalikkuse uurimiseks teostatakse järgmised sammud.

- Luuakse kontakt ühega ankurmodelleerimise teooria autoritest.
- Luuakse kontseptuaalne andmebaasi disain, mis baseerub bakalaureusetöös (Saal, 2010) kasutatud andmebaasi disainil. Tegemist on raadiojaama esitlusloendi lihtsustatud andmebaasiga, kus hoitakse andmeid muusikapalade, artistide, albumite, autoritasude ja lugude esitamise järjekorra kohta.
- Modelleeritakse andmebaas, kasutades ankurmodelleerimise veebirakendust. Andmebaasi disain on valitud selline, et kasutusel oleksid kõik (tabelis 1 toodud) ankurmodelleerimise konstruktsioonid.
- Loodud mudelit aluseks võttes genereeritakse (olemasolevat generaatorit kasutades) kood MS SQL Server jaoks.
- Lisatakse ankurmodelleerimise rakendusele võimalus genereerida koodi PostgreSQLi jaoks. Selle teostamiseks muudetakse rakenduse menüü struktuuri, koostatakse vajalike mallide loetelu ning programmeeritakse magistritöös kasutusel olevate andmetüüpide vastavustabel.
- Kaardistatakse ankurmodelleerimise mudelist koodi genereerimise tulemusena loomist vajavad andmebaasiobjektid, selgitatakse nende eesmärke ja toimeloogikat.
- Üritatakse tõlkida MS SQL Serveri jaoks genereeritud kood PostgreSQLile sobivaks. Selle käigus luuakse ka mallifailid, mis väljastavad tõlgitud koodile vastava tulemi.
- Kirjutatakse PHP-rakendus, mis täidab genereeritud tabelid testandmetega ning proovib simuleerida andmete lisamist ja otsimist ning funktsioonide ja trigerite käivitamist, püüdmaks leida vigu tõlgitud koodis.
- Kaardistatakse võimalikud kitsaskohad ja takistused.

- Tuuakse esile kohad, kus lahendusi oli rohkem kui üks.
- Antakse soovitusi, mida järgnevad magistritööde kirjutajad uurida võiks.

Praktilise osa läbiviimiseks kasutatakse personaalarvutit alljärgneva konfiguratsiooniga:

- protsessor: Intel® Core™ i5-3570K kiirusega 4.6 Ghz,
- vahemälu: 8GB G.SKILL Ares kiirusega 2100 Mhz,
- kõvaketas: WD 500 GB,
- operatsioonisüsteem: Windows 7 Ultimate SP1 64-bit,
- andmebaasisüsteem: PostgreSQL 9.3.3,
- andmebaasi haldustarkvara pgAdmin 1.18,
- Wampserver arendusraamistik (Apache 2.2/PHP 5.3),
- tarkvara arendusraamistik Eclipse Helios.

2.3. Nõuded

Loodav generaator peaks vastama alljärgnevatele tingimustele:

- võttes aluseks raadiojaama esitlusloendi kontseptuaalse andmemudeli, luuakse ankurmodelleerimise vahendit kasutades mudel, mis sisaldab kõiki võimalikke ankurmodelleerimise konstruktsioone,
- eelmainitud mudeli alusel genereeritud SQL-koodi on ilma kriitiliste vigadeta võimalik käivitada andmebaasisüsteemis PostgreSQL (versioonis 9.3, mis oli töö kirjutamise ajal kõige viimane ametlik versioon),
- koodi käivitamisel tekivad tabelid, vaated, trigerid, funktsioonid, võtmed ja kitsendused,
- tabelite struktuur on sarnane MS SQL Serveri jaoks genereeritud tabelitega,
- tabelitesse on võimalik lisada andmeid,
- funktsioonide ja vaadete poolt tagastatud andmed on võimalikult sarnased MS SQL Serveri jaoks genereeritud funktsioonide ja vaadete poolt tagastatavate andmetega.

2.4. Kontakti loomine ühe ankurmodelleerimise teooria rajajaga

Käesoleva töö autori esimene kokkupuude ankurmodelleerimise teooria ühe rajaja Lars Rönnbäck'iga oli juba 2014. aasta märtsis, kui autor uuris ankurmodelleerimise rakendust

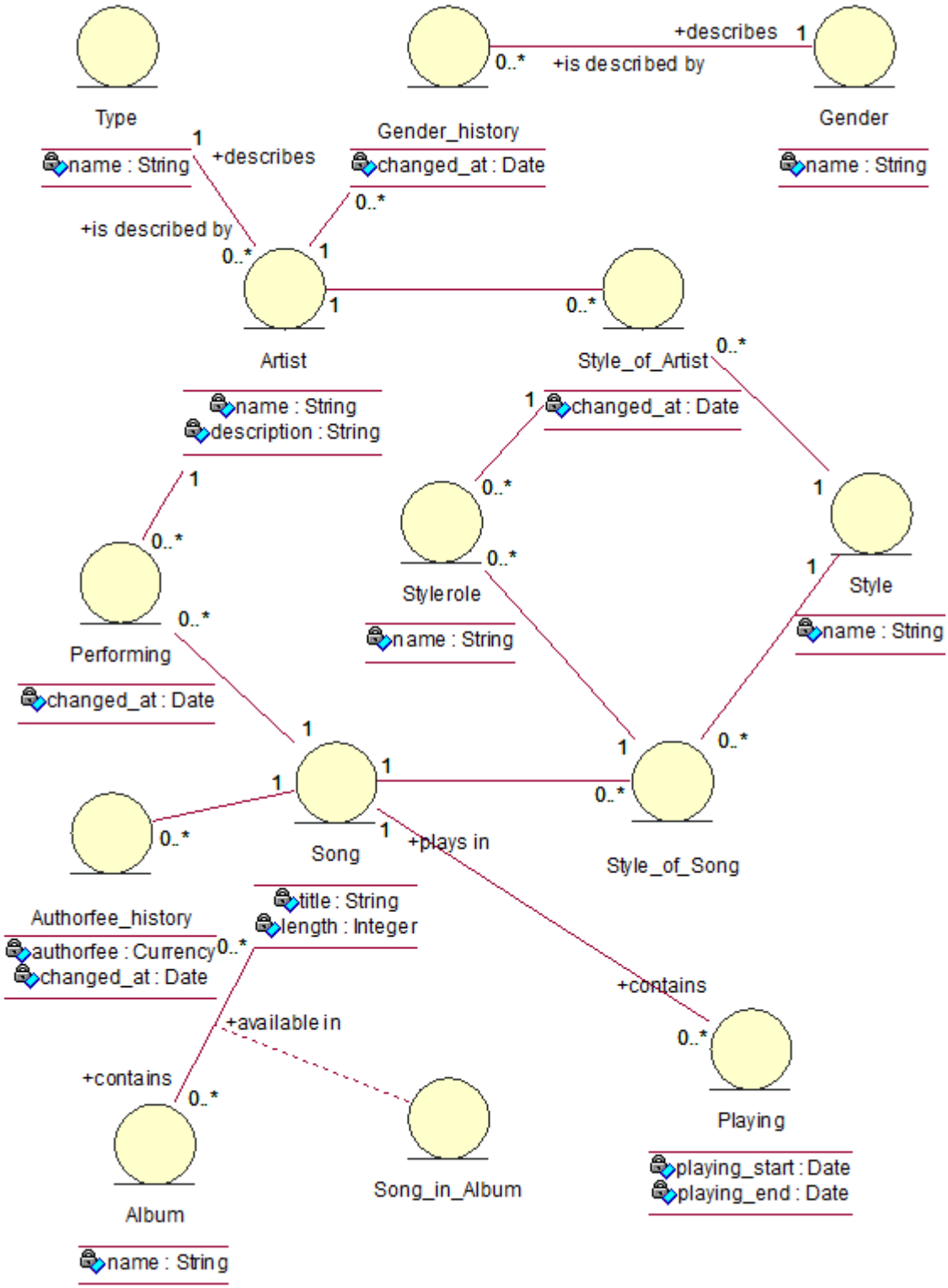
ning sellele PostgreSQLi toe lisamise teoreetilist võimalikkust. Uurimise käigus leidis autor ankurmodelleerimise rakenduse lähtekoodist vea, mis teatud tingimustel takistas koodi genereerimist. Vea kirjeldus ning parandusettepanek sai e-kirja teel edastatud ka Larsile, kes paranduse rakendusele lisas. Lisaks kirjutas Lars, et huvi korral oleks autor teretulnud rakendust täiendama.

2014. aasta oktoobri alguses võttis töö autor taas Larsiga ühendust ning teavitas, et proovib generaatorile lisada PostgreSQLi toe. Lars võttis uudise soojalt vastu ning kinnitas, et eduka projekti korral paneb ta uuendatud generaatori ankurmodelleerimise avalikule veebilehele üles.

Töö kirjutamise ajal vahetasid töö autor ning Lars Rönnbäck mitmeid e-kirju ning pidasid ka mitmeid Skype-vestlusi, lahendamaks tekkinud kitsaskohti ning saamaks paremini aru mõningatest rakendusega seotud nüanssidest. Võib öelda, et töö teostamise käigus sai autor rakenduse ekspertkasutajaks.

Samuti sai käesoleva töö autor ligipääsud ankurmodelleerimise rakenduse lähtekoodile keskkondades Google Code (https://code.google.com) ja Github (https://github.com). Töö kirjutamise ajal tegi autor üle 40 komplekti täiendusi (*commit*) ankurmodelleerimise rakenduse koodile. Projekti koduleht Github keskkonnas asub aadressil https://github.com/Roenbaeck/anchor. PostgreSQLi toe lisamiseks tegi autor haru (*branch*), mis on ligipääsetav aadressil https://github.com/Roenbaeck/anchor/tree/Elari.Postgresql.

2.5. Kontseptuaalne andmemudel



Joonis 10. Tõlkeprotsessis kasutatud infosüsteemi olemi-suhte diagramm

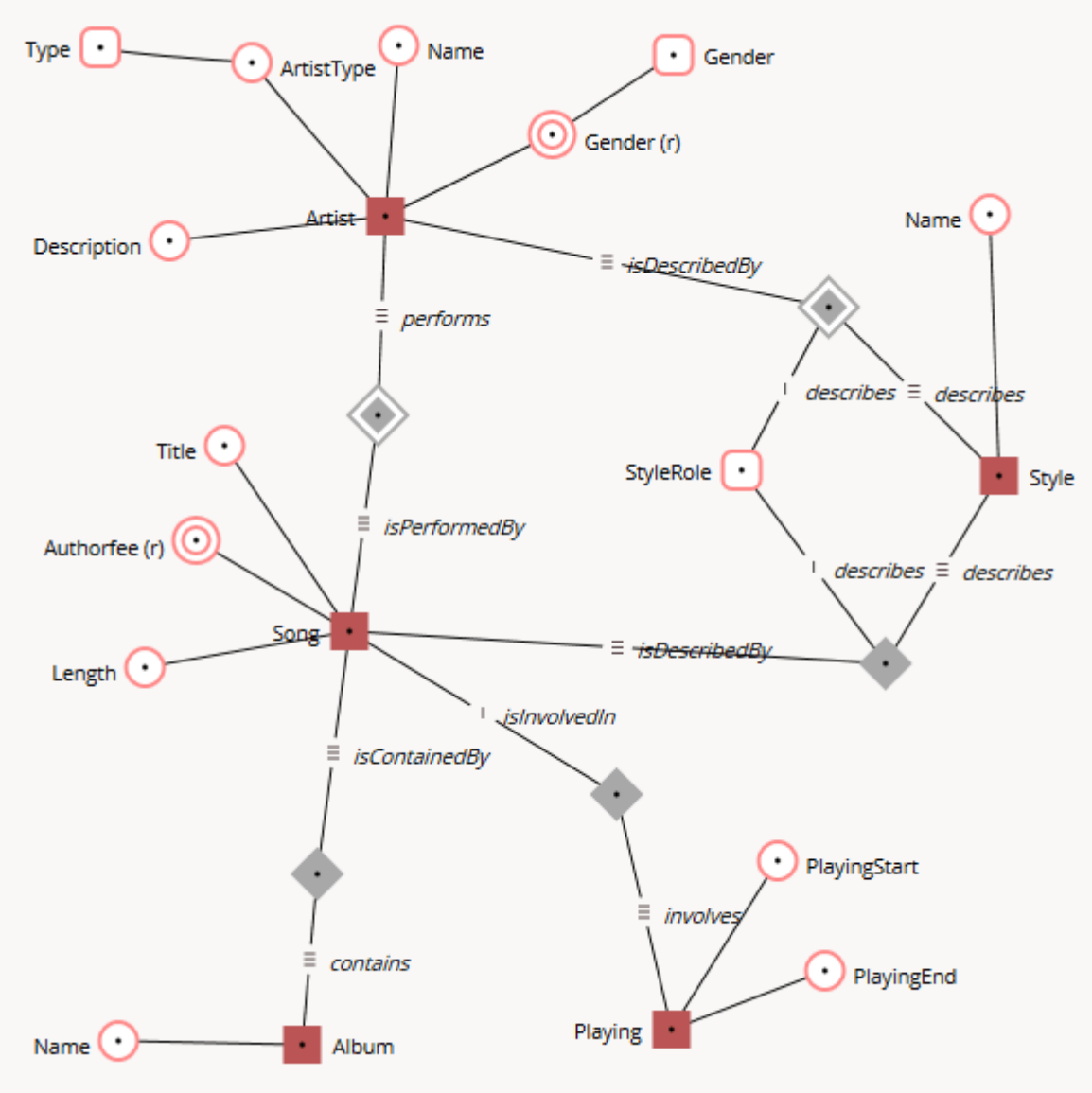
Joonisel 10 on toodud raadiojaama esitlusloendi lihtsustatud infosüsteemi üks võimalikest kontseptuaalsetest andmemudelitest, mis on joonistatud üles UML

klassidiagrammina. Mudel on valitud selline, et selle alusel genereeritud koodis oleksid kajastatud kõik võimalikud ankurmodelleerimise konstruktsioonid. Olemitüüpide lühikirjeldused esitatakse tabelis 2.

Tabel 2. Olemitüübid ja nende kirjeldused

Olemitüübi nimi	**Lühikirjeldus**
Song	Lugu. Muusikapala, mida mängitakse raadiojaama eetris ning millede hulk moodustab raadiojaama esitlusloendi. Lool on pealkiri, autoritasu määr ja pikkus.
Artist	Esitaja. Bänd või sooloartist, mis või kes on loo esitajaks. Esitajal on tüüp, sugu, nimi ja kirjeldus.
Album	Album. Kogumik, mis seob teatud hulga muusikapalasid. Albumil on nimi.
Song_in_Album	Kuulumine albumisse. Seos loo ja albumi vahel.
Style	Muusikastiil. Kategoriseerib lood või esitajad erinevatesse stiilidesse. Stiilil on nimi.
Playing	Esitlusloend. Eetris kõlanud lugude ajalugu, mis moodustab esitlusloendi. Esitlusloendil on loo tunnus, algusaeg ja lõpuaeg.
Type	Tüüp. Artisti tüüp. Tüübil on nimi (bänd, sooloartist, orkerster jne).
Gender	Sugu. Artisti sugu. Sool on nimi. Sugu võib ajas muutuda.
Style_of_Artist	Artisti stiil. Kirjeldab, millist stiili artist viljeleb, spetsifitseerides ka selle, kas stiil on primaarne või sekundaarne.
Performing	Esitamine. Seos artisti ja loo vahel.
Stylerole	Stiiliroll. Kirjeldab, kas stiili puhul on tegemist primaarse või sekundaarse stiiliga.
Style_of_Song	Laulu stiil. Kirjeldab loo stiili, spetsifitseerides ka selle, kas stiil on primaarne või sekundaarne.

