TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Susanna Peek 162848 IABM

INFOSÜSTEEMI PÕHIOLEMITE SEISUNDITE ESITAMINE SQL-ANDMEBAASIDES

Magistritöö

Juhendaja: Erki Eessaar
PhD

Tallinn 2019

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.

Autor: Susanna Peek

07.05.2019

Annotatsioon

Käesoleva magistritöö eesmärgiks on väljaselgitada võimalikud disainid infosüsteemi põhiolemite seisundite registreerimiseks SQL-andmebaasides, anda nendest struktureeritud ülevaade ning uurida nende headust mõningate mõõdetavate kvaliteedi aspektide nagu andmemaht, töökiirus ja keerukus kontekstis. Töö põhitulemiks on disainide kataloog, millele andmebaasi arendajad saavad disainivalikute tegemisel ning põhjendamisel toetuda.

Töö teoreetiline osa koosneb disainide disainimustrite formaadis kirjeldustest. Igas kirjelduses esitatakse nii lahendus SQL-andmebaaside jaoks üldiselt kui ka konkreetne tehniline lahendus e strateegia andmebaasisüsteemi PostgreSQL (11) jaoks. Kataloogi koostamise aluseks on erinevatest materjalidest leitud lahendused. Võimalike lahenduste kirjapanemiseks kasutatan mustri formaati, sest see muudab kirjelduse paremini loetavaks ning parandab erinevate lahenduste võrreldavust. Kataloogis on kokku kuus disaini. Igal disainil on null või rohkem variatsiooni.

Töö praktiline osa koosneb kõigi väljaselgitatud lahenduste realiseerimisest PostgreSQL (11) andmebaasis ning nende omavahelistest võrdlustest andmemahtude, päringute ja andmemuudatuste täitmiskiiruste, seisundite hulga muutmise ning vastavate andmebaasikeele lausete keerukuse seisukohast. Antud analüüsi läbiviimiseks kasutan kõikides andmebaasides testandmeid samasuguste omadustega ja hetkeseisundiga olemite (täpsemalt tellimuste) kohta ning seejuures võtan arvesse ka erinevaid andmehulki. Eksperimendi tulemuste põhjal täiendan töö teoreetilises osas esitatud disainide kataloogi.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 126 leheküljel, 8 peatükki, 34 joonist, 23 tabelit.

Abstract

Representing the Status of the Main Entities of an Information System in SQL Databases

The goal of this Master's thesis is to find possible designs for registering the state of the main entities of an information system in SQL databases, give structured overview of these designs, and to study their suitability in the context of various measurable quality aspects like data size, performance, and complexity. The main result of this work is a catalog of designs that database developers can use to choose a design and justify their decisions. In order to make the right choice, a developer needs to be aware of the advantages and disadvantages of each design. At the same time, one must understand that no design is universal and can fit everywhere. Thus, one needs to know the specific tasks that the designed system will have to accomplish and take it into account while choosing the best design.

As of spring 2019, SQL database management systems (DBMSs) are still widely used for managing data in information systems. This is the reason why the work concentrates to the design of SQL databases.

The theoretical part of the work consists of descriptions of designs. The designs are presented by using a format of design patterns. Each description provides solution to the SQL databases in general and a concrete solution (strategy) for the PostgreSQL (11) DBMS. I selected the DBMS because of its popularity, abundance of features, and good adherence to the SQL standard. This part of the study is based on information that I found from different materials. A pattern format is used because it improves readability and comparability. In total the design catalog presents six different designs. Each of these has zero or more variations.

The practical part of the work consists of implementation of all the found designs in a PostgreSQL (11) database and conducting measurements. I measure data size, performance of queries and data modification operations, and complexity of the corresponding database language sentences. I also measure complexity of implementing the designs in the first place and complexity of changing the set of supported states in case of a main entity type. In order to perform the practical work, I used test data about

the same entities (more specifically, orders) in case of different designs. I also took into account different data volumes. In order to find out as to whether there is a linear dependency between the increase of the execution time (i.e., decrease of performance) and the increase in the data volume I conducted performance measurements based on three different data sizes. Here (https://github.com/susannapeek1/seisundid_SQL) is all the relevant SQL code of the experiment.

The results of the experiment revealed, unsurprisingly, that each design has advantages and disadvantages. However, based on the analysis carried out in the work, the *(Separate) Table for status classifications* and the *Temporal columns* were somewhat better. Therefore, if you prefer smaller data size and less complex SQL code, then you could look at these designs. At the same time if one of the most important functions of the database application is to find all the entities that are in a particular state, then in case of *Presenting states as subytupes of a main entity type* design the execution of this query is much faster than in case of others. If one is interested in simplicity of queries that find the state of an entity and a good performance in case of this, then *Vectorcoding* is a good choice. Quite good performance in case of all the tested queries is also a characteristic of design *Columns with the Boolean type*. The problems of design *Derive status values from data about an entity and its relations* are large data size and slow queries. At the same time, it should not be forgotten that a database that is created based on this design contains much more information about the state of entities compared to other designs. As one can see, many suggestions can be given. Therefore, it is important that database developers would early one find the important criteria based on which to evaluate the designs. The results of this thesis could help in the evaluation process.

There are multiple options of future work. One line of research could be to extend it to NoSQL DBMSs that have gained popularity in recent years. It would be also interesting to find out what performance impact could have table partitioning in PostgreSQL in case of the investigated designs. Another possible development is to use the designs presented in this thesis to compare how difficult it is to build an application on these in a particular programming language. It is possible that the results will lead to the changes in the catalog. The fourth possibility is to choose the best possible design for a particular system by using, for instance, Saaty's Analytic Hierarchy Process. The requirements to the system determine the relative importance of evaluation criteria. The measurement results from this work could be used to compare the designs in terms of some of the criteria.

The thesis is in Estonian and contains 126 pages, 8 chapters, 34 figures, 23 tables.

Lühendite ja mõistete sõnastik

DBMS	Database Management System
JSON	JavaScript Object Notation
JSONB	Binary JSON
MB	Megabait
SLOC	Source Lines of Code
SQL	Structured Query Language
XML	Extensible Markup Language

Sisukord

1 Sissejuhatus 13
2 Teoreetiline taust 15
3 Seotud uuringud 18
4 Erinevad disainid seisundite esitamiseks SQL-andmebaasides 25
4.1 Disaini kirjelduse struktuur 26
4.2 Korduvad osad 31
4.3 Seisundiklassifikaatori tabel 32
4.4 Temporaalsed veerud 41
4.5 Tõeväärtustüüpi veerud 48
4.6 Vektorkodeerimine 55
4.7 Seisundite esitamine põhiolemitüübi alamtüüpidena 62
4.8 Seisundite tuletamine olemite ja seoste kohta registreeritud andmetest 72
5 Eksperiment 79
5.1 Eksperimendi eesmärk 79
5.2 Eksperimendi kirjeldus 80
5.3 Kasutatavad andmebaasisüsteemid 82
5.4 Andmebaasi realiseerimine 82
5.5 Testandmed 83
5.6 Eksperimendis testitavad operatsioonid 85
5.6.1 Andmete lugemine 85
5.6.2 Andmete muutmine 96
5.6.3 Uue seisundi lisamine 96
5.6.4 Seisundi eemaldamine 99
6 Eksperimendi tulemused 102
7 Tulemuste analüüs ja järeldused 108
7.1 Tulemuste analüüs 108
7.1.1 Andmemahud 108
7.1.2 Päringute ja andmemuudatus operatsioonide kiirused 109
7.1.3 Lausete keerukus 112
7.1.4 Pearsoni korrelatsiooni koefitsent 114
7.2 Järeldused 114

8 Kokkuvõte 118
Kasutatud kirjandus 121
Lisa 1 – allikate otsimiseks kasutatud otsingustringid 127
Lisa 2 – SQL koodi formaatimise näide leidmaks füüsilist koodiridade arvu 128

Jooniste loetelu

Joonis 1. Töö valdkonda kirjeldav mõistekaart. 17
Joonis 2. Tellimuse seisundidiagramm. 28
Joonis 3. Seisundiklassifikaatori disaini andmebaasi diagramm. 37
Joonis 4. Indeksi loomise lause. 37
Joonis 5. Trigeri loomise laused. 37
Joonis 6. Temporaalsed veerud disaini andmebaasi diagramm. 44
Joonis 7. Kitsenduse loomise lause. 44
Joonis 8. Osaliste indeksite loomise laused. 45
Joonis 9. Domeeni loomise lause. 45
Joonis 10. Tõeväärtustüüpi veerud disaini andmebaasi diagramm. 51
Joonis 11. Kitsenduse loomise lause. 51
Joonis 12. Trigeri loomise laused. 52
Joonis 13. Osaliste indeksite loomise laused. 52
Joonis 14. Vektorkodeerimine disaini andmebaasi diagramm. 58
Joonis 15. Kitsenduse loomise lause. 58
Joonis 16. Kitsenduse loomise lause. 58
Joonis 17. Trigeri loomise laused. 58
Joonis 18. Dekodeerimise funktsiooni loomise lause. 59
Joonis 19. Funktsioonil põhineva indeksi loomise lause. 59
Joonis 20. Kodeerimise funktsiooni loomise lause. 59
Joonis 21. Seisundite esitamine põhiolemitüübi alamtüüpidena konseptuaalne andmemudel. 62
Joonis 22. Seisundite esitamine põhiolemitüübi alamtüüpidena disaini andmebaasi diagramm. 67
Joonis 23. Kitsenduste trigeri loomise laused. 68
Joonis 24. Trigeri loomise laused. 69
Joonis 25. Seisundite tuletamine olemite ja seoste kohta registreeritud andmetest disaini andmebaasi diagramm. 76
Joonis 26. Osaliste indeksite loomise laused. 76
Joonis 27. Klassifikaatori väärtuste registreerimise lause. 83
Joonis 28. Tellimuste andmete genereerimise lause. 84

Joonis 29. Alternatiivne koondandmete leidmise päring disaini nr 1 jaoks. 95
Joonis 30. Alternatiivne koondandmete leidmise päring disaini nr 1 jaoks. 95
Joonis 31. Alternatiivvne koondandmete leidmise päring disaini nr 4 jaoks. 95
Joonis 32. Tellimuste elutsükleid kirjeldav seisundidiagramm peale uue seisundi lisamist. 97
Joonis 33. Tellimuste elutsükleid kirjeldav seisundidiagramm peale seisundi eemaldamist. 99
Joonis 34. Pearsoni korrelatsiooni koefitsiendi arvutamine Excelis (disaini 1 põhjal lahendatava ülesande S1 korral). 106

Tabelite loetelu

Tabel 1. Nõuete kirjeldused. 29
Tabel 2. Seisundiklassifikaatori tabel lahendused nõuetele. 38
Tabel 3. Temporaalsed veerud lahendused nõuetele. 45
Tabel 4. Tõeväärtustüüpi veerud lahendused nõuetele. 52
Tabel 5. Vektokodeerimine lahendused nõuetele. 59
Tabel 6. Seisundite esitamine põhiolemitüübi alamtüüpidena lahendused nõuetele. 69
Tabel 7. Seisundite tuletamine olemite ja seoste kohta registeeritud andmetest lahendused nõuetele. 77
Tabel 8. Andmete lugemiseks S1 kasutatavad päringud. 86
Tabel 9. Andmete lugemiseks S2 kasutatavad päringud. 86
Tabel 10. Andmete lugemiseks S3 kasutatavad päringud. 88
Tabel 11. Andmete lugemiseks S4 kasutatavad päringud. 89
Tabel 12. Andmete lugemiseks S5 kasutatavad päringud. 92
Tabel 13. Andmete muutmiseks kasutatavad päringud. 96
Tabel 14. Uue Seisundi lisamise jaoks uued seisundiväärtused Vektorkodeerimine disainile. 98
Tabel 15. Seisundi eemaldamine jaoks uued seisundiväärtused Vektorkodeerimine disainile. 101
Tabel 16. Andmebaasi maht erinevate disainide korral (MB). 102
Tabel 17. Andmebaasi andmemaht ilma indeksite erinevate disainide korral (MB). .. 103
Tabel 18. Päringute ja andmemuudatus operatsioonide geomeetrilised keskmised kiirused (sekundites) andmehulga 1 korral. 103
Tabel 19. Päringute ja andmemuudatus operatsioonide geomeetrilised keskmised kiirused (sekundites) andmehulga 2 korral. 104
Tabel 20. Päringute ja andmemuudatus operatsioonide geomeetrilised keskmised kiirused (sekundites) andmehulk 3 korral. 104
Tabel 21. Päringute ja andmemuudatus operatsioonide keerukus erinevate disainide korral (koodiridade arv). 105
Tabel 22. Seisundite lisamise ja kustutamis operatsioonide ning disainide realisatsiooni lausete keerukus erinevate disainide korral (koodiridade arv). 105
Tabel 23. Pearsoni korrelatsiooni koefitsent iga disaini-ülesande paari korral. 106

1 Sissejuhatus

Loodava infosüsteemi edukus sõltub suuresti analüüsifaasis kogutud nõuete ja formuleeritud lähteülesande põhjal tehtavatest valikutest. Seepärast on oluline, et kõik vastu võetud otsused uue süsteemi loomiseks on läbimõeldud ja põhjendatavad. Kuna infosüsteemi üheks oluliseks osaks on andmebaas, siis mõjutavad andmebaasi loomiseks tehtavad valikud suuresti selle süsteemi kasutamise efektiivsust ja mugavust [1]. Näiteks võivad need oluliselt mõjutada süsteemis läbiviidavate andmekäitlusoperatsioonide kiirust või keerukust ning süsteemi muutmise keerukust ja kulukust.