2.6. Ankurmodelleeritud mudel



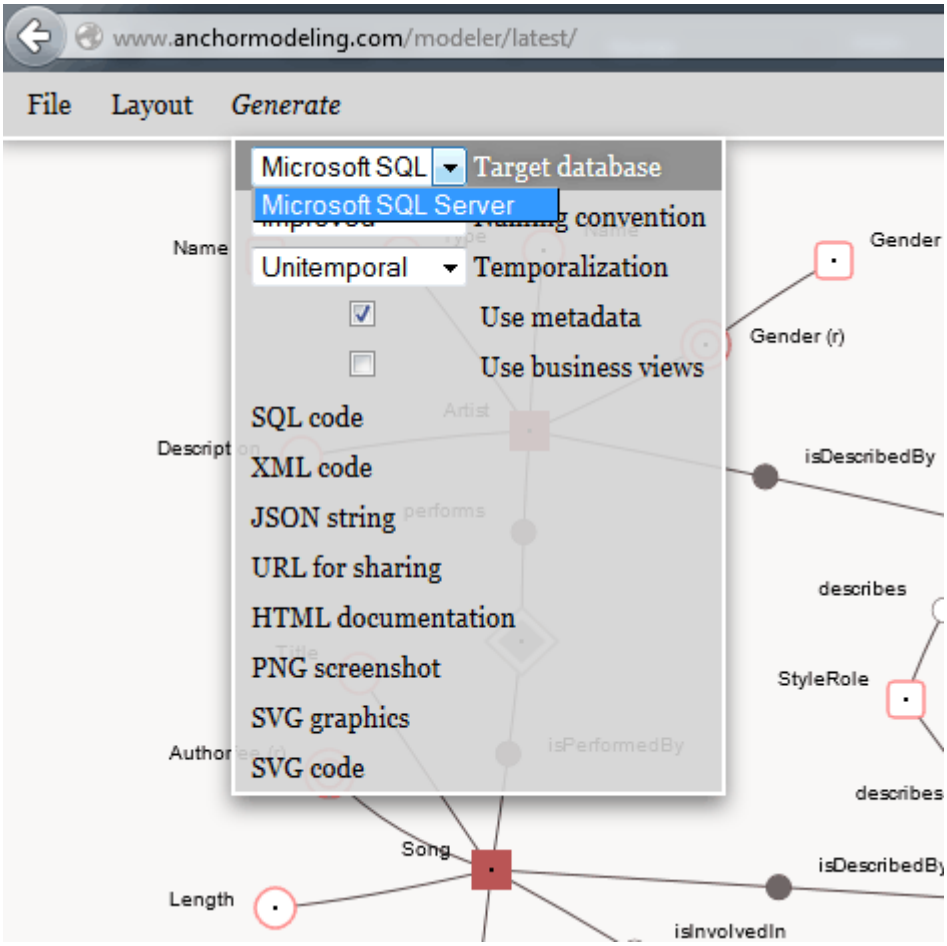
Joonis 11. Kontseptuaalsele andmemudelile vastav ankurmodelleeritud mudel

Võttes aluseks joonisel 10 toodud kontseptuaalse andmemudeli, saab luua ankurmodelleeritud mudeli (joonis 11). Tasub esile tuua, et joonised 10 ja 11 on küllaltki sarnased. Erinevus on mudeli elementide visuaalses kujus ja värvis. Põhiline sisuline erinevus seisneb selles, et joonisel 11 on atribuudid olemitest „välja toodud". Samuti on seoseid esindavad olemitüübid asendunud sidedega (*tie*). Tabelis 3 on toodud joonisel 11 kujutatud elementide tüübid.

Tabel 3. Ankurmodelleerimise elemendid

Elemendi nimi	**Tüüp**
Song	Ankur Atribuudid: - pealkiri (staatiline atribuut) - autoritasu määr (ajalooline atribuut) - pikkus (staatiline atribuut)
Artist	Ankur Atribuudid: - tüüp (sõlmitud staatiline atribuut) - sugu (sõlmitud ajalooline atribuut) - nimi (staatiline atribuut) - kirjeldus (staatiline atribuut)
Album	Ankur Atribuudid: - nimi (staatiline atribuut)
Song_in_Album Song_in_Playing	Staatilised sidemed
Style	Ankur Atribuudid: - nimi (staatiline atribuut)
Playing	Ankur Atribuudid: - algusaeg (staatiline atribuut) - lõpuaeg (staatiline atribuut)
Style_of_Artist	Sõlmitud ajalooline side
Performing	Ajalooline side
Stylerole Gender Type	Sõlmed
Style_of_Song	Sõlmitud staatiline side

2.7. Koodi genereerimine MS SQL Server jaoks



Joonis 12. Koodi genereerimine MS SQL Server jaoks

Joonisel 12 toodud ekraanitõmmis näitab, et töö kirjutamise ajal võimaldas ankurmodelleerimise vahend (http://www.anchormodeling.com/modeler/latest/) SQL-koodi genereerida ainult andmebaasisüsteemi MS SQL Server jaoks. Kehtiv generaatori versioon sel hetkel oli 0.97.6 (*rev: 624, release: Wednesday the 9th, April, 2014*). Koodi genereerimiseks kasutatud menüüsätted on lahti seletatud allpool. Autor tahab rõhutada, et need sätted ei ole dokumenteeritud ning selgitused neile saadi Lars Rönnbäckilt privaatvestluse vormis.

- *Target database: Microsoft SQL Server.*
  - Selgitus: andmebaasisüsteem, mille jaoks koodi genereerima asutakse.
  - Valikuvõimalused: Microsoft SQL Server.
  - Vaikimisi valik: Microsoft SQL Server.
- *Naming convention: Improved.*
  - Selgitus: genereeritud andmebaasiobjektide puhul jälgitavad nimetamise reeglid (Rönnbäck et al. 2010b).

    - Valikuvõimalused: *Original* (algselt loodud reeglid), *Improved* (uuendatud reeglid, mis parandavad mõned korduvate veerunimedega seotud vead).
    - Vaikimisi valik: *Improved*.
- *Temporalization: Unitemporal*.
    - Selgitus: temporaalsus; määrab ajalooliste andmete käsitluspõhimõtte (Rönnbäck, 2010).
    - Valikuvõimalused: *Unitemporal* (monotemporaalne generatsioon; jälgitakse väärtuste muutmise aega, ehk *changing time*), *Concurrent-reliance-temporal* (bitemporaalne generatsioon; jälgitakse eraldi väärtuste lisamise ja muutmise aega ehk *recording time* ja *changing time*).
    - Vaikimisi valik: *Unitemporal*.
    - Märkus: bitemporaalne generatsioon on dokumenteerimata. Lars Rönnbäcki hinnangul valmib teadusartikkel bitemporaalse generatsiooni kohta 2015. aasta esimesel poolel.
- *Use metadata*: sisse lülitatud.
    - Selgitus: genereeritud baastabelite *metadata* (ehk metaandmete) veeru igas väljas hoitakse (üldjuhul täisarvuliste väärtustena) viidet mingis teises tabelis (ning enamasti teises andmebaasis) olevale andmereale, mis esitab infot selle kohta, kuidas vaadeldav baastabeli rida andmebaasi „jõudis". Andmeaitades hoitakse tihti metaandmeid selle kohta, milline kasutaja millisel kuupäeval/kellaajal millist protsessi käivitades rea(d) tabelitesse lisas. Metaandmete veerus saab hoida selliseid viiteid.
    - Valikuvõimalused: sisse või välja lülitatud.
    - Vaikimisi valik: sisse lülitatud.
    - Märkus: Lars Rönnbäck teatas autorile, et ankurmodelleerimise rakenduse järgmises versioonis saab vaikimisi valik tõenäoliselt olema „välja lülitatud", sest metaandmete veerg tekitab generaatoriga tutvujates segadust.
- *Use business views*: välja lülitatud.
    - Selgitus: „ärivaadete" puhul on tegemist „lihtsustatud" perspektiividega, kust on eemaldatud „tehnilised" veerud (nagu identifikaatorid ja metaandmete veerud). Paljud kasutajad leidsid, et perspektiivid, mis on denormaliseeritud vaated ankrutele ja nende atribuutidele, on liiga mahukad ning liiga paljude veergudega, seetõttu lisati rakendusele lihtsamad „ärivaated".
    - Valikuvõimalused: sisse või välja lülitatud.

    - Vaikimisi valik: välja lülitatud.
- *Use equivalence*: välja lülitatud.
    - Selgitus: Lars Rönnbäcki sõnutsi on ekvivalentsi puhul tegemist hiljuti lisatud (ning kehvasti dokumenteeritud) lisaga, mis võimaldab andmebaasis hoida väärtusi, millel on sama tähendus. Näiteks „Roheline", „Green" ja „Grün" tähendavad erinevates keeltes sama asja.
    - Valikuvõimalused: sisse või välja lülitatud.
    - Vaikimisi valik: välja lülitatud.

Valides menüüst „*SQL code*", genereeritakse ning kuvatakse kasutajale andmebaasiobjektide loomiseks vajalik SQL kood (näide joonisel 6). Kood (koos selles sisalduvate kommentaaridega) on liiga pikk, et seda siin ära tuua, aga ülevaatlikult:

- genereeritud koodi pikkus on töös kasutusel oleva mudeli korral 3624 rida,
- genereeritud koodi maht on 137 KB,
- genereeritud koodis on 25 tabelit,
- genereeritud koodis on 24 vaadet,
- genereeritud koodis on 26 funktsiooni,
- genereeritud koodis on 23 trigerit,
- genereeritud koodis on 0 CHECK kitsendust,
- genereeritud koodis on 0 tühja rida,
- genereeritud koodis on 323 rida kommentaare.

Autor otsustas, et käesoleva töö raames ei realiseerita bitemporaalsust (*Concurrent-reliance-temporal)*, ekvivalentsi (*equivalence*) ja ärivaateid (*business views*), sest nende põhimõtted olid töö kirjutamise hetkel liiga halvasti dokumenteeritud. Ühtlasi on see hea näide, kuidas dokumentatsiooni ebapiisavus hakkab takistama süsteemi edasiarendamist.

2.8. Kasutajaliidese muudatused

Ankurmodelleerimise veebilehel (http://www.anchormodeling.com/modeler/latest/) avalikuks kasutamiseks mõeldud modelleerimisvahend ja sellega kaasaskäiv generaator võimaldasid töö kirjutamise ajal genereerida koodi ainult ühe andmebaasisüsteemi jaoks. Vahendi lähtekoodi sirvides jäi käesoleva magistritöö autorile esialgu mulje, et ankurmodelleerimise rakendust

luues polnud rakenduse loojad arvestanud mõttega, et rakendusel võiks olla mitme andmebaasisüsteemi tugi.

Peale ankurmodelleerimise paradigma autoriga (Lars Rönnbäck) ühenduseloomist ning vajalike ligipääsuõiguste hankimist ankurmodelleerimise projektile keskkondades Google Code ja Github sai käesoleva töö autor enda käsutusse lähtekoodi, kuhu oli juba osaliselt lisatud andmebaasisüsteemi Oracle tugi. Samuti oli tehtud hulk toetavaid arendusi, mis võimaldasid koodi genereerida rohkem kui ühe andmebaasisüsteemi jaoks. Näiteks oli rakendusele lisatud andmetüüpide konverteerimise võimalus, mis osutub vajalikuks, kui loodud mudeli alusel tahaks koodi genereerida erinevate andmebaaside jaoks.

Võttes aluseks pooleliolev Oracle toe lisamiseks äratehtu, ning seda PostgreSQL jaoks mugandades, lisas autor rakendusele võimaluse koodi genereerimiseks PostgreSQLi jaoks. Selleks tuli teha järgnevad sammud:

- lisada andmebaasisüsteemi PostgreSQLi valik rakenduse menüüsse,
- programmeerida rakendusele MS SQL Server ja PostgreSQL andmetüüpide mittetäielik vastavustabel, võimaldamaks mudelis kasutatud andmetüüpide asendamist, kui rakenduse menüüs vahetada andmebaasisüsteemi, mille jaoks koodi genereerima hakatakse,
- luua mallide loetelu, mis kirjeldab, milliseid malle ning millises järjekorras töödeldakse,
- luua (esialgu tühjad) mallid,
- luua üldised PostgreSQLi konfiguratsioonifailid, mis kirjeldavad muuhulgas aja käsitlust (näiteks MS SQL Serveris saadakse vaadeldava hetke aeg ilma ajavööndita funktsiooniga SYSDATETIME(), PostgreSQLis saab kasutada funktsiooni LOCALTIMESTAMP(0)).

2.9. Koostöövõimalus

Peale mallidest umbes poolte üleviimist PostgreSQLile vastavasse formaati avastasid autor ja ankurmodelleerimise teooria rajaja Lars Rönnbäck, et üks Ameerika Ühendriikide kodanik hakkas iseseisvalt ja kellelegi ütlemata lisama generaatori bitemporaalsele ehk *concurrent-reliance-temporal*-generatsioonile PostgreSQLi tuge, võttes aluseks käesoleva töö autori (pooleldi) tõlgitud mallid. Peale kirjavahetust antud isikuga saavutati kokkulepe, et käesoleva töö autor tegeleb *unitemporal*-generatsiooniga ja ameeriklane *concurrent-reliance-temporal*-generatsiooniga.

Autor, juhendaja Erki Eessaar ja Lars Rönnbäck nägid seda koostööd kui võimalust kiirendada generaatorile PostgreSQLi toe lisamist, kuid kahjuks peale kuu möödumist oli näha, et *concurrent-reliance-temporal*-generatsiooni tõlkimistööd on seisma jäänud, ning kahjuks poldud sellega väga kaugele jõutud. Küll aga näitab see, et ka mujal maailmas on huvi ankurmodelleerimise (ja sellele PostgreSQLi toe lisamise) vastu kõrge.

3. Teisendusreeglid

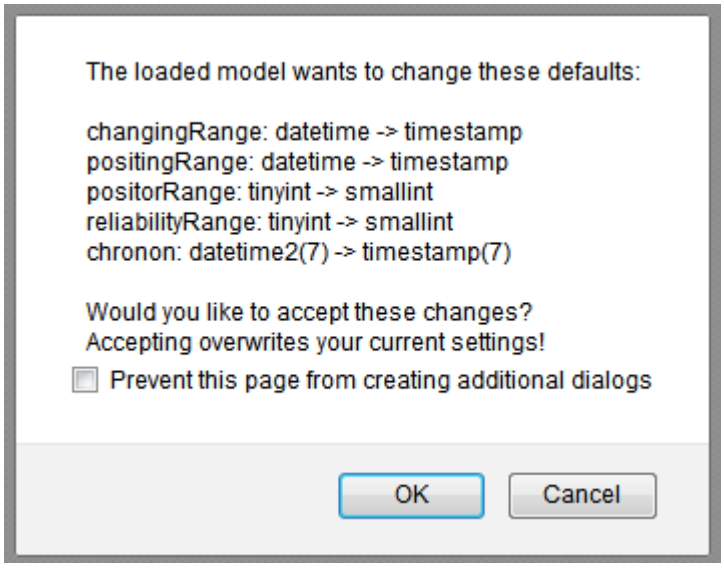
Järgnevalt tuuakse näited, kuidas ankurmudeli elementidest saab PostgreSQL kood.

3.1. Andmetüübid

Tabelis 4 on toodud käesolevas magistritöös kasutusel olevate andmetüüpide mittetäielik vastavustabel, mis on vajalik, et sama ankurmudeli alusel saaks genereerida koodi nii MS SQL Serveri kui PostgreSQLi jaoks.

Tabel 4. Andmetüübid

MS SQL Server	**PostgreSQL**
tinyint	smallint
number(n)	numeric(n)
datetime	timestamp
datetime2(n)	timestamp(n)
varchar	char(1)
varbinary(16)	bytea
IDENTITY(1,1)	smallserial serial bigserial



Joonis 13. Andmetüübiteisenduste kinnitamine

Kuigi kasutajaliides seda intuitiivselt ei esita, saab loodud mudeli alusel genereerida koodi mitme erineva (toetatud) andmebaasisüsteemi jaoks.

Kui ankurmodelleerimise rakenduse menüüs vahetada andmebaasisüsteemi valikut, mille jaoks koodi genereerima hakatakse, siis küsib rakendus kasutajalt üle, kas kasutaja soovib teha ka andmetüübiteisendused (joonis 13). Selleks tuleb ankurmodelleerimise rakenduses teostada tarkvaraarendus, lisamaks generaatori „tuumale" andmetüübiteisenduste reeglid. Järgnevalt on toodud autori poolt kirjutatud koodifragment, mis antud teisendusi läbi viib:

```
var DataTypeConverter = {
    MATCH: 0,
    REPLACE: 1,
    SQLServer_to_PostgreSQL: [
      [/"tinyint"/ig,                          '"smallint"'],
      [/"NUMBER\(([0-9]+)\)"/ig,               '"numeric($1)"'],
      [/"datetime"/ig,                         '"timestamp"'],
      [/"datetime2\(([0-9]+)\)"/ig,            '"timestamp($1)"'],
      [/"varchar"/ig,                          '"char(1)"'],
      ...
],
```

Kuna töö kirjutamisel ajal tegelesid mõned inimesed ka generaatorile Oracle toe lisamisega, siis peaks generaatoris olema alljärgnevad tüübiteisendusvõimalused:

- võimalus asendada MS SQL Server andmetüübid PostgreSQL andmetüüpidega (olemas osaliselt),
- võimalus asendada MS SQL Server andmetüübid Oracle andmetüüpidega (olemas osaliselt),
- võimalus asendada PostgreSQL andmetüübid MS SQL Server andmetüüpidega (puudub),
- võimalus asendada PostgreSQL andmetüübid Oracle andmetüüpidega (puudub),
- võimalus asendada Oracle andmetüübid MS SQL Server andmetüüpidega (puudub),
- võimalus asendada Oracle andmetüübid PostgreSQL andmetüüpidega (puudub).

On ilmne, et iga kord, kui generaatorile lisatakse uue andmebaasisüsteemi tugi, tuleb teostada üksjagu uurimistööd ning kõikidele generaatori poolt toetatud andmebaasisüsteemidele tuleb lisada võimalus asendada nendes kasutatavad andmetüübid kõigi teiste andmebaasisüsteemi andmetüüpidega. Lühidalt öeldes, kui ankurmodelleerimise

rakendus toetab n andmebaasisüsteemi, siis peab rakenduses olema kirjeldatud n * (n - 1) komplekti tüüpide vastavusi. Samuti ei saa jätta arvestamata andmebaasisüsteemide evolutsiooniga. Andmebaasisüsteemide uutesse versioonidesse võidakse lisada uute tüüpide toetust (nt viimasel ajal väga populaarseks saanud JSON tüüp) ning väiksema tõenäosusega ka tüüpide toetust eemaldada. Sellisel juhul oleks tänu erinevatele andmebaasisüsteemide versioonidele iga toetatud andmebaasisüsteemi kohta vaja mitte üks, vaid mitu generaatorit.