Infosüsteemi kohta nõuete kogumisel on oluliseks ülesandeks leida põhiolemitüübid e põhiobjektid. Nendele vastavaid andmeid hakatakse talletatama andmebaasis. [1] Näiteks e-poe infosüsteemi üheks põhiolemitüübiks on *Tellimus*. Tellimuste elutsükleid kirjeldav seisundidiagramm (olekumasin), kirjeldab kõikvõimalikud infosüsteemi jaoks huvipakkuvad tellimuse seisundid. See millises seisundis on parajasti tellimus, määrab, kuidas selle tellimuse puhul tuleb infosüsteemil reageerida erinevatele võimalikele sündmustele (näiteks soovile tellimus katkestada). [2] Näiteks võivad ärireeglid määrata, et tellimuse saab katkestada ainult siis, kui seda pole veel tarnijale edasi saadetud. Seega on loomulik, et infosüsteemil peab olema iga tellimuse korral informatsioon selle hetkeseisundist ning võimalik, et ka seisundite muutumise ajaloost.

Andmebaasi disaini valikud sõltuvad andmebaasi loomiseks kasutatava andmebaasisüsteemi valikust. Käesoleva töö kirjutamise ajal on SQL (*Structured Query Language*) andmebaasikeele abil loodud andmebaasid infosüsteemides endiselt valdavad. Andmebaasisüsteemide populaarsuse indeksi [3] kohaselt on 2019. aasta mais esikümnes kuus süsteemi, mis võimaldavad andmebaasi loomist SQL andmebaasikeele abil ja selle keelega määratud ehitusplokke kasutades. SQL-andmebaasi põhiline ehitusplokk on tabel.

Küsimus, millist tabelite disaini võiks kasutada põhiolemite seisundite hoidmiseks SQL-andmebaasides, on tuttav kindlasti enamikele infosüsteemi arendusega tegelevatele inimestele. Kuna antud teema on aktuaalne enamike andmebaasirakenduste loomise korral, siis sellest tulenevalt on tänaseks päevaks välja mõeldud väga mitmeid võimalusi kuidas seda teha. Siiski võib paljusid neid pidada mõne põhilahenduse variatsiooniks, sest erinevused teineteise vahel on väikesed. Näiteks on tabelite struktuur identne, aga

seisundi veerule valitud andmetüüp erinev. Samas pole ma leidnud mitte ühtegi allikat, kus oleks terviklikult esitatud kõik need põhilahendused ning neid omavahel võrreldud lähtuvalt enamlevinud nõuetest seisundite haldamist ja talletamist nõudvatele infosüsteemidele. Kahjuks näitab minu (autori) isiklik kogemus, et kuigi antud teemat üldjuhul uue süsteemi planeerimisel puudutatakse, siis tehakse seda siiski väga pealiskaudselt. Sageli ei mõelda läbi seisunditele vastavaid ärireegleid ning kuidas iga andmemuudatus andmebaasis võib mõjutada olemi seisundit. Selle tulemusel ilmnevad kahe silma vahele jäänud detailid alles liiga hilises arendusfaasis, mil muudatuste tegemine võib olla väga keeruline ja kulukas.

Valitud teema aktuaalsust põhjendab ka asjaolu, et seoses artefaktipõhise äriprotsesside modelleerimisega on jälle muutunud aktuaalseks andmekeskne lähenemine süsteemide analüüsimisele [4]. Arefaktipõhine lähenemisviis protsesside kirjeldamisele keskendub sellele kuidas olemite andmeid vastavalt nende elutsüklile protsessi käigus konkreetse toimingu või ülesande kaudu muudetakse/ajakohastatakse [5].

Töö tegemise metoodikaks on disainiteadus [6], mille tulemusena valmib tehniline tehis (disainide kataloog), milles esitatud tehniliste lahenduste headust hinnatakse praktiliste katsetustega. Täpsemalt on töö eesmärgiks väljaselgitada võimalikud diainid infosüsteemi põhiobjektide seisundite registreerimiseks SQL-andmebaasides, panna need süsteemselt ühtset struktuuri kasutades kirja ning uurida nende headust praktiliste mõõtmist sisaldavate eksperimentidega. Antud magistritöö tulem hõlbustab infosüsteemi arendajatel disainivalikute tegemist ja selle põhjendamist.

Töö esimeses peatükis annan lühiülevaate töö teoreetilisest taustast. Toon välja peamised tööga seotud mõisted ja kirjeldan töö paremaks mõistmiseks nende omavahelisi seoseid. Järgnevas peatükis annan ülevaate varem tehtud selleteemalistest töödest. Peale mida esitan disainide kataloogis mustrite formaadis kõik leitud disainid põhiolemite seisundite esitamiseks SQL-andmebaasis. Edasi kirjeldan eksperimenti, mille viisin läbi disainide omavaheliseks võrdluseks. Seejärel esitan eksperimendi tulemused ja järeldused, mille põhjal täiendan ka kataloogis esitatud disainide kirjeldust.

2 Teoreetiline taust

Selles peatükis kirjeldan mõisteid, mida on vaja teada magistritööd lugedes.

Infosüsteem on omavahel sidestatud ja mitmel tasemel eksisteerivate andmete, funktsioonide, protsesside, sündmuste, asukohtade ja väärtuste süsteem [7]. Infotehnoloogiliste vahendite abil peetavas infosüsteemides kasutatakse andmete haldamiseks ja talletamiseks infotehnoloogiliste vahendite (näiteks andmebaasisüsteemide abil) loodud andmebaase. Sellest tulenevalt sisaldab infosüsteemi analüüsimine ja realiseerimine ka andmebaasi projekteerimist ning valmisehitamist. Juba analüüsi käigus on oluline mõista, millised on uuritavale organisatsioonile olulised põhiolemitüübid, mille kohta hakatakse andmebaasis andmeid hoidma. [1] Olem on reaalse (võimalik, et infosüsteemi poolt juhitava) maailma abstraktne või füüsiline asi e komponent [8]. Konkreetse kliendi konkreetne tellimus on näide abstraktsest olemist. Andmebaasi analüüsimise käigus on vaja leida nende asjade üldistused e olemitüübid (näiteks *Tellimus*). Kui tüüp on nime omav lõplik väärtuste hulk [9], siis olemitüüp on nime omav lõplik olemite hulk. Andmebaasi analüüsimisel pakuvad huvi sellised olemitüübid, millesse kuuluvate olemite kohta on vaja koguda andmeid (näiteks e-kaubanduse ettevõtte andmebaasis on kindlasti vaja hoida andmeid tellimuste kohta). Nendest olemitüüpidest omakorda mõned on põhiolemitüübid. Põhiolemitüüpe iseloomustab see et nendesse kuuluvatel olemitel on infosüsteemi mõttes huvipakkuv elutsükkel (näiteks tellimus võib olla ootel, kinnitatud, tasutud jne). Põhiolemitüübid võivad olla aluseks süsteemi jaotamisel alamosadeks e allsüsteemideks. Mõnikord kutsutakse põhiolemitüüpe ka põhiobjektideks.