3.2. PostgreSQL üldised kitsaskohad/erisused seoses ankurmudelite realiseerimisega

PostgreSQLis luuakse ankurmudeli elementide realiseerimiseks tabelid ning nendele ka identse sisuga vaated, sest PostgreSQLis pole tabelitele võimalik teha INSTEAD OF INSERT trigereid, mida läheb vaja tabelitesse lisatavate andmete kontrollimisel. Ankurmodelleerimise paradigma autori sõnutsi on vajalik andmeaita lisatavad (ajaloolised) andmed eelsorteerida, vältimaks (temporaalseid) duplikaate ning lihtsustamaks duplikaatide leidmist. Tabelitesse andmete lisamise asemel lisatakse andmed vaadetesse. Erisus kehtib kõikidele generaatori poolt loodavatele tabelitele.

Tuleb aru saada, et ankurmodelleerimine on algselt loodud andmeaitade loomiseks. Andmeaidas, kuhu laaditakse andmeid paljudest erinevatest allikatest, võib tõepoolest tekkida olukord, kus tabelisse üritatakse lisada duplikaate. Samuti põhjus, miks ankurmodelleerimise paradigma kasutab andmete eelsorteerimist, seisneb selles, et andmeaitades ei sisestata andmeid tabelitesse tüüpiliselt lausega

```
INSERT INTO tabel (veergude nimekiri) VALUES (väärtuste nimekiri);,
```

vaid

```
INSERT INTO tabel (veergude nimekiri) SELECT veergude nimekiri FROM mõni teine tabel;
```

ehk sageli miljoneid ridu korraga. Kui lisatavad andmed on ajalises mõttes juhuslikus järjekorras, on sisestuse käigus tabelisse uue rea lisamisel keerukas kontrollida, kas sisestatud väärtus erineb eelnevalt viimati sisestatud väärtusest.

Kui tabeli T veerule c on loodud klasterdatud indeks, siis on T read sorteeritud füüsiliselt c väärtuste alusel. Andmete muutmisel hoiab andmebaasisüsteem ise ridade järjekorda. Igas tabelis saab olla maksimaalselt üks klasterdatud indeks (sest sorteerida saab ühe veergude komplekti alusel). Selline indeks kiirendab otsimist veergude hulga alusel, mille

järgi read on sorteeritud. Read, kus nendes veergudes on ühesugused/sarnased väärtused, paiknevad füüsiliselt lähestikku.

PostgreSQLis pole klasterdatud indeksit, kuid tabelit saab indeksi alusel klasterdada. Seda operatsiooni tuleb perioodiliselt korrata, kuna erinevalt MS SQL Serverist ei hoia PostgreSQL klasterdamise järel ridade järjekorda ning uued/muudetud read pole enam järjekorras. Ridade järjestuse muutmiseks saab käivitada CLUSTER lauset. Eelnevalt tuleks aga iga loodava tabeli korral määrata, et klasterdamine toimub primaarvõtme indeksi alusel. Selleks tuleb genereerida ALTER TABLE lause. Antud lause annab võimaluse klasterdamist läbi viia, kuid ei kohusta selleks. Erisus kehtib kõikidele generaatori poolt loodavatele tabelitele.

3.3. Ankurmudeli elementide realiseerimine PostgreSQLis

3.3.1. Ettevalmistus

PostgreSQLis antakse funktsiooni käivitamise (EXECUTE) õigus vaikimisi kõigile kasutajatele (PUBLIC). Seega tuleb peale funktsioonide loomist võtta laialt avalikkuselt funktsiooni käivitamise õigus ning anda see õigus seda vajavatele kasutajatele. Selle õiguse vaikimisi kõigile kasutajatele (PUBLIC) andmine on võimalik ALTER DEFAULT PRIVILEGES lause abil keelata.

```
ALTER DEFAULT PRIVILEGES REVOKE EXECUTE ON FUNCTIONS FROM PUBLIC;
```

Kõik skeemiobjektid luuakse PostgreSQLis vaikimisi skeemis nimega *public*. Modelleerimisvahendis on võimalik määrata skeem (vahendi terminoloogias kapsel (*capsule*)), millesse skeemiobjektid paigutada. Modelleerimisvahend võimaldab teha nii, et mudelielementidest tekkivad skeemiobjektid paigutatakse erinevatesse skeemidesse. Erinevate skeemide tuge käesolevaks hetkeks loodud generaatoris ei ole. Küll aga saab määrata, et kõik objektid paigutuvad ühte kindlasse skeemi. Vastavalt modelleerimiskeskkonnas tehtud valikutele genereeritakse nt laused:

```
CREATE SCHEMA dbo;
SET search_path=dbo;
```

Teise lausega muudeti skeemiobjektide otsinguteed, millest tulenevalt pannakse kõik järgnevalt loodavad skeemiobjektid skeemi *dbo*.

3.3.2. Ankur

Iga ankur esitab mingit olemitüüpi. Järgnevalt esitatakse näide ankru realisatsioonist PostgreSQL andmebaasis:

```
CREATE TABLE IF NOT EXISTS _SO_Song (
    SO_ID serial not null,
    Metadata_SO int not null,
    constraint pkSO_Song primary key (
        SO_ID
    )
);
ALTER TABLE IF EXISTS ONLY _SO_Song CLUSTER ON pkSO_Song;
CREATE OR REPLACE VIEW SO_Song AS SELECT SO_ID, Metadata_SO FROM _SO_Song;
```

Vastavalt eeltoodule on näha, et luuakse tabel, öeldakse, et klasterdamine (kui seda kasutatakse) toimub primaarvõtmele automaatselt loodava indeksi alusel, ning luuakse ka tabeliga identne vaade. SERIAL notatsiooni abil luuakse automaatselt arvujada generaator ja seotakse see tabeli primaarvõtme veeruga, et saavutada iga uue rea korral sellesse veergu unikaalse täisarvu genereerimine.

Samuti luuakse kõikidele genereeritavatele tabelitele metaandmete veerg, kus saab hoida viidet informatsioonile, kuidas rida andmebaasi „jõudis". Veergu ei looda, kui enne koodi genereerimist eemaldada ankurmodelleerimise rakenduse menüüs valik „*Use metadata*".

Tabeli ALTER-lause ning vaate CREATE-lause genereeritakse igale ankurmudeli elemendi baasil tekkivale tabelile, ning järgnevates näidetes neid seetõttu ära ei tooda.

3.3.3. Sõlm

Iga sõlm esitab klassifikaatoriks olevat olemitüüpi. Järgnevalt esitatakse näide sõlme realisatsioonist PostgreSQL andmebaasis:

```
CREATE TABLE IF NOT EXISTS _TYP_Name (
    TYP_ID smallserial not null,
    TYP_Name varchar(100) not null,
    TYP_Checksum bytea not null,
    Metadata_TYP int not null,
    constraint pkTYP_Name primary key (
        TYP_ID
    ),
    constraint uqTYP_Name unique (
        TYP_Checksum
    )
);
```

Kitsaskohad/erisused võrreldes realisatsiooniga MS SQL Serveris:

- kui ankru näites oli primaarvõtme veeru puhul kasutatud *serial* notatsiooni (mille korral veeru tüübiks saab *integer*), siis sõlme puhul, mida definitsiooni järgi kasutatakse väikese hulga olemite esitamiseks, on primaarvõtme veeru korral kasutatud *smallserial* notatsiooni (mille korral saab veeru tüübiks *smallint*). Käesoleva töö autor lisas generaatori mallidesse koodiread, mis asendavad MS SQL Serveri korral kasutatavad võtmeveergude tüübid *int*, *smallint* ja *bigint* PostgreSQLi korral vastavate notatsioonidega *serial*, *smallserial* ja *bigserial*,
- ankurmudelit luues saab atribuudile (ja tema variatsioonidele) ning sõlmele lisada võimaluse selle väärtuste alusel räsi arvutamiseks. Räsiväärtuste alusel on võimalik efektiivselt tuvastada, kas sisestatav väärtus on juba tabelis olemas, sest räsi (sõltuvalt arvutamise meetodist) on üldjuhul kindla pikkuse ja formaadiga, erinevalt tekstilistest andmebaasiväljadest, mille sisu võib olla suvalises formaadis ja juhusliku pikkusega (näiteks 500 tühikut). Samuti pole MS SQL Serveri puhul teatud andmetüüpe võimalik omavahel võrrelda (nagu *geography*), kuid võrrelda saab nende baasil arvutatud räsisid. Tõsi küll, PostgreSQLis selliseid probleeme pole ning vajadusel saab defineerida uusi operaatoreid (sh võrdluse operaatoreid). Sõlme loomiseks genereeritud SQL-koodi sirvides on näha, et tabelis on veerg *TYP_Checksum*, kus saab hoida räsiväärtusi. Tabelisse sisestatud väärtuse põhjal räsi leidmiseks kasutab generaator levinud räsifunktsioni MD5. Kuigi on teada stringide paare, mis annavad ühesuguse MD5 räsiväärtuse, siis neid ei ole nii palju, et see takistaks räsiväärtuste abil otsingute kiirendamist. Kuna MD5 korral on räsiväärtuste kollisioonid e kokkupõrked harvad, siis luuakse räsiväärtuse veerule ikkagi ka unikaalsuse kitsendus (ka ankurmodelleerimise üks autoritest L. Rönnbäck toetab seda lähenemist),
- erinevalt MS SQL Serverist ei toeta PostgreSQL tabelites arvutatavaid veerge (veerge, milles olev väärtus arvutatakse dünaamiliselt välja teiste samas reas olevate väärtuste põhjal), mistõttu tuleb PostgreSQLis räsi arvutamiseks sõlme puhul kasutada BEFORE INSERT OR UPDATE trigerit, mis seotakse vastava tabeliga, kus arvutatud väärtusi hoida tahetakse. Järgnevalt on toodud sellise trigeri näide.

```
CREATE OR REPLACE FUNCTION tcsTYP_Name() RETURNS trigger AS '
    BEGIN
        NEW.TYP_Checksum = cast(
            substring(
                MD5(
                    cast(
```

```
                        cast(NEW.TYP_Name as text) as bytea
                    )
                ) for 16
            ) as bytea
        );
        RETURN NEW;
    END;
' LANGUAGE plpgsql;
```

```
CREATE TRIGGER tcsTYP_Name
BEFORE INSERT OR UPDATE OF TYP_Name ON _TYP_Name
FOR EACH ROW EXECUTE PROCEDURE tcsTYP_Name();
```

Pange tähele veergu *Metadata_TYP*. Selles olevad täisarvulised väärtused omavad viidet andmereale, mis esitab infot selle kohta, kuidas vaadeldav baastabeli rida andmebaasi „jõudis". Kuna viidatav rida võib olla teises andmebaasis ja isegi teises serveris, siis sellele veerule välisvõtit ei deklareerita.

3.3.4. Staatiline attribuut

Iga staatiline atribuut esitab olemi nimelist omadust, mille väärtuste ajalugu ei soovita andmebaasis säilitada. Järgnevalt esitatakse näide staatilise atribuudi realisatsioonist PostgreSQL andmebaasis:

```
CREATE TABLE IF NOT EXISTS _SO_TIT_Song_Title (
    SO_TIT_SO_ID int not null,
    SO_TIT_Song_Title varchar(255) not null,
    Metadata_SO_TIT int not null,
    constraint fkSO_TIT_Song_Title foreign key (
        SO_TIT_SO_ID
    ) references _SO_Song(SO_ID),
    constraint pkSO_TIT_Song_Title primary key (
        SO_TIT_SO_ID
    )
);
```

Kitsaskohad/erisused võrreldes realisatsiooniga MS SQL Serveris:

- PostgreSQLi eripärade tõttu tuleb atribuudile luua trigerite hierarhia, mis atribuudile vastavasse vaatesse lisatavad andmed eelsorteerib ning veendub, et alustabelisse ei sisestata (vastavalt atribuudi tüübile temporaalseid või mittetemporaalseid) duplikaate. Temporaalne duplikaat on väärtus, millel on viimati sisestatud (ehk kehtiva) väärtusega võrdne sisu. Erisus kehtib kõikidele atribuudi tüüpidele. Sarnane loogika on kasutusel ka MS SQL Serveri jaoks genereeritud koodis, aga MS SQL Serveri puhul piisab ühest trigerist. Mainitud hierarhia on lahti seletatud allpool.

Oletame, et tahame andmebaasi sisestada tabelis 5 toodud ridu. Tegemist on muusikapala autoritasu määraga, mis võib ajas muutuda. Tabel demonstreerib, et teine ja kolmas lisatav rida omavad võrdset autoritasu väärtust, mis nagu viitaks, et andmebaasi üritatakse sisestada korduvaid järjestikulisi väärtusi (ehk temporaalseid duplikaate). Tegelikkuses on teise rea muutmise aeg varasem kui esimesel real, ning tegemist ei ole temporaalse duplikaadiga, sest hind muutus ajas 1,33 € – 2,01 € – 1,33 €.

Tabel 5. Vaatesse lisatavad andmed

Muusikapala ID	**Muusikapala autoritasu**	**Muutmise aeg**
42	2,01 €	2013-01-01
42	1,33 €	2012-02-12
42	1,33 €	2014-05-09

Tabelis 6 on toodud samad read pärast andmete sorteerimist (muutmise aja järgi kasvavalt). On näha, et korduvaid järjestikulisi väärtusi ei sisestada ei tahetud. Selle väljaselgitamiseks ongi vaja trigereid.

Tabel 6. Alustabelisse lisatavad (juba sorteeritud) andmed

Muusikapala ID	**Muusikapala autoritasu**	**Muutmise aeg**
42	1,33 €	2012-02-12
42	2,01 €	2013-01-01
42	1,33 €	2014-05-09

Vajaminev trigerite hierarhia koosneb kolmest trigerist iga atribuudi kohta ning see on lahti seletatud allpool (muusikapala autoritasu ehk ühe ajaloolise atribuudi näitel). Kõik need trigerid on seotud atribuudile vastava vaatega (mitte atribuudile vastava tabeliga).

Enne lisama asumist, iga lisamistöö kohta (BEFORE INSERT, FOR EACH STATEMENT):

- luuakse atribuudi tabeliga sarnane ajutine tabel, kus hoitakse ka lisatava rea järjekorranumbrit ning infot, kas tegemist on korduva väärtusega või mitte.

```
CREATE OR REPLACE FUNCTION tcsSO_AUT_Song_Authorfee() RETURNS trigger AS '
BEGIN
  -- temporary table is used to create an insert order
  -- (so that rows are inserted in order with respect to temporality)
  CREATE TEMPORARY TABLE IF NOT EXISTS _tmp_SO_AUT_Song_Authorfee (
    SO_AUT_SO_ID int not null,
    Metadata_SO_AUT int not null,
    SO_AUT_ChangedAt timestamp not null,
    SO_AUT_Song_Authorfee numeric(5,2) not null,
    SO_AUT_Version bigint not null,
    SO_AUT_StatementType char(1) not null,
    primary key(
      SO_AUT_Version,
      SO_AUT_SO_ID
    )
  ) ON COMMIT DROP;

  RETURN NEW;
END;
' LANGUAGE plpgsql;
```

```
CREATE TRIGGER tcsSO_AUT_Song_Authorfee
BEFORE INSERT ON SO_AUT_Song_Authorfee
FOR EACH STATEMENT
EXECUTE PROCEDURE tcsSO_AUT_Song_Authorfee();
```

Lisamise asemel, iga rea kohta (INSTEAD OF INSERT, FOR EACH ROW):

- sisestatud read lisatakse ajutisse tabelisse koos eritunnusega.

```
CREATE OR REPLACE FUNCTION tcsiSO_AUT_Song_Authorfee() RETURNS trigger AS '
BEGIN
  -- insert rows into the temporary table
  INSERT INTO _tmp_SO_AUT_Song_Authorfee (
    SO_AUT_SO_ID,
    Metadata_SO_AUT,
    SO_AUT_ChangedAt,
    SO_AUT_Song_Authorfee,
    SO_AUT_Version,
    SO_AUT_StatementType,
  )
  SELECT
    NEW.SO_AUT_SO_ID,
    NEW.Metadata_SO_AUT,
    NEW.SO_AUT_ChangedAt,
    NEW.SO_AUT_Song_Authorfee,
    0,
    ''X'';

  RETURN NEW;
END;
' LANGUAGE plpgsql;
```

```
CREATE TRIGGER tcsiSO_AUT_Song_Authorfee
INSTEAD OF INSERT ON SO_AUT_Song_Authorfee
FOR EACH ROW
EXECUTE PROCEDURE tcsiSO_AUT_Song_Authorfee();
```

Peale ridade ajutisse tabelisse lisamist, iga lisamistöö kohta (AFTER INSERT, FOR EACH STATEMENT):