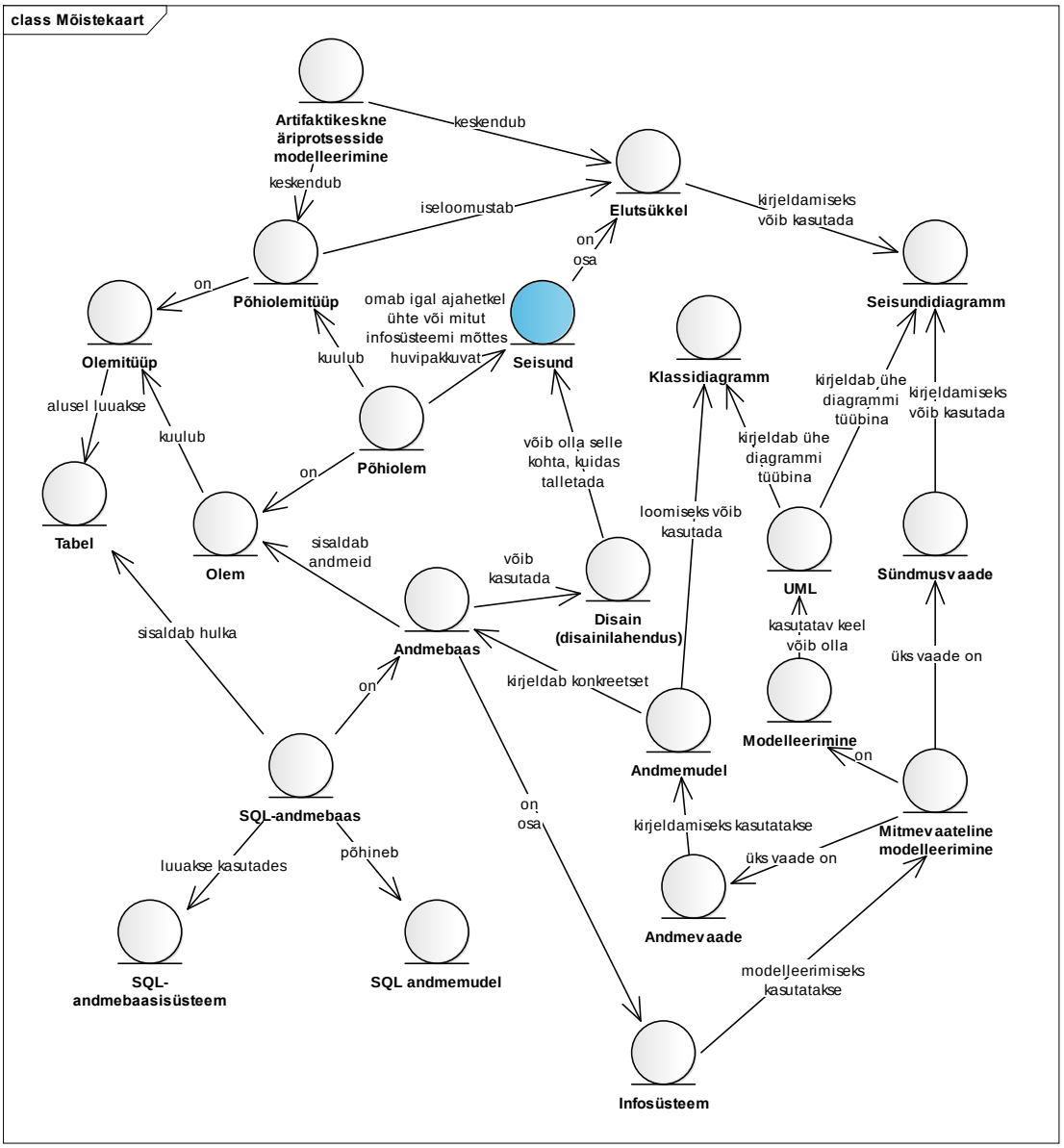
Süsteemiarenduse käigus modelleeritakse (kirjeldatakse lihtsustatult) süsteemi mitmevaateliselt [1]. Üheks modelleeritavaks vaateks on sündmusvaade. Selle esitamiseks kasutatakse tihtipeale seisundidiagramme (olekumasinaid). Nende abil kirjeldatakse põhiolemite (põhiolemitüüpi kuuluvate olemite) elutsükleid. Täpsemalt modelleeritakse iga seisundidiagrammiga ühe olemitüübi kohta selle infosüsteemi jaoks huvipakkuvaid seisundeid ning sündmuseid, mis olemeid ühest seisundist teise viivad. Selline seisundidiagramm kujutab ühte tüüpi olemite kõikvõimalikke elutsükleid. Iga seisund määrab, milline on olemite reaktsioon teatud sündmustele. [10] Eesti keele seletav sõnaraamat [8] defineerib seisundit kui olemi olekut teatud ajahetkel, mis võib

olla tingitud erinevatest teguritest. Infosüsteemi kontekstis nimetab Silverston seda oma teoses [11], mis esitab nende autorite hinnangul häid andmemudeleid, andmete olekuna.

Infosüsteemide mitmevaatelisel modelleerimisel olemitüüpidele (andmetele), olemite seisunditele ja nende elutsüklitele keskendumine ei ole uus idee [12] ning sellel on palju avaldumisvorme. Ka artefaktipõhine (tehisepõhine) lähenemisviis äriprotsesside kirjeldamisele [5] keskendub sellele kuidas olemite andmeid vastavalt nende elutsüklile protsessi käigus konkreetse toimingu või ülesande kaudu muudetakse/ajakohastatakse. Modelleerimiseks laialdaselt kasutatava UML keel kirjeldab ühe diagrammitüübina seisundidiagrammi ja pakub seega ühtlustatud esitusviisi seisundidiagrammide koostamiseks [13].