- leitakse iga rea järjekorranumber peale ridade sortimist muutmise aja järgi (ning peale korduvate väärtuste kõrvaleheitmist),
- leitakse, mitu erinevat unikaalset muutmise aega sisestatud ridade hulgas esineb,
- käiakse muutmise aja järgi kasvavalt üle ridade ning leitakse, kas tegemist on lisandunud väärtuse või (temporaalse) duplikaadiga. Selleks kasutatakse korduva sisestatud väärtuse leidmise funktsiooni (*restatement finder function*) *rfSO_AUT_Song_Authorfee*(), mis võrdleb sisestatud väärtust temale ajaliselt eelneva (või järgneva) sisestatud väärtusega,
- lisatakse rida atribuudi tabelisse juhul, kui tegemist pole (temporaalse) duplikaadiga,
- jäetakse rida lisamata, kui tegemist on (temporaalse) duplikaadiga,
- kustutatakse ajutine tabel.

```
CREATE OR REPLACE FUNCTION it_SO_AUT_Song_Authorfee() RETURNS trigger AS '
  DECLARE maxVersion int;
  DECLARE currentVersion int = 0;
BEGIN
  -- find ranks for inserted data (using self join)
  UPDATE _tmp_SO_AUT_Song_Authorfee
  SET SO_AUT_Version = v.rank
  FROM (
  SELECT
    DENSE_RANK() OVER (
      PARTITION BY
          SO_AUT_SO_ID
      ORDER BY
          SO_AUT_ChangedAt ASC
    ) AS rank,
    SO_AUT_SO_ID AS pk
  FROM _tmp_SO_AUT_Song_Authorfee
  ) AS v
  WHERE SO_AUT_SO_ID = v.pk
  AND SO_AUT_Version = 0;

  -- find max version
  SELECT
  MAX(SO_AUT_Version) INTO maxVersion
  FROM
  _tmp_SO_AUT_Song_Authorfee;

  -- loop over versions
  LOOP
    currentVersion := currentVersion + 1;

    -- set statement types
    UPDATE _tmp_SO_AUT_Song_Authorfee
```

```
    SET
    SO_AUT_StatementType =
      CASE
          WHEN AUT.SO_AUT_SO_ID is not null
          THEN ''D'' -- duplicate
          WHEN rfSO_AUT_Song_Authorfee(
            v.SO_AUT_SO_ID,
            v.SO_AUT_Song_Authorfee,
            v.SO_AUT_ChangedAt
          ) = 1
          THEN ''R'' -- restatement
          ELSE ''N'' -- new statement
      END
    FROM
      _tmp_SO_AUT_Song_Authorfee v
    LEFT JOIN
      _SO_AUT_Song_Authorfee AUT
    ON
      AUT.SO_AUT_SO_ID = v.SO_AUT_SO_ID
    AND
      AUT.SO_AUT_ChangedAt = v.SO_AUT_ChangedAt
    AND
      AUT.SO_AUT_Song_Authorfee = v.SO_AUT_Song_Authorfee
    WHERE
      v.SO_AUT_Version = currentVersion;

    -- insert data into attribute table
    INSERT INTO _SO_AUT_Song_Authorfee (
      SO_AUT_SO_ID,
      Metadata_SO_AUT,
      SO_AUT_ChangedAt,
      SO_AUT_Song_Authorfee
    )
    SELECT
      SO_AUT_SO_ID,
      Metadata_SO_AUT,
      SO_AUT_ChangedAt,
      SO_AUT_Song_Authorfee
    FROM
      _tmp_SO_AUT_Song_Authorfee
    WHERE
      SO_AUT_Version = currentVersion
    AND
      SO_AUT_StatementType in (''N'',''R'');

    EXIT WHEN currentVersion >= maxVersion;
  END LOOP;

  DROP TABLE IF EXISTS _tmp_SO_AUT_Song_Authorfee;

  RETURN NULL;
END;
' LANGUAGE plpgsql;
```

```
CREATE TRIGGER it_SO_AUT_Song_Authorfee
AFTER INSERT ON SO_AUT_Song_Authorfee
FOR EACH STATEMENT
EXECUTE PROCEDURE it_SO_AUT_Song_Authorfee();
```

Kuigi näites võeti aluseks ajalooline atribuut, kehtib antud erisus kõikidele atribuudi tüüpidele. Autor tahab esile tuua, et sarnane trigerite hierarhia tuleb luua ka ankrute ja sidemete puhul.

Tõenäoliselt saab antud funktsionaalsust PostgreSQLis ka kuidagi teisiti (lihtsamalt) teostada, kuid see eeldab analüüsi ning suure hulga testide läbiviimist.

3.3.5. Ajalooline attribuut

Iga ajaloolise atribuudi korral peab süsteem säilitama selle atribuudi väärtuste ajalugu. Järgnevalt esitatakse näide ajaloolise atribuudi realisatsioonist PostgreSQL andmebaasis:

```
CREATE TABLE IF NOT EXISTS _SO_AUT_Song_Authorfee (
    SO_AUT_SO_ID int not null,
    SO_AUT_Song_Authorfee numeric(5,2) not null,
    SO_AUT_ChangedAt timestamp not null,
    Metadata_SO_AUT int not null,
    constraint fkSO_AUT_Song_Authorfee foreign key (
        SO_AUT_SO_ID
    ) references _SO_Song(SO_ID),
    constraint pkSO_AUT_Song_Authorfee primary key (
        SO_AUT_SO_ID,
        SO_AUT_ChangedAt
    )
);
```

Tabelis on lisaks muudele veergudele ka veerg *SO_AUT_ChangedAt*. Antud veerus olevad väärtused näitavad, millal väärtus (antud juhul autoritasu) tabelisse lisati või millal väärtust muudeti (monotemporaalse generatsiooni puhul ei eristata lisamise ja muutmise aega). Teisisõnu, veerg kirjeldab, millal väärtus kehtima hakkas. Seda veergu kasutab ankurmodelleerimine ka sõlmitud ajaloolise atribuudi, ajaloolise sideme ja sõlmitud ajaloolise sideme puhul. Seda, kas see veerg peab olema tüüpi *timestamp* (väärtus esitab kuupäeva+kellaaega) või *date* (väärtus esitab kuupäeva), saab määrata ankurmudelit luues.

Ankurmodelleerimise vahendis saab ajaloolise atribuudi puhul määrata ka alljärgnevaid sätteid (autor tahab esile tuua, et antud valikud ei ole dokumenteeritud, ning autori arvates piisaks ühest valikust):

- *Restatable*:
    - Selgitus: määrab, kas atribuudi tabelis temporaalsed duplikaadid (ehk kaks identset järjestikulist väärtust muutuvas ajas) on lubatud või mitte (kas kitsendus luuakse või mitte).
    - Valikuvõimalused: sisse lülitatud (lubatud) või välja lülitatud (mittelubatud).

- Vaikimisi valik: sisse lülitatud.
- Märkus: antud valiku välja lülitamisel luuakse atribuudi tabelile CHECK-kitsendus, mis korduva sisestatud väärtuse leidmise funktsiooni (*restatement finder function*) abil temporaalsed duplikaadid keelab. Magistritöös kasutatud ankurmudel selliseid atribuute ei sisaldanud, aga kitsendus näeks välja selline (kui seda muusikapala autoritasu puhul kasutatud oleks):

```
ALTER TABLE SO_AUT_Song_Authorfee
  ADD CONSTRAINT rcSO_AUT_Song_Authorfee CHECK (
          rfSO_AUT_Song_Authorfee (
          SO_AUT_SO_ID,
          SO_AUT_Song_Authorfee,
          SO_AUT_ChangedAt
      ) = 0
  );
```

- *Idempotent*:
  - Selgitus: määrab, kas ajas järjestikuliste korduvate sisestatud väärtuste puhul korduvaid väärtusi ignoreeritakse või mitte.
  - Valikuvõimalused: sisse lülitatud (ignoreeritakse) või välja lülitatud (ei ignoreerita).
  - Vaikimisi valik: välja lülitatud.
  - Märkus: osutub kasulikuks süsteemis, mille korral andmebaasi sisestatakse tihti hulganisti korduvaid väärtuseid (näiteks valveandurite võrk). Ei sobi kasutamiseks, kui andmeid sisestatakse andmebaasi asünkroonselt. Duplikaatide leidmiseks kasutatakse korduva sisestatud väärtuse leidmise funktsiooni (*restatement finder function*).

3.3.6. Sõlmitud staatiline attribuut

Iga sõlmitud staatiline atribuut ühendab sõlme ja ankru ilma vajaduseta säilitada ajaloolisi väärtuseid. Järgnevalt esitatakse näide sõlmitud staatilise atribuudi realisatsioonist PostgreSQL andmebaasis:

```
CREATE TABLE IF NOT EXISTS _AR_ART_Artist_ArtistType (
    AR_ART_AR_ID int not null,
    AR_ART_TYP_ID smallint not null,
    Metadata_AR_ART int not null,
    constraint fk_A_AR_ART_Artist_ArtistType foreign key (
        AR_ART_AR_ID
    ) references _AR_Artist(AR_ID),
```

```
    constraint fk_K_AR_ART_Artist_ArtistType foreign key (
        AR_ART_TYP_ID
    ) references _TYP_Type(TYP_ID),
    constraint pkAR_ART_Artist_ArtistType primary key (
        AR_ART_AR_ID
    )
);
```

3.3.7. Sõlmitud ajalooline attribuut

Iga sõlmitud ajalooline atribuut ühendab sõlme ja ankru koos vajadusega säilitada ajaloolisi väärtuseid. Järgnevalt esitatakse näide sõlmitud ajaloolise atribuudi realisatsioonist PostgreSQL andmebaasis:

```
CREATE TABLE IF NOT EXISTS _AR_GEN_Artist_Gender (
    AR_GEN_AR_ID int not null,
    AR_GEN_GEN_ID smallint not null,
    AR_GEN_ChangedAt date not null,
    Metadata_AR_GEN int not null,
    constraint fk_A_AR_GEN_Artist_Gender foreign key (
        AR_GEN_AR_ID
    ) references _AR_Artist(AR_ID),
    constraint fk_K_AR_GEN_Artist_Gender foreign key (
        AR_GEN_GEN_ID
    ) references _GEN_Gender(GEN_ID),
    constraint pkAR_GEN_Artist_Gender primary key (
        AR_GEN_AR_ID,
        AR_GEN_ChangedAt
    )
);
```

3.3.8. Staatiline side

Iga staatiline side esitab seose kahe ankru vahel ilma vajaduseta säilitada ajaloolisi väärtuseid. Järgnevalt esitatakse näide staatilise sideme realisatsioonist PostgreSQL andmebaasis:

```
CREATE TABLE IF NOT EXISTS _SO_isContainedBy_AL_contains (
    SO_ID_isContainedBy int not null,
    AL_ID_contains int not null,
    Metadata_SO_isContainedBy_AL_contains int not null,
    constraint SO_isContainedBy_AL_contains_fkSO_isContainedBy foreign key
(
        SO_ID_isContainedBy
    ) references _SO_Song(SO_ID),
    constraint SO_isContainedBy_AL_contains_fkAL_contains foreign key (
        AL_ID_contains
    ) references _AL_Album(AL_ID),
    constraint pkSO_isContainedBy_AL_contains primary key (
        SO_ID_isContainedBy,
        AL_ID_contains
    )
);
```

3.3.9. Sõlmitud staatiline side

Iga sõlmitud staatiline side esitab seose vähemalt kahe ankru ja ühe sõlme vahel ilma vajaduseta säilitada ajaloolisi väärtuseid. Järgnevalt esitatakse näide sõlmitud staatilise sideme realisatsioonist PostgreSQL andmebaasis:

```
CREATE TABLE IF NOT EXISTS _SO_isDescribedBy_ST_describes_SRO_describes (
    SO_ID_isDescribedBy int not null,
    ST_ID_describes int not null,
    SRO_ID_describes smallint not null,
    Metadata_SO_isDescribedBy_ST_describes_SRO_describes int not null,
    constraint
SO_isDescribedBy_ST_describes_SRO_describes_fkSO_isDescribedBy foreign key
(
        SO_ID_isDescribedBy
    ) references _SO_Song(SO_ID),
    constraint SO_isDescribedBy_ST_describes_SRO_describes_fkST_describes
foreign key (
        ST_ID_describes
    ) references _ST_Style(ST_ID),
    constraint SO_isDescribedBy_ST_describes_SRO_describes_fkSRO_describes
foreign key (
        SRO_ID_describes
    ) references _SRO_StyleRole(SRO_ID),
    constraint pkSO_isDescribedBy_ST_describes_SRO_describes primary key (
        SO_ID_isDescribedBy,
        ST_ID_describes
    )
);
```

3.3.10. Ajalooline side

Iga ajalooline side esitab seose kahe ankru vahel koos vajadusega säilitada ajaloolisi väärtuseid. Järgnevalt esitatakse näide staatilise sideme realisatsioonist PostgreSQL andmebaasis:

```
CREATE TABLE IF NOT EXISTS _SO_isPerformedBy_AR_performs (
    SO_ID_isPerformedBy int not null,
    AR_ID_performs int not null,
    SO_isPerformedBy_AR_performs_ChangedAt date not null,
    Metadata_SO_isPerformedBy_AR_performs int not null,
    constraint SO_isPerformedBy_AR_performs_fkSO_isPerformedBy foreign key(
        SO_ID_isPerformedBy
    ) references _SO_Song(SO_ID),
    constraint SO_isPerformedBy_AR_performs_fkAR_performs foreign key (
        AR_ID_performs
    ) references _AR_Artist(AR_ID),
    constraint pkSO_isPerformedBy_AR_performs primary key (
        SO_ID_isPerformedBy,
        AR_ID_performs,
        SO_isPerformedBy_AR_performs_ChangedAt
    )
);
```

3.3.11. Sõlmitud ajalooline side

Iga sõlmitud ajalooline side esitab seose vähemalt kahe ankru ja ühe sõlme vahel koos vajadusega säilitada ajaloolisi väärtuseid. Järgnevalt esitatakse näide sõlmitud ajaloolise sideme realisatsioonist PostgreSQL andmebaasis:

```
CREATE TABLE IF NOT EXISTS _ST_describes_AR_isDescribedBy_SRO_describes (
    ST_ID_describes int not null,
    AR_ID_isDescribedBy int not null,
    SRO_ID_describes smallint not null,
    ST_describes_AR_isDescribedBy_SRO_describes_ChangedAt timestamp not
null,
    Metadata_ST_describes_AR_isDescribedBy_SRO_describes int not null,
    constraint ST_describes_AR_isDescribedBy_SRO_describes_fkST_describes
foreign key (
        ST_ID_describes
    ) references _ST_Style(ST_ID),
    constraint
ST_describes_AR_isDescribedBy_SRO_describes_fkAR_isDescribedBy foreign key
(
        AR_ID_isDescribedBy
    ) references _AR_Artist(AR_ID),
    constraint ST_describes_AR_isDescribedBy_SRO_describes_fkSRO_describes
foreign key (
        SRO_ID_describes
    ) references _SRO_StyleRole(SRO_ID),
    constraint pkST_describes_AR_isDescribedBy_SRO_describes primary key (
        ST_ID_describes,
        AR_ID_isDescribedBy,
        ST_describes_AR_isDescribedBy_SRO_describes_ChangedAt
    )
);
```

3.4. Perspektiivid

Kuuenda normaalkuju kasutamise tulemusena tekib andmebaasi väga palju tabeleid. Mõistlik on kasutada abstraktsioonikihti, mis andmebaasi kasutajate jaoks virtuaalselt denormaliseeriks ning ajalooliste andmete käitlemist lihtsustaks. Selleks kasutab ankurmodelleerimine perspektiive. Perspektiivid luuakse ankrutele ja sidemetele, ning need baseeruvad hüpoteetilisel täielikul vaatel (*complete view*), mida tegelikult ei eksisteeri. Hüpoteetiline täielik vaade sisaldab ka kõiki ajaloolisi andmeid (ajapunkti ei ole määratud).

Autor tahab esile tuua, et ankurmodelleerimise teooria ühe rajaja, Lars Rönnbäcki hinnangul saab perspektiivide abil lihtsustada kuni 95% ankurmodelleeritud andmebaasis tehtavatest SQL-päringutest.

Ankru perspektiivid baseeruvad niisiis täielikul vaatel, mis ankru tabeli virtuaalselt denormaliseerib, ühendades ankru kõigi tema (sõlmitud ja mittesõlmitud) atribuutidega,

kasutades LEFT OUTER JOIN (vasakpoolse välisühendamise) operaatorit. Iga ankru kohta loob generaator neli perspektiivi, võttes aluseks kõige tüüpilisemad andmebaasipäringud.