Tänu SQLi [14] populaarsusele minevikus ja jätkuvale populaarsusele tänapäeval on paljudes infosüsteemides andmete hoidmiseks kasutusel SQL-andmebaasid. Nende loomiseks kasutatakse SQL-andmebaasisüsteeme e andmebaaside loomiseks, haldamiseks ning ligipääsu võimaldamiseks mõeldud tarkvara, kus saab kasutada SQL andmebaasikeelt ning milles andmebaasi üleshitamiseks on kasutusel SQL aluseks olev andmemudel. Andmemudeli all pean siinkohal silmas andmebaasikeele abstraktset kirjeldust, mis määrab ära seal andmete esitamiseks kasutatavad andmestruktuuride tüübid, nende põhjal tehtavate operatsioonide tüübid, kasutatavad kitsenduste tüübid ning andmetüübid. [15] Sõnaga andmemudel viidatakse ka teisele mõistele – konkreetse andmebaasi kirjeldusele. Selline kirjeldus on süsteemi mitmevaatelisel modelleerimisel osaks andmevaatest. Sellise mudeli koostamiseks saab UMLis kasutada klassidiagrammi. Kasutan seda võimalust ka ise disainide kataloogi näidetes.

Joonis 1 on mõistekaart, mis nimetab töö paremaks mõistmiseks vajalike üldiseid mõisteid ja näitab nende vahelisi seoseid.



Joonis 1. Töö valdkonda kirjeldav mõistekaart.

3 Seotud uuringud

Selles peatükis annan lühiülevaate varem põhiolemite seisundite SQL-andmebaasides esitamise kohta tehtud töödest ja selleks pakutud disainidest. Vastava materjali otsimiseks kasutasin Google ja Google Scholari otsingumootoreid kui ka mitmeid infotehnoloogia valdkonna andmebaase nagu ACM Digital Library, Safari Books Online ja Cambridge Core. *Lisa 1* sidaldab kõiki märksõnu, mida kasutasin informatsiooni leidmiseks.

Otsingu tulemusena ei leidnud ma ühtegi allikat, kus oleks kõiki erinevaid võimalike andmebaasi disaine üheskoos struktureeritud viisil kirjeldatud. Samuti ei leidnud ma teaduskirjanduse hulgast palju neid teoseid, kus oleks esitatud korraga rohkem kui üks võimalik disain. Seetõttu toon antud peatükis välja ka mõningad allikad, mis selgitavad ainult ühte võimaliku lahendust, kuid mida pean oluliseks antud töös välja tuua. Lisaks teaduskirjandusele toon varasemate uuringute all välja ka lühiülevaate arendajate foorumite postitustest leitud informatsioonist. Enamikele nendele allikatele viitan ka hiljem mustrite kataloogis kui näidetele ühe või teise disaini kasutamise kohta.

Silverstoni raamatuseeria erinevad osad [16] [17] [11] kirjeldavad nende autorite hinnangul häid ja laialt kasutatavaid andmemudeleid, mis sobivad erinevates valdkondades toimetavatele organisatsioonidele. Antud seeria kaks esimest raamatut [16] [17] pakuvad läbivalt seisundi talletamiseks ainult ühte lahendust, mis kasutavad kõrge tasemeni normaliseeritud tabeleid. Seisundite koodid, nimed ning võimalik et muud metaandmed (andmed andmete kohta) on eraldi tabelis (seisundiklassifikaatori tabel) ning tabelis, kus on selle klassifikaatori poolt iseloomustatavad põhiolemite andmed, on sellele klassifikaatori tabelile viitav välisvõti (veergude hulk, kuhu kuulub üks või mitu veergu). Seega ei toimu seisundite metaandmete dubleerimist põhiolemite andmetega tabeli erinevates ridades. Selle disaini puhul on tegemist seeria kolmandas osas [11] välja toodud teise taseme seisundi mustriga.

Eelnevalt kirjeldatud raamatuseeria kolmandas osas [11] on pühendatud eraldi peatükk olemite seisunditele. See on kõige lähem sellele, mida antud töös üritan saavutada, kuid ka seal ei tooda välja kõiki neid lahendusi, mida plaanin antud töös kirjeldada. Selles on välja toodud kuus erinevat mustrit seisundite andmete hoidmiseks vastavalt erinevatele ärinõuetele, millele uus või olemasoleva süsteem peab vastama:

- Esimese taseme seisundi muster (*Level 1 Status Pattern*): kirjeldab võimalust defineerida põhiolemitüübile vastavas tabelis erinevat tüüpi sündmustele (nt loomine, kinnitamine, tühistamine) vastavad temporaalsed (ajaliste andmetega) veerud (nt loomise_aeg, kinnitamise_aeg, tühistamise_aeg). Väärtus sellises veerus näitab, et põhiolemiga on (mingil ajal) vastavat tüüpi sündmus toimunud. Selle sündmuse tulemusena jõudis põhiolem uude seisundisse. Teistpidi, kui väärtust sellises veerus ei ole (seal on NULL), siis pole põhiolemiga vastavat tüüpi sündmust toimunud ja see ei ole seega vastavas seisundis. Kui ei ole oluline teada sündmuse toimumise aega, vaid ainult selle toimumise/mittetoimumise fakti, siis võivad veerud olla tõeväärtustüüpi (nt on_loodud, on_kinnitatud, on_tühistatud).

- Teise taseme seisundi muster, hetkeseisund (*Level 2 Status Pattern, current status*): kirjeldab võimalused, kuidas iga põhiolemi korral saab registreerida selle täpselt ühe hetkeseisundi. Selleks tuleb luua eraldi klassifikaatori tabel, milles hoitakse võimalike kokkulepitud seisundite identifikaatoreid (koodid arvutisüsteemi jaoks ja nimed inimeste jaoks) ning võimalik et ka muid metaandmeid seisundite kohta. Põhiolemitüübile vastavas tabelis on välisvõti, mis viitab klassifikaatorite tabelile. Kui kõigi erinevat tüüpi põhiolemite võimalike hetkeseisundite hulk on täpselt ühesugune (näiteks aktiivne ja mitteaktiivne), siis piisab terve andmebaasi peale ühest seisundiklassifikaatori tabelist. Kui erinevat tüüpi põhiolemite seisundid võivad olla erinevad, siis tuleb luua iga põhiolemitüübi kohta eraldi seisundiklassifikaatori tabel. Teine lahendus on kindlasti paindlikum. Seda lahendust kasutatakse raamatuseerias läbivalt.
- Kolmanda taseme seisundi muster (*Level 3 Status Pattern*): lisaks eelmisele tasemele lubab antud muster omada ühel olemil korraga rohkem kui ühte seisundit. Kontseptuaalselt on põhiolemitüübi ja seisundiklassifikaatori vahel mitu-mitmele seos. Andmete registreerimiseks kasutatakse vahetabelit, mis ühtlasi talletab andmeid seisundite ajaloo kohta ning säilitab seisundimuudatuste olulistele omadustele, nagu näiteks toimumise aeg, vastavaid andmeid.
- Neljanda taseme seisundi muster (*Level 4 Status Pattern*): sarnane kolmanda taseme mustriga, kuid erinevus seisneb selles, et erinevate põhiolemitüüpide

seisundimuudatuste ajalugu on kõik koos ühes tabelis, samas kui kolmanda taseme korral oleks need andmed iga põhiolemitüübi kohta eraldi tabelis. See lahendus on võimalik, kui kõigi erinevat tüüpi põhiolemite võimalike hetkeseisundite hulk on täpselt ühesugune (näiteks aktiivne ja mitteaktiivne) või juhul kui erinevate põhiolemitüüpide kohta on defineeritud erinev hulk seisundeid, siis on nende kohta kasutusel ühine seisundiklassifikaatorite tabel, millele saab seisundite ajaloo tabelis viidata.

- Seisundite kategooria muster (*Status Category Pattern*): kirjeldab mustrit, mida on võimalik kasutada tasemete 2–4 korral. See lisab võimaluse defineerida erinevaid seisundite kategooriaid ja nende tüüpe.
- Seisunditüüpide seosed (*Status Type with Multi Rollup and Rules Pattern*): kirjeldab mustrit, mida on võimalik kasutada tasemete 2–4 korral. Lisab võimaluse kirjeldada seisundiklassifikaatorite väärtuste vahelisi seoseid ja nendele seostele kehtivaid reegleid (näiteks kui olem võib olla korraga mitmes seisundis ja on parajasti seisundis x, siis ei tohi see samal ajal olla seisundis y).

Kui võrrelda raamatus esitatud mustrite kataloogi käesolevas töös leiduva kataloogiga, siis peamine erinevus seisneb disainide arvus, struktuuris ja eksperimendis. Minu magistritöös esitan struktureeritud kujul kuus võimaliku disaini, kusjuures iga disaini juures analüüsitakse selle sobivust erinenevate nõuetega (Tabel 1). Minu esitatavate disainide hulgas on selliseid, mida [11] ei esita. Lisaks sellele viin disainide üleselt läbi eksperimendi (vt peatükk 5), mille tulemused aitavad otsustada, millistes olukordades tuleks eelistada üht disaini teisele.

David C. Hay raamat [18], mis sarnaselt Silverstoni raamatutele [16] [17] [11], kirjeldab andmete modelleerimise mustreid, kasutab läbivalt samasugust lahendust kui Silverstoni raamatuseeria esimesed kaks raamatut (teise taseme seisundi muster).

Karwin'i raamat [19] annab ülevaate halbadest SQL-andmebaasi disainidest koos seletustega kuidas neid ära tunda ning probleeme paremini lahendada. Selle lõputöö kontekstis on olulised nii 8-ndas kui 11-ndas peatükis esitatud probleemid ning lahendused. Kuigi Karwini raamat [5] ei esita nende probleemide näidetena seisundite haldamist, on seisundite haldamise mõned võimalikud disainid nende probleemide näited. Peatüki 8 (*Multicolumn Attributes*) kohaselt oleks halvaks lahenduseks kui

põhiolemitüübi tabelis luuakse iga seisundi jaoks eraldi (näiteks tõeväärtustüüpi) veerg (vt jaotised 4.4 ja 4.5). Probleemina näeb Karwin, et see võib raskendada päringuid, andmetele kehtivate reeglite jõustamist ja võimalike seisundite hulga muutmist. Lahenduseks pakub ta eraldi tabeli kasutuselevõttu, mis sisuliselt tähendaks Silverstoni teise taseme seisundi mustri kasutamist. Peatüki 11 (*31 Flavours*) kohaselt oleks halvaks lahenduseks seisundi klassifikaatori asemel kirjeldada ja jõustada võimalike seisundite hulka CHECK kitsenduse või loendustüübi abil (vt jaotis 4.3).

Dokument [20] on mõeldud kasutamiseks juhendina tulemaks toime vajadusega registreerida infosüsteemis põhiolemite seisund. See sisaldab endas viite mustrit:

- põhiolemite hetkeseisundi registeerimise vajadus,
- põhiolemite hetkeseisundi registeerimine SQL-andmebaasis,
- põhiolemite seisundimuudatuste ajaloo registreerimine,
- põhiolemite seisundimuudatuste võimaldamine graafilises kasutajaliideses,
- põhiolemite teatavate seisundimuudatuste keelamine SQL-andmebaasis.

Muster nimega „Põhiolemite hetkeseisundi registeerimise vajadus“ on kõigi ülejäänud mustrite rakendamise eelduseks, sest see kirjeldab nõuded süsteemile, millest tulenevalt on vaja leida tehniline lahendus põhiolemite hetkeseisundite registreerimiseks. Selleks lahenduseks on juba eelnevalt välja toodud teise taseme seisundi muster [11], mis soovitab eraldi seisundiklassfikaatori tabeli loomist. See on ka üks minu pakutavatest mustritest (vt jaotis 4.3). „Põhiolemite seisundimuudatuste ajaloo registreerimise“ muster vastab kolmanda taseme seisundi mustrile [11].

Vellemaa bakalaureusetöö [21] sisuks on välja tuua mõned põhilised lahendused klassfikaatorite esitamiseks SQL-andmebaasideks ning hinnata nende sobilikkust konkreetse olukorra jaoks võrreldes neid erinevate kriteeriumite alusel. Minu töö jaoks on see oluline, sest üheks disainiks on seisundiklassifikaatori tabeli loomine (vt jaotis 4.3). Täpsemalt kirjeldatakse Vellemaa töös mustrite formaadis kolm erinevat disaini:

- iga klassifikaator eraldi tabelis,
- kõik klassifikaatorid ühises tabelis,
- kunstlik ühendaja.

Seejärel võrreldakse pakutud lahendusi päringute ja kitsenduste jõustamise keerukuse, skeemi arusaadavuse ning andmete terviklikkuse ja paindlikkuse alusel. Autor järeldab tehtud uuringu põhjal, et parimat disaini kirjeldab muster „Kunstlik ühendaja“, sest selle

kasutamise tulemus on arusaadav, paindlik ning sellega on lihtne ümber käia. Lahendust, kus iga klassifikaatori kohta on ainult eraldi tabel, ei valitud parimaks väiksema paindlikuse tõttu, kuigi muudes aspektides oli see eelmisega sarnane. „Kõik klassifikaatorid ühises tabelis“ mustri osas leidis autor, et see oli vaadeldavatest kõige ebapraktilisem lahendus, sest see on oma olemuselt keerukas ning päringute kirjutamine on ebamugav. Lisaks nõuab see andmekontrollide puhul imperatiivset lähenemist (trigerite loomist) deklaratiivse lähenemise (andmebaasis deklareeritud kitsendused) asemel.