- Kõige viimane perspektiiv (*latest view* ehk *latest perspective*). Erinevus täielikust vaatest seisneb selles, et ajalooliste atribuutide puhul leitakse alampäringuga iga atribuudi väärtuste kõige viimane ehk kehtiv versioon. Kui väärtuse kõige viimane versioon on tulevikus (näiteks prognoositud inflatsioon viie aasta pärast), siis tagastab alampäring kõige kaugemal tulevikus oleva väärtuse. Antud perspektiivil baseerub mitmeid trigereid, sh ka enamus andmete sisestamisega seotud trigereid.
- Ajapunkti perspektiiv (*point-in-time function* ehk *point-in-time perspective*). Sarnaneb „kõige viimase perspektiiviga", aga atribuutide väärtused esitatakse mingi kindla hetke seisuga. Ühendamisoperatsioonid atribuutidele vastavate tabelitega teostatakse läbi atribuudi tagasikerimise funktsiooni (*attribute rewinder function*), millega saab leida atribuudi tabeli sisu etteantud hetkel.
- Intervalli perspektiiv (*interval function* ehk *difference perspective*), mis sarnaneb „ajapunkti perspektiiviga". Erinevus seisneb ainult selles, et tagastatakse väärtused, mis jäävad kahe määratud ajahetke vahele.
- Nüüd-perspektiiv (*now view* ehk *now perspective*). Baseerub „ajapunkti perspektiivil", kuid ajahetkeks on vaate avamise hetk, mis tähendab, et tagastatakse kehtivad andmed. Nüüd-perspektiiv tagastab samad andmed nagu „kõige viimane perspektiiv", välja arvatud juhul, kui andmebaasis on andmeid „tuleviku" kohta (näiteks prognoositud andmed). Tulevikus olevaid andmeid ei tagastata.

Nii ajapunkti funktsioon kui intervalli funktsioon on tabelifunktsioonid. See tähendab, et need tagastavad ridade hulga. Kuna nendel funktsioonidel on ka parameetrid, siis võib neist mõelda kui parametriseeritud vaadetest, mille väärtus sõltub pöördumisel kasutatud argumentidest. Tabelifunktsioone on võimalik PostgreSQLis luua. Tunnete need ära sellest, et nende päisesse on kirjutatud RETURNS TABLE.

	so_id integer	metadata_so integer	so_tit_so_id integer	metadata_so integer	so_tit_song_ character vai	so_aut_so_id integer	metadata_so integer	so_aut_chan timestamp w	so_aut_song numeric	so_len_so_id integer	metadata_so integer	so_len_song smallint
1	1	7	1	42	rwPvUwsZucL	1	35	2014-12-13	1.10	1	20	1383
2	2	3	2	8	L VQINxNmCy	2	17	2014-11-03	1.10	2	32	762
3	3	46	3	27	UMvkCZFfoYN	3	23	2014-10-21	3.20	3	49	1623
4	4	26	4	33	mWnuIuGTUsJ	4	1	2014-11-15	2.20	4	17	2377
5	5	6	5	25	Su rPlxlGdR	5	24	2014-12-22	2.20	5	40	1772
6	6	20	6	32	CeT Xz lLvS	6	41	2014-12-08	3.10	6	23	1594
7	7	26	7	22	OyMKhqL IcM	7	24	2014-11-24	3.10	7	4	2167
8	8	19	8	49	QUGYDEt UV	8	24	2014-11-19	3.10	8	47	1452
9	9	37	9	6	QK	9	12	2014-10-29	2.20	9	26	1346
10	10	10	10	46	ZaqokgDn	10	37	2014-12-18	1.10	10	14	3186
11	11	49	11	26	VjPShVHPB B	11	24	2014-10-26	2.10	11	18	3246
12	12	13	12	4	eqZd XCJH M	12	19	2014-10-28	2.20	12	21	2635
13	13	42	13	19	mzKJZHRPxiB	13	48	2014-12-01	3.20	13	15	2804
14	14	22	14	43	vxkcUSah DQ	14	7	2014-12-02	1.20	14	42	1037
15	15	47	15	27	ahruBpffX E	15	22	2014-12-21	3.10	15	28	358
16	16	43	16	24	DdtfPQTYcNn	16	39	2014-11-24	1.20	16	25	1429
17	17	39	17	50	iFVjheASowi	17	32	2014-10-31	1.10	17	44	1646
18	18	29	18	38	wzRYLlzthmg	18	1	2014-10-22	3.10	18	44	1448
19	19	22	19	12	oqYqhJxTiUY	19	18	2014-12-15	3.10	19	23	3406
20	20	21	20	41	IThqOlg	20	31	2014-11-26	1.20	20	48	531
21	21	20	21	8	dQf kAPgWZa	21	15	2014-12-26	2.20	21	1	573
22	22	30	22	16	MKqsPFwwVnz	22	18	2014-10-30	3.10	22	9	3517
23	23	6	23	12	inGSiQls kp	23	12	2014-12-21	1.20	23	39	687
24	24	32	24	41	hPumIMWbJrS	24	48	2014-11-02	3.10	24	27	2643
25	25	1	25	23	MlMrnJCmrwP	25	47	2014-12-19	1.10	25	22	2285
26	26	48	26	16	Y puUsNsrFE	26	8	2014-12-25	3.20	26	24	2747
27	27	3	27	49	BIIjsBdY lX	27	6	2014-10-25	2.10	27	13	2103
28	28	38	28	9	myoCfzJWy	28	37	2014-11-27	1.20	28	49	3265
29	29	27	29	2	xTeprhJFexa	29	27	2014-12-23	3.10	29	47	2280
30	30	32	30	15	fmPvHOGfkCq	30	22	2014-11-11	1.10	30	25	1835
31	31	42	31	11	seUnqMsamaG	31	47	2014-11-23	3.10	31	35	3153
32	32	24	32	2	YcNvfoMgTUn	32	50	2014-11-26	1.20	32	1	1496
33	33	12	33	29	AmaKRxLTSaY	33	5	2014-10-26	2.20	33	12	3260

Joonis 14. Muusikapala ankru „nüüd-perspektiiv”

Joonisel 14 on toodud muusikapala ankru „nüüd-perspektiivi” poolt tagastatavad andmed (genereeritud testandmete põhjal).

Analoogsed perspektiivid luuakse ka sidemete tabelitele. Erinevus seisneb vaid selles, et ankrute ja atribuutide asemel päritakse andmeid sidemete ja sõlmede tabelitest.

Generaatoriga saab soovi korral luua ka „ärivaated” (*business views*), mis on perspektiivide lihtsustatud versioonid (vähem veerge). Selleks tuleb enne koodi genereerimist rakenduse menüüs määrata valituks „*Use business views*”. Käesolevas töös „ärivaateid” ei tõlgitud, kuid autor usub, et need lisanduvad peagi.

Järgnevalt esitatakse muusikapala ankru kõik perspektiivid (ilma veergude loeteluta).

3.4.1. Kõige viimane perspektiiv (*latest perspective*)

```
CREATE OR REPLACE VIEW lso_song AS
 SELECT
    ... palju veerge ...
   FROM so_song so
   LEFT JOIN so_tit_song_title tit
     ON tit.so_tit_so_id = so.so_id
   LEFT JOIN so_aut_song_authorfee aut
     ON aut.so_aut_so_id = so.so_id
   AND aut.so_aut_changedat = ((
```

```
     SELECT MAX(sub.so_aut_changedat) AS max
     FROM so_aut_song_authorfee sub
    WHERE sub.so_aut_so_id = so.so_id
   ))
   LEFT JOIN so_len_song_length len
   ON len.so_len_so_id = so.so_id;
```

Vaade kasutab SELECT MAX(...) alampäringut atribuudi kõige viimase väärtuse leidmiseks (väärtus võib olla ka tulevikus). Sellist alampäringut tuleb kasutada ankru kõigi ajalooliste atribuutide korral. Nagu käesolevast vaatest näha, siis pealkirja (*title*) ja pikkuse (*length*) korral ei olnud seda vaja teha, sest need on staatilised atribuudid.

3.4.2. Ajapunkti perspektiiv (point-in-time perspective)

```
CREATE OR REPLACE FUNCTION pso_song(
  IN changingtimepoint timestamp without time zone
) RETURNS TABLE(... palju veerge ...) AS
$BODY$
  SELECT
    ... palju veerge ...
  FROM SO_Song SO
  LEFT JOIN SO_TIT_Song_Title TIT
    ON TIT.SO_TIT_SO_ID = SO.SO_ID
  LEFT JOIN
    rSO_AUT_Song_Authorfee(CAST(changingTimepoint AS timestamp)) AUT
    ON AUT.SO_AUT_SO_ID = SO.SO_ID
  AND AUT.SO_AUT_ChangedAt = (
        SELECT
            max(sub.SO_AUT_ChangedAt)
        FROM
            rSO_AUT_Song_Authorfee(
              CAST(changingTimepoint AS timestamp)
            ) sub
        WHERE
            sub.SO_AUT_SO_ID = SO.SO_ID
  )
  LEFT JOIN SO_LEN_Song_Length LEN
    ON LEN.SO_LEN_SO_ID = SO.SO_ID;
$BODY$
  LANGUAGE sql;
```

Ühendamisoperatsioonid atribuutidele vastavate tabelitega teostatakse läbi atribuudi tagasikerimise funktsiooni (*rSO_AUT_Song_Authorfee*). See funktsioon tagastab ajaloolise atribuudi tabelist read, milles hoitava väärtuse muutmisaeg on väiksem või võrdne funktsioonile etteantud ajaga. Funktsiooni kood on toodud allpool. Selline funktsioon tuleb genereerida iga ajaloolise atribuudi kohta.

```
CREATE OR REPLACE FUNCTION rso_aut_song_authorfee(
  IN changingtimepoint timestamp without time zone
) RETURNS TABLE(
```

```
  metadata_so_aut integer,
  so_aut_so_id integer,
  so_aut_song_authorfee numeric,
  so_aut_changedat timestamp without time zone
) AS
$BODY$
  SELECT
    Metadata_SO_AUT,
    SO_AUT_SO_ID,
    SO_AUT_Song_Authorfee,
    SO_AUT_ChangedAt
  FROM
    SO_AUT_Song_Authorfee
  WHERE
    SO_AUT_ChangedAt <= changingTimepoint;
$BODY$
  LANGUAGE sql;
```

3.4.3. Intervalli perspektiiv (*difference perspective*)

```
CREATE OR REPLACE FUNCTION dso_song(
  IN intervalstart timestamp without time zone,
  IN intervalend timestamp without time zone,
  IN selection text DEFAULT NULL::text
) RETURNS TABLE(... palju veerge ...) AS
$BODY$
  SELECT
    timepoints.inspectedTimepoint,
    timepoints.mnemonic,
    pSO.*
  FROM (
    SELECT DISTINCT
        SO_AUT_SO_ID AS SO_ID,
        CAST(
          SO_AUT_ChangedAt AS timestamp without time zone
        ) AS inspectedTimepoint,
        'AUT' AS mnemonic
    FROM
        SO_AUT_Song_Authorfee
    WHERE
        (selection is null OR selection like '%AUT%')
    AND
        SO_AUT_ChangedAt BETWEEN intervalStart AND intervalEnd
  ) timepoints
  CROSS JOIN LATERAL
    pSO_Song(timepoints.inspectedTimepoint) pSO
  WHERE
    pSO.SO_ID = timepoints.SO_ID;
$BODY$
  LANGUAGE sql;
```

Leides andmebaasist vaadeldava ajaloolise atribuudi kohta kõik etteantud vahemikku jäänud ajahetked, leitakse neile hetkedele vastavad andmed, kasutades korduvalt „ajapunkti perspektiivi”.

3.4.4. Nüüd-perspektiiv (now perspective)

```
CREATE OR REPLACE VIEW nso_song AS
  SELECT
    *
  FROM pso_song(LOCALTIMESTAMP);
```

Baseerub „ajapunkti perspektiivil", andes sellele ette hetkeaja. Kuna „ajapunkti perspektiiv" tagastab andmed, mille muutmisaeg on väiksem või võrdne etteantud ajaga, siis tulevikus olevaid andmeid ei tagastata.

4. Realiseerimise tulem

Peale vajalike teisendusreeglite läbiviimist, millest enamuse moodustas koodi kirjutamine programmeerimiskeeles Javascript, valmis generaator, mis suudab koostada andmebaasiobjektide loomiseks vajaliku koodi PostgreSQLi jaoks.

4.1. Mallid

MS SQL Serveri koodi genereerimiseks on 19 malli, mille tulemusena valmib näitemudeli korral üle 3000 rea koodi. Osad neist mallidest olid väga pikad ning raskesti loetavad, mistõttu „tükeldas" autor need väiksemateks osadeks. PostgreSQLi jaoks tekkis kokku 39 malli, kuid kõiki ei tõlgitud, sest käesolev töö ei kasutanud ekvivalentsi (*equivalence*) ja ärivaateid (*business views*). Samuti ei teisendatud faile, mis loodud mudeli katsetamise seisukohast tundusid ebavajalikud.

Järgnevalt on toodud mittetäielik nimekiri mallidest, mida oli vaja tõlkida või muuta, saavutamaks PostgreSQL jaoks genereeritud koodi puhul MS SQL Serveriga sarnast tulemust ning võimaldamaks koodi käivitada PostgreSQL andmebaasisüsteemis.

4.1.1. CreateSchemaTracking.js

Tegemist on malliga, mis loob tabelid, vaated ja funktsioonid, jälgimaks andmebaasiskeemi muutumist ajas. Skeemi spetsifikatsiooni hoitakse andmebaasis XML-formaadis ning see on tuletatav kasutatud ankurmudelist.

Loodavad vaated võimaldavad sirvida ankrute, sõlmede, sidemete ja atribuutide nimistut. Samuti genereeritakse funktsioonid, mis genereerivad andmebaasielementide kopeerimise ja kustutamise skriptid.

Käesoleva töö raames antud faili PostgreSQL formaati ei viidud, kuid autor tegi mõned katsetused ning tuvastas, et see on võimalik.

All on toodud koodinäide CreateSchemaTracking.js failist (juba PostgreSQL formaadis), mis leiab skeemi parameetrid, võttes aluseks tabelis oleva XMLi:

```
...
-- Schema expanded view ----------------------------------------------
-- A view of the schema table that expands the XML attributes into columns
CREATE OR REPLACE VIEW _Schema_Expanded AS
SELECT
  version,
  activation,
  schema,
```

```
(xpath('/schema[1]/@format[1]', schema))[1]::text as format,
(xpath('/schema[1]/@date[1]', schema))[1]::text as date,
(xpath('/schema[1]/@time[1]', schema))[1]::text as time,
(xpath('/schema[1]/metadata[1]/@temporalization[1]', schema))[1]::text as
temporalization,
...veel väga palju ridu...
FROM
      _Schema;
...
```

4.1.2. NamingConvention.js

Antud mallis kirjeldatakse PostgreSQLile omaseid väärtuseid (skeemi vaikimisi nimi, hetkeaeg jms). Mallist võib mõelda kui konfiguratsioonifailist.

Fragment NamingConvention.js failist:

```
...
schema.metadata.encapsulation = 'public';
schema.metadata.chronon = 'timestamp without time zone';
schema.metadata.now = 'LOCALTIMESTAMP';
schema.metadata.identitySuffix = 'id';
schema.metadata.checksumSuffix = 'ck';
...
```

4.1.3. DatabaseInitialization.js

Antud malli saab kasutada skeemi loomiseks (ning tulevikus ka muude sarnaste või seotud toimingute jaoks). Mall kasutab väärtusi ka failist NamingConvention.js. Selle faili lisas autor (MS SQL Serveri puhul sellist malli ei eksisteeri).

Fragment DatabaseInitialization.js failist (skeemi loomine):

```
...
CREATE SCHEMA IF NOT EXISTS $schema.metadata.encapsulation;
SET search_path=$schema.metadata.encapsulation;
...
```

PostgreSQL andmebaasis luuakse skeem *public* andmebaasi loomisel automaatselt. Koostöös malliga NamingConvention.js saab määrata, et andmebaasis tuleb ankurmudeli realiseerimiseks mõeldud andmebaasiobjektide jaoks luua mõni uus skeem.

4.1.4. CreateKnots.js, CreateKnotTriggers.js

Antud mallid loovad sõlmede tabelid (ja identsed vaated). Lisaks luuakse trigerid, mis arvutavad räsi väärtusi.

Fragment CreateKnots.js failist (sõlme loomine):

```
CREATE TABLE IF NOT EXISTS _$knot.name (
    $knot.identityColumnName $(knot.isGenerator())?
      $knot.identityGenerator not null, : $knot.identity not null,
    $knot.valueColumnName $knot.dataRange not null,
    $(knot.hasChecksum())? $knot.checksumColumnName bytea not null,
    $(schema.METADATA)?
      $knot.metadataColumnName $schema.metadata.metadataType not null,
    constraint pk$knot.name primary key (
        $knot.identityColumnName
    ),
    constraint uq$knot.name unique (
        $(knot.hasChecksum())?
          $knot.checksumColumnName : $knot.valueColumnName
    )
);
```

4.1.5. CreateAnchorsAndAttributes.js, CreateAnchorTriggers.js, CreateAttributeTriggers.js

Antud mallid loovad tabelid (ning identsed vaated) ankrutele, ajaloolistele atribuutidele, sõlmitud ajaloolistele atribuutidele, sõlmitud staatilistele atribuutidele ja staatilistele atribuutidele. Lisaks luuakse PostgreSQLi spetsiifiline trigerite hierarhia, mis sisestatud ridu sorteerib ja tabelitesse temporaalsete duplikaatide lisamise välistab, ning mille kohta anti ülevaade teisendusreeglite peatükis. Sarnane hierarhia tuleb luua nii ankrute, sidemete kui ka atribuutide tabelitele ehitatud vaadetele. Käesoleva töö raames programmeeriti antud loogika valmis atribuutide jaoks. Selle teostamiseks tuli korduvalt kaasata nii magistritöö juhendaja Erki Eessaar kui ka ankurmodelleerimise teooria rajaja Lars Rönnbäck, arutamaks mitmeid kitsaskohti ning võimalikke lahenduskäike.