Hoberman kirjutab oma raamatus [22] SQL-andmebaasi tabelite täiendavast normaliseemisest (üle esimese normaalkuju; seda nimetatakse ka lihtsalt „normaliseerimiseks“) ja denormaliseerimisest. Normaliseerimise eesmärgiks on vähendada andmete liiasust ja selle tulemusena vältida andmete muutmise anomaaliaid ning ka teha andmete struktuur kasutajatele arusaadavamaks. SQL-andmebaasi denormaliseerimise all peetakse silmas vastupidist protsessi, mille käigus vähendatakse ühe või mitme andmebaasi tabeli normaliseerituse astet läbi tabelite ühendamise või veergude dubleerimise. Oma raamatus kirjeldab Hoberman denormaliseerimise algoritmi, mis aitab otsustada, milliste tabelite astmeid tuleks vähendada. Antud algoritmi kasutasin ma oma bakalaureusetöös [23] võrdlemaks kõrge tasemeni normaliseeritud (kõik tabelid viiendal normaalkujul) ja denormaliseeritud tabelite disaini. Minu töö tõi välja, et algoritm soovitab denormaliseerida põhiolemitüüpidele vastavaid tabeleid, lisades nendesse seisundiklassifikaatorite tabelites olevad andmed (näiteks nimetused). Seisundiklassifikaatorite tabelid säilivad, st nüüd on seisundite kohta käivad metaandmed dubleeritud erinevates tabelites. Mõlemas loodud andmebaasis tegin järgnevad testid: mõõtsin andmemahte, päringute täitmise ja konkreetse olemi andmete muudatuse kiirust ning keerukust ja ka kitsenduste jõustamise keerukust. Ootuspäraselt selgus, et kõik koondpäringud, kaasa arvatud need, mis hõlmasid erinevate olemite seisundite leidmist, olid denormaliseeritud andmebaasis kiiremad ja vähem keerukad. Samas kitsenduste jõustamine ja andmemuudatuste töökiirus oli kõrge tasemeni normaliseeritud tabelitega andmebaasis parem. Järeldasin testide tulemustest, et kuigi tabelite denormaliseerimise juures leidub positiivseid külgi, on andmebaasi tabelite kõrge tasemeni normaliseerimine operatiivandmetega andmebaasides siiski otstarbekam kui denormaliseerimine. Minu töö kontekstis on see tulemus jällegi oluline, sest üheks pakutavaks disainiks on kasutada

seisundiklassifikaatorite tabeleid (vt jaotis 4.3). Minu eelnev töö soovitab sellega seoses denormaliseerimist vältida.

Rõzova magistritöö [24] annab ülevaate disainimustritest, mida võiks kasutada üldistusseoste (*generalization*) realiseerimiseks SQL-andmebaasides. Sellest tulenevalt kirjeldab töö autor mustrite kataloogi, mis sisaldab 14 erinevat disaini. Lõpuks võrdleb autor disaine näiteks kitsenduste jõustamise lihtsuse, andmebaasi struktuuri muutmise lihtsuse ja päringute jõudluse alusel. Töö tulemiks on võrdlustabel, mis aitab nendest mustritest hõlpsamini aru saada ja võimaldab valida lahenduse vastavalt kriteeriumitele, mida tarkvara peab täitma. Antud töös kirjeldatud disainid on olulised, sest kontseptuaalses andmemudelis võib põhiolemitüübile modelleerida alamtüübid, millest igaüks vastab mingile seisundile, milles põhiolemid võivad viibida (vt jaotis 4.7).

Ossipova bakalaureusetöö [25] sisuks on analüüsida mitme väite ühe andmeväärtusena esitamise eeliseid ja puuduseid. Selleks, et viia läbi eksperiment ja leida vastused töös esitatud uurimisküsimustele, realiseeris autor viis erinevat andmebaasi disaini:

- traditsiooniline,
- vektorkodeerimisega,
- kahemõõtmelise massiivi tüüpi veeruga,
- JSON tüüpi veeruga,
- JSONB tüüpi veeruga.

Kõikide realiseeritud andmebaaside puhul viis ta läbi testid mõõtmaks päringute ja andmemuudatuste kiirust ja keerukust. Töö tulemustena selgus, et vaadeldavas kontekstis osutus kõige sobivamaks siiski traditisioonilise disainiga andmebaas. Samas järeldas töö, et ühe andmeväärtusena mitme väite esitamine on mugav ja rakendatav infosüsteemides, kus ei toimu pidevat andmete uuendamist ja kus olulisemaks teguriks on päringute kiirus. Minu töös on see oluline, sest põhiolemite hetkeseisundeid võib esitada ka vektorkoodi kasutades (vt jaotis 4.6).

Artikkel [26] annab ülevaate mustritest, mis aitavad objektorienteeritud tarkavras seisundimuudatustega toime tulla.Täpsemalt räägib see süsteemi disainerite ees seisvatest väljakutsetest olekumasinate (*state machine*) realiseerimisel ning nende väljakutsete soovitatavatest lahendustest. Töös esitatakse 13 erinevat mustrit, millest üheksa kirjeldatakse eraldi alampeatükkides mustrite formaadis ning neli on tuletatud lisana

mõnest neist pakutud lahendustest. Disainideks on näiteks: *State Object, Basic FSM, State-Driven Transitions, Owner-Driven Transitions, Layered Organization*. Kuigi antud magistritöös ei keskenduta tarkvaras olekumasinate realiseerimisele on siiski hea teada, kuidas tulla tarkvaras toime vajadusega seisundimuudatusi läbi viia.

Otsides materjale antud teemal tehtud varasemate uuringute kohta jõudsin mitmel korral erinevate arendajate foorumite sissekanneteni. Enamus neist viitasid jällegi disainile (klassifikaatori tabel), mida olen siin peatükis juba mitmeid kordi nimetanud [27] [28] (vt jaotis 4.3) Lisaks võib leida foorumitest postitusi kus arutletakse, kui optimaalne on hoida seisundit esitavaid väärtuseid põhiolemitüübile vastavas tabelis ilma klassifikaatori tabelita ning kui seda tehakse, siis millist andmetüüpi vastavate veergude puhul kasutada. Täpsemalt pakuti kasutada tabelis tõeväärtustüüpi väärtuseid [29] (vt jaotis 4.5), temporaalseid väärtuseid [30] (vt jaotis 4.4), vektorkoode [29] (vt jaotis 4.6) kui ka lihtsaid sõnalisi väärtuseid [27].

Selles peatükis tõin välja allikad, mille leidsin kasutades erinevaid otsingumootoreid ja avatud andmebaase. Ülevaatest selgub, et leidub küllaltki palju materjale, mis puudutab põhiolemite seisundite kohta andmebaasis andmete hoidmist. Siiski ei leidnud ma otsingutulemustena ühtegi allikat, mis koondaks neid kokku hästi struktureeritud ühtseks materjaliks ning viiks leitud disainidega läbi võrdlevaid eksperimente. Sellisest materjalist oleks arendajatel kindlasti abi leidmaks süsteemi nõuetest tulenevalt sobivaima lahenduse planeeritavale andmebaasile.