Fragment CreateAnchorsAndAttributes.js failist (loob ankru tabeli):

```
CREATE TABLE IF NOT EXISTS _$anchor.name (
    $anchor.identityColumnName $(anchor.isGenerator())?
      $anchor.identityGenerator not null, : $anchor.identity not null,
    $(schema.METADATA)?
      $anchor.metadataColumnName $schema.metadata.metadataType not null, :
      $anchor.dummyColumnName boolean null,
    constraint pk$anchor.name primary key (
        $anchor.identityColumnName
    )
);
```

4.1.6. CreateTies.js, CreateTieTriggers.js

Antud mallid loovad tabelid (ning identsed vaated) staatilisele sidemele, ajaloolisele sidemele, sõlmitud staatilisele sidemele ning sõlmitud ajaloolisele sidemele. Samuti luuakse eelmises

punktis mainitud trigerite hierarhia, mis sarnaneb ankrute ja atribuutide jaoks looduga (käesoleva töö raames sidemete trigerite hierarhiat ei programmeeritud).

Fragment CreateTies.js failist (sideme loomine; antud näide esitab ilmekalt, miks on malle raske lugeda, kui neis väga palju IF-tingimusi kasutada):

```
CREATE TABLE IF NOT EXISTS _$tie.name (
~*/
    var role;
    while (role = tie.nextRole()) {
/*~
    $role.columnName $(role.anchor)? $role.anchor.identity not null, :
$role.knot.identity not null,
~*/
    }
/*~
    $(tie.timeRange)? $tie.changingColumnName $tie.timeRange not null,
    $(schema.METADATA)?
      $tie.metadataColumnName $schema.metadata.metadataType not null,
~*/
    while (role = tie.nextRole()) {
        var knotReference = '';
        if(role.knot) {
            knotReference += '' + (role.knot.isEquivalent() ?
role.knot.identityName : role.knot.name) + '';
        }
/*~
    constraint ${(tie.name + '_fk' + role.name)}$ foreign key (
        $role.columnName
    ) references $(role.anchor)?
_$role.anchor.name($role.anchor.identityColumnName), :
_$knotReference($role.knot.identityColumnName),
~*/
    }
    // one-to-one and we need additional constraints
    if(!tie.hasMoreIdentifiers()) {
        while (role = tie.nextRole()) {
            if(role.isAnchorRole()) {
                if(tie.isHistorized()) {
/*~
    constraint ${tie.name + '_uq' + role.name}$ unique (
        $role.columnName,
        $tie.changingColumnName
    ),
~*/
                }
                else {
/*~
    constraint ${tie.name + '_uq' + role.name}$ unique (
        $role.columnName
    ),
~*/
                }
            }
        }
    }
/*~
    constraint pk$tie.name primary key (
```

```
~*/
    if(tie.hasMoreIdentifiers()) {
        while (role = tie.nextIdentifier()) {
/*~
        $role.columnName~*/
            if(tie.hasMoreIdentifiers() || tie.isHistorized()) {
                /*~,~*/
            }
        }
    }
    else {
        while (role = tie.nextRole()) {
/*~
        $role.columnName~*/
            if(tie.hasMoreRoles() || tie.isHistorized()) {
                /*~,~*/
            }
        }
    }
    if(tie.isHistorized()) {
/*~
        $tie.changingColumnName
~*/
    }
/*~
    )
);
```

4.1.7. AddAttributeRestatementConstraints.js, AddTieRestatementConstraints.js

Antud mallid loovad iga ajaloolise atribuudi ja sideme jaoks „korduva sisestatud väärtuse leidmise funktsiooni" (*restatement finder function*), mille abil saab otsida, kas tabelisse lisatava väärtusega „külgneb" mõnda identset väärtust muutuvas ajas. Teisisõnu, funktsioon leiab, kas rea lisamisel tekiks kronoloogilises järjestuses kahte identset väärtust. Funktsiooni poolt tagastatud väärtus on *boolean*-tüüpi ehk funktsioon tagastab *TRUE* (kui tekiks kaks identset ajas „külgnevat" väärtust) või *FALSE* (ei teki identseid ajas „külgnevaid" väärtuseid).

Fragment AddAttributeRestatementConstraints.js failist (funktsiooni loomine):

```
CREATE OR REPLACE FUNCTION rf$attribute.name(
    id $anchor.identity,
    $(attribute.isEquivalent())? eq $schema.metadata.equivalentRange,
    value $valueType,
    changed $attribute.timeRange
) RETURNS smallint AS '
    BEGIN
        IF EXISTS (
            SELECT
                value
            WHERE
                value = (
                    SELECT
                        pre.$valueColumn
                    FROM
```

```
                        $(attribute.isEquivalent())? e$attribute.name(eq)
pre : $attribute.name pre
                    WHERE
                        pre.$attribute.anchorReferenceName = id
                    AND
                        pre.$attribute.changingColumnName < changed
                    ORDER BY
                        pre.$attribute.changingColumnName DESC
                    LIMIT 1
            )
        )
        OR EXISTS(
            SELECT
                value
            WHERE
                value = (
                    SELECT
                        fol.$valueColumn
                    FROM
                        $(attribute.isEquivalent())? e$attribute.name(eq)
fol : $attribute.name fol
                    WHERE
                        fol.$attribute.anchorReferenceName = id
                    AND
                        fol.$attribute.changingColumnName > changed
                    ORDER BY
                        fol.$attribute.changingColumnName ASC
                    LIMIT 1
            )
        )
        THEN
            RETURN 1;
        END IF;

        RETURN 0;
    END;
' LANGUAGE plpgsql;
```

4.1.8. CreateAttributeRewinders.js

Antud mall genereerib ajaloolisele (ja sõlmitud ajaloolisele) atribuudile „atribuudi tagasikerimise funktsiooni" (*attribute rewinder function*). Funktsioon tagastab ajaloolise atribuudi tabelist read, milles hoitava väärtuse muutmisaeg on väiksem või võrdne funktsioonile etteantud ajaga. Funktsiooni kasutavad mitmed trigerid ja vaated.

Fragment CreateAttributeRewinders.js failist (funktsiooni loomine):

```
CREATE OR REPLACE FUNCTION r$attribute.name (
    $(attribute.isEquivalent())?
      equivalent $schema.metadata.equivalentRange,
    changingTimepoint $attribute.timeRange
) RETURNS TABLE(
    $(schema.METADATA)?
      $attribute.metadataColumnName $schema.metadata.metadataType,
    $attribute.anchorReferenceName $anchor.identity,
    $(attribute.isEquivalent())?
```

```
     $attribute.equivalentColumnName $schema.metadata.equivalentRange,
    $(!attribute.isKnotted() && attribute.hasChecksum())?
      $attribute.checksumColumnName bytea,
    $attribute.valueColumnName $attributeValueColumnType,
    $attribute.changingColumnName $attribute.timeRange
) AS '
    SELECT
        $(schema.METADATA)? $attribute.metadataColumnName,
        $attribute.anchorReferenceName,
        $(attribute.isEquivalent())? $attribute.equivalentColumnName,
        $(!attribute.isKnotted() && attribute.hasChecksum())?
          $attribute.checksumColumnName,
        $attribute.valueColumnName,
        $attribute.changingColumnName
    FROM
        $(attribute.isEquivalent())?
          e$attribute.name(equivalent) : $attribute.name
    WHERE
        $attribute.changingColumnName <= changingTimepoint;
' LANGUAGE SQL;
```

4.1.9. DropAnchorPerspectives.js, CreateAnchorPerspectiveLatest.js, CreateAnchorPerspectivePointInTime.js, CreateAnchorPerspectiveNow.js, CreateAnchorPerspectiveDifference.js

Antud mallid kustutavad ning loovad ankru perspektiive. Kui MS SQL Serveri korral oli perspektiivide kustutamise ja loomise kood ühes mallifailis, siis PostgreSQLi puhul tõstis autor koodi loogilised osad eraldi failidesse, sest üks mall oli raskesti loetav.

Perspektiividest oli juttu teisendusreeglite peatükis, ja seetõttu neid siin uuesti lahti ei seletata. Küll aga tahaks autor esile tuua, et eelpool korduvalt mainitud trigerite hierarhia järel võttis perspektiivide „tõlkimine" kõige rohkem aega ning kaasata tuli ka ankurmodelleerimise teooria rajaja Lars Rönnbäck, selgitamaks paljusid tehnilisi nüansse.

Fragment CreateAnchorPerspectivePointInTime.js failist („ajapunkti perspektiivi" loomine; autori hinnangul on tegemist äärmiselt raskesti tõlgitava koodiga):

```
CREATE OR REPLACE FUNCTION p$anchor.name (
    changingTimepoint $schema.metadata.chronon
)
RETURNS TABLE (
    $anchor.identityColumnName $anchor.identity,
    $(schema.METADATA)?
      $anchor.metadataColumnName $schema.metadata.metadataType,~*/
      while (attribute = anchor.nextAttribute()) {
/*~
    $(schema.IMPROVED)?
      $attribute.anchorReferenceName $anchor.identity, $(schema.METADATA)?
      $attribute.metadataColumnName $schema.metadata.metadataType,
    $(attribute.timeRange)?
      $attribute.changingColumnName $attribute.timeRange,
    $(attribute.isEquivalent())?
```

```
     $attribute.equivalentColumnName $schema.metadata.equivalentRange,
~*/
            if(attribute.isKnotted()) {
                  knot = attribute.knot;
/*~
    $(knot.hasChecksum())? $attribute.knotChecksumColumnName bytea,
    $(knot.isEquivalent())?
      $attribute.knotEquivalentColumnName
      $schema.metadata.equivalentRange,
    $attribute.knotValueColumnName $knot.dataRange,
    $(schema.METADATA)?
      $attribute.knotMetadataColumnName $schema.metadata.metadataType,
~*/
            }
/*~
    $(attribute.hasChecksum())? $attribute.checksumColumnName bytea,
    $attribute.valueColumnName $(attribute.isKnotted())? $knot.identity:
$attribute.dataRange~*/
/*~$(anchor.hasMoreAttributes())?,
~*/
      }
/*~
) AS '
SELECT
    $anchor.mnemonic\.$anchor.identityColumnName,
    $(schema.METADATA)? $anchor.mnemonic\.$anchor.metadataColumnName,
~*/
        while (attribute = anchor.nextAttribute()) {
/*~
    $(schema.IMPROVED)?
$attribute.mnemonic\.$attribute.anchorReferenceName,
    $(schema.METADATA)? $attribute.mnemonic\.$attribute.metadataColumnName,
    $(attribute.timeRange)?
$attribute.mnemonic\.$attribute.changingColumnName,
    $(attribute.isEquivalent())?
$attribute.mnemonic\.$attribute.equivalentColumnName,
~*/
            if(attribute.isKnotted()) {
                knot = attribute.knot;
/*~
    $(knot.hasChecksum())? k$attribute.mnemonic\.$knot.checksumColumnName
AS
  $attribute.knotChecksumColumnName,
    $(knot.isEquivalent())?
k$attribute.mnemonic\.$knot.equivalentColumnName AS
  $attribute.knotEquivalentColumnName,
    k$attribute.mnemonic\.$knot.valueColumnName AS
  $attribute.knotValueColumnName,
    $(schema.METADATA)? k$attribute.mnemonic\.$knot.metadataColumnName AS
$attribute.knotMetadataColumnName,
~*/
            }
/*~
    $(attribute.hasChecksum())?
$attribute.mnemonic\.$attribute.checksumColumnName,

$attribute.mnemonic\.$attribute.valueColumnName$(anchor.hasMoreAttributes()
)?,
~*/
        }
/*~
```

```
FROM
    $anchor.name $anchor.mnemonic
~*/
        while (attribute = anchor.nextAttribute()) {
            if(attribute.isHistorized()) {
                if(attribute.isEquivalent()) {
/*~
LEFT JOIN
    r$attribute.name(0, CAST(changingTimepoint AS
  $attribute.timeRange)) $attribute.mnemonic
~*/
                }
                else {
/*~
LEFT JOIN
    r$attribute.name(CAST(changingTimepoint AS $attribute.timeRange))
$attribute.mnemonic
~*/
                }
/*~
ON
    $attribute.mnemonic\.$attribute.anchorReferenceName =
  $anchor.mnemonic\.$anchor.identityColumnName
AND
    $attribute.mnemonic\.$attribute.changingColumnName = (
        SELECT
            max(sub.$attribute.changingColumnName)
        FROM
            $(attribute.isEquivalent())? r$attribute.name(0,
CAST(changingTimepoint AS $attribute.timeRange)) sub :
r$attribute.name(CAST(changingTimepoint AS $attribute.timeRange)) sub
        WHERE
            sub.$attribute.anchorReferenceName =
  $anchor.mnemonic\.$anchor.identityColumnName
   )~*/
            }
            else {
                if(attribute.isEquivalent()) {
/*~
LEFT JOIN
    e$attribute.name(0) $attribute.mnemonic
~*/
                }
                else {
/*~
LEFT JOIN
    $attribute.name $attribute.mnemonic
~*/
                }
/*~
ON
    $attribute.mnemonic\.$attribute.anchorReferenceName =
  $anchor.mnemonic\.$anchor.identityColumnName~*/
            }
            if(attribute.isKnotted()) {
                knot = attribute.knot;
                if(knot.isEquivalent()) {
/*~
LEFT JOIN
    e$knot.name(0) k$attribute.mnemonic
~*/
```

```
                }
                else {
/*~
LEFT JOIN
    $knot.name k$attribute.mnemonic
~*/
                }
/*~
ON
    k$attribute.mnemonic\.$knot.identityColumnName =
  $attribute.mnemonic\.$attribute.knotReferenceName~*/
            }
            if(!anchor.hasMoreAttributes()) {
                /*~;~*/
            }
        }
/*~
' LANGUAGE SQL;~*/
```

4.1.10. DropTiePerspectives.js, CreateTiePerspectiveLatest.js, CreateTiePerspectivePointInTime.js, CreateTiePerspectiveNow.js, CreateTiePerspectiveDifference.js

Antud mallid kustutavad ning loovad sidemete perspektiivid. Need on analoogsed ankru vastavate mallidega.

4.1.11. AddDescriptions.js

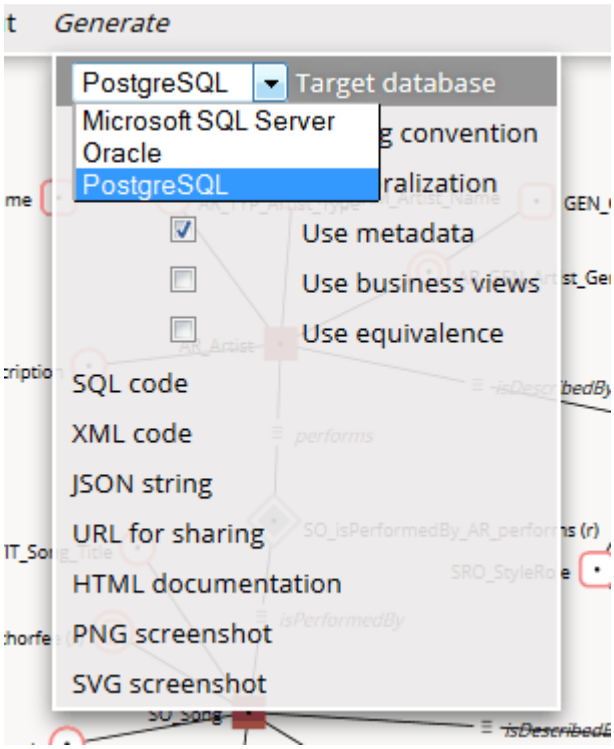
Antud mall lisab andmebaasiobjektidele kommentaarid, mis sisestati mudelisse.

Fragment AddDescriptions.js failist (sidemele kommentaari lisamine):

```
while (tie = schema.nextTie()) {
    if(tie.description &&
       tie.description._description &&
       tie.description._description.length > 0) {
/*~
COMMENT ON TABLE
_$tie.name
IS '$tie.description._description';
~*/
    }
    var role;
    while (role = tie.nextRole()) {
        if(role.description &&
           role.description._description &&
           role.description._description.length > 0) {
/*~
COMMENT ON COLUMN
_$tie.name\.$role.columnName
IS '$role.description._description';
~*/
        }
    }
}
```

4.2. Uuendatud ankurmodelleerimise rakendus

PostgreSQLi (ning pooliku Oracle) toega rakendus on peale töö valmimist saadaval aadressil http://apex.ttu.ee/AM/. Tegemist on rakenduse versiooniga, mida kasutatakse arendustöödeks (kasutades keskkonda Github). Samuti saadi ankurmodelleerimise teooria autorilt kinnitus, et peale generaatori valmimist pannakse PostgreSQLi toega generaatori versioon üles ankurmodelleerimise avalikule veebilehele (anchormodeling.com). Ankurmodelleerimise kodulehelt saab seejärel uuendatud generaatori versiooni ka alla tõmmata ning isiklikuks kasutamiseks kuhugi sobilikku kohta lahti pakkida (ZIP-fail).



Joonis 15. Rakenduse menüü

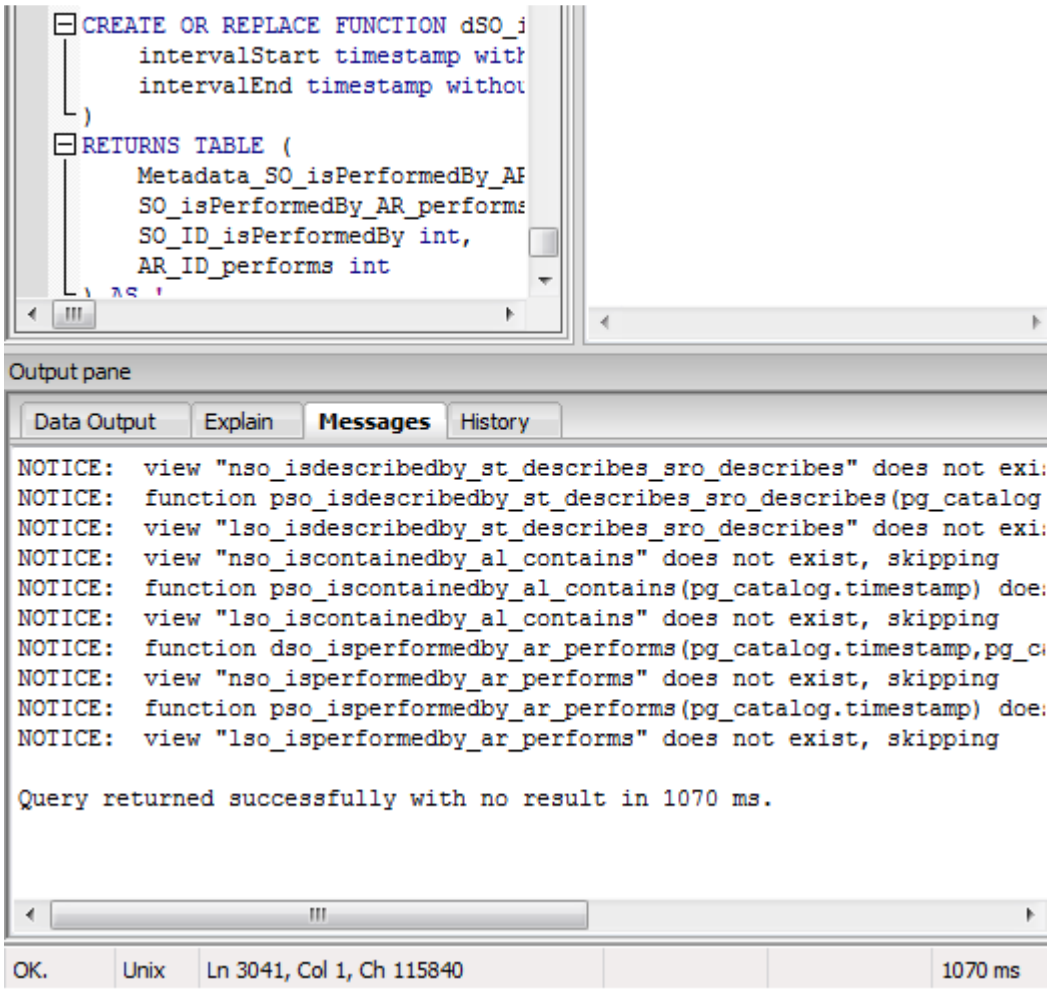
Joonisel 15 on toodud rakenduse menüü pärast PostgreSQLi toe lisamist. Autori hinnangul on tõenäoline, et ankurmodelleerimise avalikule veebilehele (http://www.anchormodeling.com/modeler/latest/) ilmub PostgreSQLi toega rakenduse versioon 2015. aasta esimeses kvartalis.

5. Generaatori katsetamine

Käesoleva töö testimist viisid läbi töö autor ja juhendaja. Testimise paremaks läbiviimiseks lõi autor ka PHP-rakenduse, mis andmebaasi testandmed sisestas ning vaadetest ja funktsioonidest andmeid pärida proovis.

5.1. Genereeritud lausete käivitamine PostgreSQLis

Ankurmodelleerimise rakendus genereeris näitemudeli põhjal üle 3000 rea koodi, mida autor püüdis pgAdmin tarkvara kasutades PostgreSQL andmebaasis käivitada. Selleks kopeeris autor koodi pgAdmin päringute tegemiseks mõeldud aknasse ning vajutas nuppu „*Execute query*". Andmebaasiobjektide genereerimine võttis natuke üle sekundi ning peale mitmete parandustööde teostamist mallides ühtegi kriitilist viga enam ei esinenud. Sellekohane ekraanitõmmis on toodud joonisel 16.



Joonis 16. Andmebaasiobjektide genereerimine PostgreSQLis

5.2. PHP-rakendus

Genereeritud koodi testimiseks lõi autor kasutajaliideseta PHP-veebirakenduse, mis koosnes alljärgnevatest osadest:

- konfiguratsioonifail, mis kirjeldas genereeritavate ridade hulka iga ankurmodelleerimise elemendi jaoks,

```
// conf
$typeNames = 150;
$styleRoles = 50;
$genders = 20;
$playing = 25000;
$styles = 1500;
$artists = 3000;
$albums = 2000;
$songs = 5000;
```

- funktsioonid, mis oli abiks testandmete väljamõtlemisel,

```
function generate($minLength = 2, $maxLength = 100, $method = 'alpha')
{
    if ($method == 'alpha')
        $chars = 'abcdefghijklm nopqrstuvwxyz ABCDEFGHIJKLMN OPQRSTUVWXYZ';
    elseif ($method == 'numeric')
       return rand($minLength, $maxLength);

    $length = strlen($chars);
    $output = '';

    for ($i = 1; $i <= rand($minLength, $maxLength); $i++) {
        $output .= $chars[rand(1, $length) - 1];
    }

    return trim($output);
}
```

- testandmete genereerimise fail, mis genereeris andmed ankrutele, sõlmedele, sidemetele ja atribuutidele, ja mis koosnes järgnevatest osadest:

keskkonna seadistamine:

```
set_time_limit(60);
require_once 'functions.php';
require_once 'conf.php';
$db = pg_connect("host=localhost options='--client_encoding=UTF8'
user=postgres dbname=postgres password=*****");
```

ankrute jaoks andmete genereerimine (esitaja näitel):

```
// AR_Artist
for ($i = 0; $i < $artists; $i++) {
    $query = "
        INSERT INTO AR_Artist (Metadata_AR) VALUES (
        ".generate(1, $artists, 'numeric')."); ";
    pg_query($db, $query);
}
```

atribuutide jaoks andmete genereerimine (esitaja nime näitel):

```
// AR_NAM_Artist_Name
for ($i = 0; $i < $artists; $i++) {
    $query = "
        INSERT INTO AR_NAM_Artist_Name (
            AR_NAM_AR_ID,
            AR_NAM_Artist_Name,
            Metadata_AR_NAM
        ) VALUES (
        ".($i + 1).",
        '".generate(1, 100)."',
        ".generate(1, $artists, 'numeric')."); ";
    pg_query($db, $query);
}
```

seoste genereerimine (esitaja, tema stiili ja stiilirolli näitel):

```
// ST_describes_AR_isDescribedBy_SRO_describes
for ($i = 0; $i < $styles; $i++) {
    $query = "
        INSERT INTO ST_describes_AR_isDescribedBy_SRO_describes (
            ST_ID_describes,
            AR_ID_isDescribedBy,
            SRO_ID_describes,
            ST_describes_AR_isDescribedBy_SRO_describes_ChangedAt,
            Metadata_ST_describes_AR_isDescribedBy_SRO_describes
        ) VALUES (
        ".($i + 1).",
        ".generate(1, $artists, 'numeric').",
        ".generate(1, $styleRoles, 'numeric').",
        NOW() - INTERVAL '".generate(1, 6400000, 'numeric')." second',
        ".generate(1, $styles, 'numeric')."); ";
    pg_query($db, $query);
}
```

- andmete otsimist teostav fail, mis proovis andmeid otsida kõigi generaatori poolt loodud funktsioonide abil (kokku 24 funktsiooni, mille hulka kuulusid ka perspektiivid, „atribuudi tagasikerimise funktsioonid” ja „korduva sisestatud väärtuse leidmise funktsioonid”):

andmete otsimine (esitaja „intervalli perspektiivi” kaudu):

```
// 1. dAR_Artist
for ($i = 0; $i < 2; $i++) {
    $query = "SELECT * FROM dAR_Artist(
        LOCALTIMESTAMP - interval '1 year',
        LOCALTIMESTAMP + interval '1 year',
        'abc'
    ); ";
    $val = fetchAll($query);
    if ($val === false) {
        echo 'Query failed:<br />' . $query . '<br />';
        echo pg_last_error($conn) . '<br /><br />';
    }
}
```

Tegemist ei ole üldlahendusega mistahes ankurmudeli andmebaaside testimiseks, vaid koodiga, mis on loodud konkreetse rakenduse struktuuri silmas pidades. Andmete lisamise ja otsimise aega ei mõõdetud.

PHP-rakenduse katsetamise käigus õnnestus mitu veergude ja ajaliste andmetüüpide mittevastavusega seotud viga üles leida. Peale vigade parandamist õnnestus nii andmete PostgreSQLi sisestamine kui ka otsimine. See viitab, et koodi genereerimine töös kasutatud mudeli jaoks õnnestus ning mallide tõlkimine on olnud tulemusrikas.

5.3. Disainiotsused

Järgnevalt selgitatakse mõningaid koodigeneraatori loomise käigus tehtud disainiotsuseid.

- Koodigeneraatori loomisel aluseks võetud näite-ankurmudeli loomisel kasutatud elemendid (muusikapala, autoritasu, stiil jne) pärinevad osaliselt või täielikult autori bakalaureusetööst (Saal, 2010).
- Autor programmeeris töö käigus MS SQL Serveri ja PostgreSQLi andmetüüpide mittetäieliku vastavustabeli. Autor ei koostanud täielikku vastavustabelit (kus oleks mõlema andmebaasisüsteemi kõik andmetüübid), sest autori arvates võiks mõni mõlemat mainitud andmebaasisüsteemi hästi tundev isik selleteemalise põhjaliku uuringu ise läbi viia, võttes arvesse ka mõlema andmebaasisüsteemi erinevad versioonid.
- Autor lisas malle tõlkides mallidesse kommentaare ja tühje ridu, parandamaks mallide loetavust ning võimaldamaks kolmandatel isikutel paremini mõista, mida loodud kood teeb. Autori hinnangul tuleks sama teha ka MS SQL Serveri mallidega (ning autor ütles seda ka Lars Rönnbäckile). Teiste andmebaasisüsteemide toe lisandumisel tuleks

ka nende mallid vormistada nii, et generaatoriga mitte väga lähedalt kursis olev isik oskaks mallikoodi sirvides oletada, mida kood teeb.

- Autor tükeldas osa malle väiksemateks osadeks, saamaks paremini aru nende loogikast. Samuti on tükeldamisest kasu, kui mallide tõlkimisega tegeleb mitu inimest (väiksem tõenäosus, et kaks inimest soovivad korraga sama malli muuta). Töö käigus selgus ka, et ühe malli tõlkimine võib koos testimisega võtta kuni kolm nädalat. Mallide tükeldamise soovitus on edastatud ka Lars Rönnbäckile. Autor hindab selle töö käigus mallide tõlkimisele kulunud ajaperioodi pikkuseks kolm kuud.
- Rakenduse testimiseks loodi PHP-rakendus, sest autoril on pikk PHP-programmeerimise kogemus. Lisaks on arvutisse PHP-arendusraamistiku installeerimine ning seadistamine äärmiselt lihtne ja mugav.
- Autor leidis generaatorit katsetades mallidest mõned ebakorrapärasused ning edastas selle info ka ankurmodelleerimise teooria autorile. PostgreSQLi jaoks loodud mallides neid vigu üldjuhul ei parandatud, sest autor soovis võimalikult sarnast tulemust MS SQL Serveri jaoks genereerituga. Näiteks „erinevuste perspektiivi" korral tekkis andmetüüpide konflikt, kui ajaloolise atribuudi puhul aega hoidvaks veeruks määrata midagi muud kui *timestamp* (selle vea autor parandas, et võimaldada perspektiivi käivitamist).

6. Arendusvaade

Järgnevalt tuuakse esile, mida võiks ankurmodelleerimise rakendusega seoses veel ära teha.

6.1. Täielik andmetüüpide vastavustabel

Enne ankurmudeli koostamist tuleks rakenduse menüüst valida andmebaasisüsteem, mille jaoks mudelit koostama asutakse (joonis 12). Kahjuks ei pruugi kasutaja sellest teadlik olla, mistõttu võiks ankurmodelleerimise rakendus andmebaasisüsteemi (ja sellega seotud andmetüüpide) valikut paremini esitada ja rõhutada.

Oletame, et ankurmudel loodi MS SQL Serveri jaoks, kasutades MS SQL Serveri spetsiifilisi andmetüüpe. Kui kasutaja tahaks sama mudeli alusel genereerida koodi PostgreSQLi jaoks, saab ta menüüs andmebaasisüsteemi vahetada (kuigi ka see ei pruugi kasutaja jaoks intuitiivne olla). Peale andmebaasisüsteemi vahetamist asendab rakendus MS SQL Serveri andmetüübid PostgreSQLi andmetüüpidega (joonis 13). Selleks on vaja vastavustabeleid. Näiteks PostgreSQL ei toeta vaikimisi andmetüüpi *tinyint* (seda saaks teostada domeeni kasutuselevõtuga, mis koosneks andmetüübist *smallint* + kitsendusest VALUE BETWEEN 0 AND 255).

Töö käigus selgus, et kui ankurmodelleerimise rakendus toetab n andmebaasisüsteemi, siis peab rakenduses olema kirjeldatud n * (n - 1) komplekti tüüpide asendusreegleid, võimaldamaks tüüpide teisendust igast (toetatud) andmebaasisüsteemist igasse (toetatud) andmebaasisüsteemi. Kui arvestada, et sama andmebaasisüsteemi erinevad versioonid toetavad erinevat hulka andmetüüpe, siis tuleks tüüpide asendusreeglid teha andmebaasisüsteemide versioonide kohta.

Ilma vastavustabeliteta oleks generaatori kasutamine selles mõttes piiratud, et kui ankurmudel loodi silmas pidades andmebaasisüsteemi A, siis peale uue andmebaasisüsteemi kasutuselevõttu tuleks mudel uuesti luua, pidades silmas andmebaasisüsteemi B (st muutes mudelielementide kirjeldustes andmetüüpe käsitsi).

Vastavustabelite loomise teeb keeruliseks see, et kui generaator toetaks näiteks viite andmebaasisüsteemi, tuleks programmeerida 5 * 4 = 20 komplekti andmetüüpide asendusreegleid. Lisaks peaks isik või isikute grupp, kes reegleid kirjutab, olema detailselt kursis kõikide toetatud andmebaasisüsteemidega, mis eeldab väga laia oskuste profiili. Samuti tuleb vastavustabeleid koos andmebaasisüsteemide uute versioonide väljatulekuga juurde lisada.

6.2. Kasutajaliides

Ankurmodelleerimise rakenduse kasutajaliides võiks mõningaid valikuid paremini esitada ja selgitada. Samuti sisaldab rakendus valikuid, mida tõenäoliselt ei kasutata. Autori arvates tuleks läbi viia analüüs, mille käigus palutakse erinevatel ankurmodelleerimise teooriaga mittekursis olevatel isikutel rakendust kasutada, ning uurida, kuidas saaks kasutajaliidest selgemaks ja intuitiivsemaks teha.

6.3. XML

Ankurmodelleerimise rakendus võimaldab loodud mudelit arvuti kõvakettale salvestada XML-formaadis. Samuti loob generaator tabeli, kus mudelit antud formaadis hoitakse. Tabeli alusel saab jälgida mudelis tehtud muudatusi.

XML-ist saab tuletada ankrud, atribuudid, sõlmed ja sided. Samuti saab sellest genereerida andmebaasiobjektide kustutamist ja kopeerimist võimaldavad skriptid.

Käesoleva töö raames jäeti XMLiga seotud mallid tõlkimata. Küll aga veendus autor, et nende tõlkimine on võimalik. Põhjus seisneb selles, et autori arvates tuleks XMLi kasutamisest loobuda ning ankurmodelleerimise rakendus üle viia kaasaegsemale JSON-formaadile.

6.4. Drop laused

Ankurmodelleerimise rakendus ei genereeri ankurmudeli elementidele DROP-lauseid (v.a. perspektiivid). Rakenduse menüüle tuleks juurde teha valik, kus saaks määrata, mida reaalselt genereeritakse. Autori ettepanek oleks lisada menüüsse järgnevad (kuid mitte ainult) valikud:

- võimalus määrata, kas genereeritakse DROP-laused või mitte,
- võimalus määrata, kas genereeritakse skeemi jälgimine, kasutades selleks andmebaasis hoitavat XML dokumenti või mitte.

6.5. Operatiivandmed

Autori arvates tuleks teostada analüüs, et selgitada, kas ja kuidas saab ankurmodelleerimine hakkama operatiivandmetega ning kuidas oleks luua sellistele andmetele operatiivandmete haldamise rakendust. Analüüsi tulemused võiks huvi pakkuda paljudele organisatsioonidele, mis vajaksid kuuenda normaalkuju ja ankurmodelleerimise poolt pakutud võimalusi.

6.6. Kitsendused

Autorile pakuks huvi analüüs, milliseid kitsendusi oleks võimalik või mõistlik ankurmodelleeritud andmebaasis jõustada ning kuidas seda oleks kõige parem teha.

6.7. Sisula

Mallimootor Sisula eemaldab genereeritud koodist kõik tühjad read, mis teeb koodi küll lühemaks, aga halvendab selle loetavust. Autori ettepanek oleks uurida, milliseid täiendusi tuleks teha Sisulasse, et genereeritud koodi loetavust parendada.

6.8. Trigerite hierarhia

Ankurmodelleerimise teooria kohaselt tuleb andmete lisamisel andmebaasi teostada kontrollid, mis nõuavad andmete ajutist salvestamist ja sorteerimist. SQL-andmebaasides saab selliseid kontrolle teostada trigeritega. Käesoleva töö raames programmeeriti vajalikud trigerid atribuutide jaoks ning see võttis äärmiselt palju aega. Lahendus koosneb atribuudi puhul kolmest trigerist iga atribuudi kohta. Trigerite loogikat ei programmeeritud ankrute ja sidemete jaoks.

Autori arvates tuleks teostada analüüs, selgitamaks, kas trigerite abil teostatud tegevusi saaks teha kuidagi lihtsamalt, arusaadavamalt ning jätkusuutlikumalt. Samuti tuleks läbi viia test, et tuvastada, kuidas käitub loodud hierarhia, kui andmebaasi lisada korraga miljoneid ridu (ning soovituslikult mitmest andmeallikast korraga).

6.9. Koodi refaktoorimine

Ankurmodelleerimise rakenduse koodigeneraator baseerub mallidel, mis kahjuks on enamasti väga pikad, paljude tingimuste (IF-lausete) ahelad ning ilma koodi lähemalt selgitavate kommentaarideta. Mõned kommentaarid on, aga need kirjeldavad, MIDA kood teeb, selle asemel et kirjeldada, KUIDAS kood seda teeb.

Käesoleva töö autor tükeldas tõlkimise käigus mitmed PostgreSQLi mallid väiksemateks osadeks ning lisas loetavuse parandamiseks koodi ka tühje ridu ja kommentaare, aga seal on veel suur töö ära teha. Sarnane töö tuleks ette võtta ka MS SQL Serveri mallidega.

Samuti esineb koodis mõnes kohas SQL-lauseid, kus kasutatakse veergude loetelu asemel tärni (*). Sellised päringud võivad hakata andma valesid vastuseid, kui andmebaasi struktuur muutub, seetõttu tuleks igal pool kasutusele võtta päringud, kus on SELECT klauslis kirjeldatud veerud. Lisaks on sellised päringud ka ennast paremini dokumenteerivad, sest juba päringu lausest selgub, kui palju ja millised veerud on selle päringu tulemuses.

Autori hinnangul tuleks kaaluda mallide täielikku refaktoorimist. See lihtsustaks oluliselt generaatorile teiste andmebaasisüsteemide toe lisamist.

6.10. Dokumentatsioon

Kui välja arvata mõned teadusartiklid, siis puudub ankurmodellerimise rakendusel dokumentatsioon. Ilma selleta on äärmiselt keeruline teha rakenduse koodis parandusi või lisada rakendusele teiste andmebaasisüsteemide tuge.

Autori arvates tuleks ankurmodelleerimise rakendusele koostada põhjalik dokumentatsioon, mis selgitab detailselt, kuidas mudelielementidest saab reaalne kood. Selline dokument peaks vastama küsimustele MIKS ja KUIDAS. Seni on rakenduse parendamiseks tehtav töö pärsitud.

Käesoleva töö läbiviimisel võeti aluseks olemasolev MS SQL Serveri tugi. Antud meetod eeldas autorilt väga palju uurimistööd, katsetamist ja analüüsi, mida põhjaliku dokumentatsiooni olemasolu korral oleks saanud (vähemalt osaliselt) vältida.

6.11. Oracle tugi

Autor märkas, et rakenduse lähtekoodile on proovitud lisada andmebaasisüsteemi Oracle tuge. Kahjuks tundub, et toe arendustööd on töö valmimise ajaks seiskunud.

Autori hinnangul võiks asjast huvitatud inimesed arendustöödega jätkata. Autor oleks nõus selles projektis nõustajana osalema. Käesolev magistritöö oskab tõenäoliselt anda mitmeid suunitlusi tööde läbiviimiseks.

6.12. Jõudluse testimine ja parandamise viiside otsimine

Autori arvates võiks mõne tulevase bakalaureusetöö teema olla ankurmodelleeritud andmebaasis toimuvate operatsioonide jõudluse võrdlus analoogse, kuid maksimaalselt viiendal normaalkujul oleva andmebaasi jõudlusega. Võrdlus võiks võtta kaks lähenemist:

operatiivandmete lähenemine ja andmeaida lähenemine. Antud uuring aitaks vastata küsimusele, kas ankurmodelleerimise kasutamine oleks põhjendatud.

Samuti tuleb iga andmebaasisüsteemi kontekstis uurida, milliseid spetsiifilisi ankurmudeli andmebaaside jaoks sobivaid jõudluse parandamise võimalusi see pakub.

6.13. Muud tähelepanekud

Autoril tekkisid töö käigus alljärgnevad mõtted.

- Andmetüüpe ei valideerita: modelleerimise käigus võib ankurmudeli elemendi andmetüübiks määrata suvalise sõne. Vaba sisestus tähendab, et modelleerijal on liiga suur võimalus teha vigu. Rakendus võiks vähemasti kontrollida, kas menüüs valitud andmebaasisüsteem, mille jaoks modelleerimist teostatakse, toetab sisestatud andmetüüpi. Parem oleks, kui andmetüüpe saaks valida liitboksist. Liitboks peab pakkuma valiku vastavalt sellele, millise andmebaasisüsteemi jaoks mudelit luuakse. Selleks peaksid muidugi olema süsteemis kirjeldatud andmebaasisüsteemide andmetüübid ning olemas olema andmetüüpide (vastavus)tabelid.
- PostgreSQL konverteerib tabelite, võtmete, funktsioonide jms nimed, mis on regulaarsed identifikaatorid, väiketähtedeks. Fraasist *AR_NAM_Artist_Name* saab niisiis *ar_nam_artist_name*. Andmebaasiobjekte genereerides saaks objektide nimed ümbritseda jutumärkidega, vältimaks väiketähtedeks konverteerimist, aga päringuid koostades enam jutumärkideta nimesid kasutada ei saaks. Seetõttu ei ole mõistlik jutumärke kasutada, ning tuleb leppida väikeste tähtedega. Ankurmodelleerimise teooria autorit on kitsaskohast teavitatud, ning ta kinnitas, et uurib võimalikke variante selle küsimuse lahendamiseks.
- Ankurmudeli baasil genereeritud koodis puuduvad laused, mis käsitlevad ligipääsu- ja käivitamisõiguseid loodud andmebaasiobjektidele. Mõistlik oleks viia läbi analüüs, mille tulemusel võiks näiteks selguda, kas keelata ligipääs baastabelitele, aga lubada see vaadetele.
- Autori hinnangul võtab ankurmodelleerimisele uue andmebaasisüsteemi toe lisamine aega 3-6 kuud, sõltuvalt lisamisega tegelevate isikute teadmistepagasist ning koostööst ankurmodelleerimise rakenduse arendamisega seotud isikutega. Seda on „hobikorras” tegemiseks natuke liiga palju, mis võib selgitada ka Oracle arenduste seiskumist.

Kokkuvõte

Käesoleva magistritöö eesmärk oli realiseerida olemasolevale ankurmodelleerimise veebipõhisele vahendile generaator PostgreSQL andmebaaside loomise jaoks. Eesmärgi täitmiseks kavandati ja seejärel realiseeriti PostgreSQLis ankurmudel raadiojaama esitlusloendi kohta, mis sisaldas kõiki võimalikke ankurmodelleerimise konstruktsioone. Paralleelselt teostati tarkvaraarendus, mis koosnes suure hulga MS SQL Serveri jaoks loodud mallide viimisest PostgreSQLi jaoks sobivale kujule. Genereeritud koodi testimiseks loodi PHP-veebirakendus, millega genereeriti testandmed ning püüti näitemudeli alusel loodud funktsioonidega andmeid otsida. Magistritöö tulemusena leiti, et ankurmudelite realiseerimine PostgreSQLis on võimalik, toodi esile tekkinud kitsaskohad ning koostöös ankurmodelleerimise ideoloogia rajajaga täiendati modelleerimistarkvara. Autori arvates on ankurmodelleerimise abil andmebaaside loomine paljulubav lähenemine andmeaitades ning miks mitte ka operatiivandmebaasides kasutamiseks. See lubab muuhulgas hästi toime tulla puuduvate andmetega, säilitada ajaloolisi andmeid ning teha andmebaasi struktuuri muudatusi nii, et see võimalikult vähe andmebaasi kasutajaid häiriks.

Järgnevalt on nimetatud töö kõige olulisemad tulemused.

- Ankurmodelleerimise rakendusele lisandus PostgreSQLi tugi (mis vajab veel täiendamist, kuid kõige olulisemad asjad on teostatud).
- Mitmed ankurmodelleerimise aspektid, mida pole kusagil dokumenteeritud, on töös lahti seletatud.
- Loodi kontakt ühe ankurmodelleerimise teooria rajaja ning modelleerimisvahendi autoriga, Lars Rönnbäckiga.
- Toodi esile kitsaskohad, mis takistasid generaatorile PostgreSQLi (ja tõenäoliselt ka teiste andmebaasisüsteemide) toe lisamist.
- Kirjeldati tegevusi ankurmodelleerimise rakenduse edasiseks paremaks muutmiseks.

Töö kokkuvõtteks võib öelda, et tutvumine ankurmodelleerimisega oli väga huvitav ja hariv. Samuti loodi töö käigus mitmeid kontakte, mida (nii autoril kui ka juhendajal) oleks võimalik tulevikus ära kasutada. Autor usub, et jätkab peale magistritöö valmimist iseseisvalt generaatori täiendamist, et lahendada küsimused, mis tööst välja jäid.

Ankurmodelleerimise rakendusele Oracle toe lisamine (ning *concurrent-reliance-temporal* generatsiooni PostgreSQL formaati viimine) on töö valmimise ajaks küll seiskunud, kuid autor loodab, et valminud magistritöös õpitut aluseks võttes oleks võimalik teiste

andmebaasisüsteemide toe lisamisele ka hoogu juurde anda (eriti peale käesoleva töö inglise keelde tõlkimist ning avaldamist).

Autor soovib tänada käesoleva magistritöö juhendajat Erki Eessaart igakülgse osutatud abi eest.

Summary

The goal of this thesis was to implement a code generator based on anchor models. The generated code must implement anchor models in PostgreSQL database management system. The code generator would accompany an existing web-based anchor modeling system and would be created by using its existing framework for creating code generators.

To achieve the goal, the author firstly studied possibility of implementing anchor models in PostgreSQL. For this purpose an anchor model of a radio station's set list was designed and then implemented in PostgreSQL. The model used all the elements of anchor models. In parallel, the author created a large number of templates for generating code for PostgreSQL based on the existing templates for Microsoft SQL Server. It was not an easy task because the code generation approach of anchor modeling and templates for MS SQL Server are not well documented. The author also created a PHP web application for testing the database that is produced based on the anchor model of a radio station's set list. This application generated test data and tried to search specific data from created tables by using generated functions. As a result of this work it was found out that implementing anchor models in PostgreSQL is possible. In addition, the potential bottlenecks were identified and the modeling software was updated in cooperation with the author of the anchor modeling approach. The author of this thesis thinks that anchor modeling is a promising method to be used in data warehouses and perhaps also in operational databases. It deals well with missing information, supports elegantly recording of historic attribute/tie values, and allows us to change the structure of database with a minimal disruption.

The most important results of this study are the following.

- The anchor modeling system now supports generating code for PostgreSQL. This still needs work but the most crucial parts are complete.
- Several non-documented aspects of anchor modeling have been explained.
- The contact was made with Lars Rönnbäck, one of the creators of the anchor modeling theory and system.
- Bottlenecks of adding PostgreSQL support to the generator were identified. These may also apply to other database management systems.
- A number of activities still to be done regarding the modeling system were listed.

In short, this introduction to anchor modeling was very interesting and educational. In the course of this work several new contacts were made that may prove useful to the author or

the instructor. The author believes that he will continue making the improvements to the generator even after finishing the thesis to resolve the issues not covered in this work.

The efforts to add Oracle support and concurrent-reliance-temporal code generation for PostgreSQL to the anchor modeling application have unfortunately stopped for the time being. Nevertheless, the author hopes that this thesis can be used as a foundation and an accelerator in adding the support of additional database management systems, especially after translating and publishing it in English.

The author wishes to thank Erki Eessaar, the instructor of this work, for the extensive help that he provided.

Kasutatud materjalid

1. Australian Passport Office. Sex and Gender Diverse Passport Applicants. [WWW] https://www.passports.gov.au/web/sexgenderapplicants.aspx (05.01.2015)
2. Bartolini, G., 2009. Data warehousing with PostgreSQL. [WWW] https://wiki.postgresql.org/images/3/38/PGDay2009-EN-Datawarehousing_with_PostgreSQL.pdf (29.11.2014)
3. Database normalization. Wikipedia. [WWW] http://en.wikipedia.org/wiki/Database_normalization (09.01.2015)
4. DB-Engines Ranking. [WWW] http://db-engines.com/en/ranking (11.01.2015)
5. Eessaar, E., 2014. *Andmebaasi loogilise disaini tulemuse parandamine ja headuse kontrollimine*. Tallinn: TTÜ Kirjastus
6. Eessaar, E., Saal, E., 2013. Evaluation of Different Designs to Represent Missing Information in SQL Databases. *Innovations and Advances in Computer, Information, Systems Sciences, and Engineering*, Lecture Notes in Electrical Engineering, Vol.152: International Conference on Systems, Computing Sciences and Software Engineering (SCSS 11). Eds. K. Elleithy, T. Sobh. Springer. pp. 173-187.
7. Falkenberg, E.D., 1992. Evolving Information Systems: Beyond Temporal Information Systems. *Database and Expert Systems Applications*, pp. 282-287. Springer Vienna.
8. Microsoft SQL Server. [WWW] http://msdn.microsoft.com/en-us/library/bb545450.aspx (09.01.2015)
9. PostgreSQL 9.3.5 Documentation. [WWW] http://www.postgresql.org/docs/9.3/static/index.html (09.01.2015)
10. Potter, E., 2013. *Ankurmodelleerimise ja Oracle Workspace Manageri võrdlus temporaalsete andmete haldamisel SQL-andmebaasides*. Tallinna Tehnikaülikool
11. Rational Rose Modeler. [WWW] http://www-03.ibm.com/software/products/en/rosemod (09.01.2015)
12. Riigiportaali „eesti.ee“ tehnoloogilise raamistiku versioonimise tarkvara tellimine. [WWW] https://www.ria.ee/riigiportaali-eestiee-versioonimise-teenuse-osutami/ (29.11.2014)
13. Rönnbäck, L., 2010.Three Concepts of Time in Anchor Models. [WWW] http://www.anchormodeling.com/wp-

content/uploads/2010/08/Three_Concepts_of_Time_in_Anchor_Models.pdf (07.01.2015)

14. Rönnbäck, L., Regardt, O., Bergholtz, M., Johannesson, P., Wohed, P., 2010a. Agile Information Modeling in Evolving Data Environments. *Data & Knowledge Engineering*, vol. 69, no. 12, pp. 1229-1253. [WWW] http://www.anchormodeling.com/wp-content/uploads/2011/05/Anchor-Modeling.pdf (19.11.2014)
15. Rönnbäck, L., Regardt, O., Bergholtz, M., Johannesson, P., Wohed, P., 2010b. Anchor Modeling: Naming Convention. [WWW] http://www.anchormodeling.com/wp-content/uploads/2010/09/AM-Naming.pdf (07.01.2015)
16. Rönnbäck, L., Regardt, O., Bergholtz, M., Johannesson, P., Wohed, P., 2010c. From Anchor Model to Relational Database. [WWW] http://www.anchormodeling.com/wp-content/uploads/2010/09/AM-RDB.pdf (10.10.2014)
17. Rönnbäck, L. sisula. [WWW] https://github.com/Roenbaeck/sisula (29.11.2014)
18. Saal, E., 2010. *Puuduvate andmete esitamine SQL-andmebaasides.*Bakalaureusetöö. Tallinna Tehnikaülikool
19. SQL: 2011. Wikipedia. [WWW] http://en.wikipedia.org/wiki/SQL:2011 (09.01.2015)
20. What is Table Elimination? [WWW] https://mariadb.com/kb/en/mariadb/documentation/optimization-and-tuning/query-optimizations/table-elimination/what-is-table-elimination/ (28.11.2014)
21. Privaatne kommunikatsioon ühe ankurmodelleerimise teooria rajaja Lars Rönnbäckiga.