4 Erinevad disainid seisundite esitamiseks SQL-andmebaasides

Selles peatükis annan ülevaate erinevatest disainidest mida on võimalik kasutada põhiolemite seisundite talletamiseks SQL-andmebaasides. Kokku esitan kuus disaini. Selleks, et disainid oleksid paremini võrreldavad ja nende kirjeldused lihtsamalt loetavad, esitan need kõik mustri formaadis. Muster on struktureeritud kirjeldus, mille abil on hea väljendada korduvaid probleeme ja nende lahendusi. Mustrite nimed moodustavad valdkonna sõnavara ja seega on oluline, et need oleksid lühikesed ja tabavad. Mustri näol on tegemist malliga, mis selgitab probleemi ja lahendust viisil, et seda saaks kasutada paljudel erinevatel juhtudel. Mustri kirjelduse osaks on näited, kus kirjeldatakse konkreetsete probleemide lahendusi (nt kirjeldatakse, kuidas esitada seisundeid tabelis *Tellimus*).

Idee esitada probleemide ja nende lahenduste kirjeldused mustritena pärineb arhitekt Christopher Alexander'ilt, kes kasutas neid ehitusvaldkonnas hoonete projekteerimise parimate praktikate väljendamiseks [31]. Mustrid on leidnud palju kasutust ka tarkvarateaduses, kus on näiteks populaarsed disainimustrid [32], mis kirjeldavad sagedasi tehnilisi probleeme süsteemide ülesehituses (struktuuris) ja käitumises ning nende probleemide lahendusi. Käesolevas töös viidatakse palju andmete modelleerimise mustritele või universaalsetele andmemudelitele. Viimased nendest on mustrite sarnased selles mõttes, et need väljendavad häid lahendusi sagedastele probleemidele kuid nende tekstiline kirjeldus ei ole nii liigendatud kui mustrite puhul. Andmebaasi disainimustrite kogumikud, millele töös viidatakse, on näiteks [19] [21] [24].

Ma ei nimeta käesolevas kataloogis esitatud lahendusi mustriteks, vaid disainideks. Ma kasutan disainide kirjeldamiseks mustrite formaati. Muster tähendab korduvalt kasutust leidnud head lahendust. Selles aga, milline on hea disain, ei saa ma olla lõplikult kindel enne, kui olen läbi viinud eksperimente (vt peatükk 5).

Üheks mustri tunnuseks on korduvus, st et probleem on sagedane/oluline ja pakutavat lahendust on päriselus vähemalt kolm korda edukalt kasutatud [33]. Antud printsiibist on lähtutud ka sellesse magistritöösse võimalikke disainide kandidaate otsides. Seetõttu koosneb kataloog vaid põhilahendustest olemite seisundite esitamiseks. Nimetaksin neid

makrolahendusteks, sest kõigil neil leidub palju variatsioone (näiteks erinevaid lahendusi omavahel kombineerides), mida plaanin lühidalt nimetada, kuid ei hakka kõiki detailselt välja tooma.

Nagu eelnevalt sai öeldud, siis on selles magistritöös kasutatud erinevate disainide esitamiseks ühtset struktuuri võttes aluseks mustrite kirjeldamisel kasutatava struktuuri. Täpsemalt on võetud aluseks nii Eessaare õppematerjalis [20], Miku [15] ja Krönströmi [34] magistritöös välja toodud formaat kui ka Fowleri raamatus [35] olev peatükk antud teemal. Nende omavahelisel kombineerimisel koostasin järgneva struktuuri, mis minu arvates aitab kõige paremini kirjeldada lahendusi just selles lõputöös käsitletavale probleemile.

Disainide kataloogide loomine on näide algoritmilisest mõtlemisest, mille nelja nurgakivi kirjeldatakase [36].

- Suur probleem (kuidas luua süsteeme) jagatakse väiksemateks alamprobleemideks ning iga alamprobleemi kohta võib luua oma mustrite/disainide kataloogi.
- Kataloogi loomine eeldab tutvumist sellega, kuidas sama probleemi on varem teiste poolt lahendatud.
- Kataloogi loomisel tuleb osata eristada olulist informatsiooni mitteolulisest.
- Kataloogis kirjeldatud lahendused saavad olla aluseks konkreetsete probleemide lahendamisele.

4.1 Disaini kirjelduse struktuur

Nimi eesti keeles: disaini eestikeelne nimetus, mida kasutatakse läbivalt töös kui on vajalik sellele lahendusele viidata. Seda kasutatakse disainile vastava jaotise nimena.

Nimi inglise keeles: disaini ingliskeelne nimetus, mida enamasti kasutatakse vastava disaini lähteallikas. Sellest tuleneb ka lahenduse eestikeelne nimetus.

Alternatiivsed ingliskeelsed nimed: juhul kui disain on tuntud mitme erineva nimetuse all, siis tuuakse need siin välja.

Allikad: lähtematerjalid, mis on võetud aluseks disaini selles magistritöös kirjeldamisel. Esitatakse viited kasutatud materjalides toodud töödele.

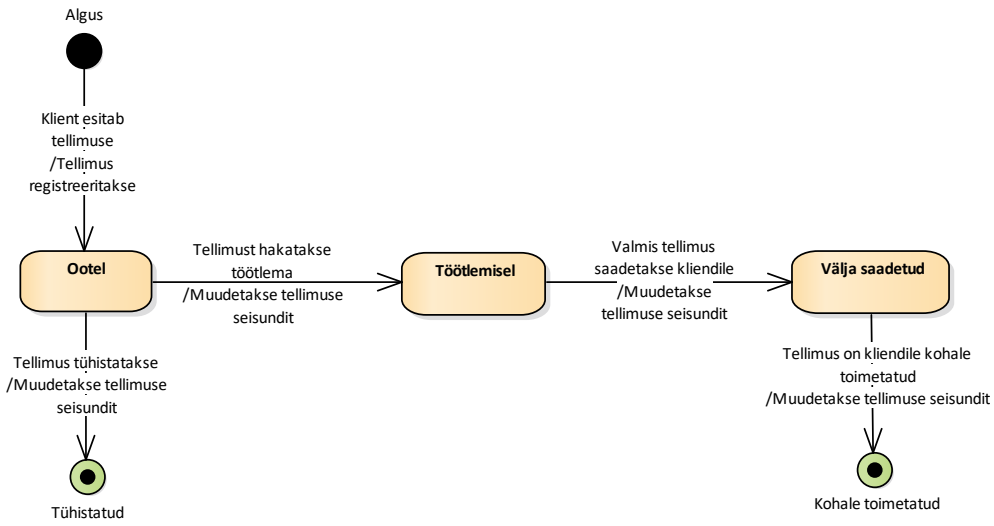
Lahendus: disaini sõnaline selgitus. Disain pakub välja baasjoone – erinevate nõuete (Tabel 1) rahuldamiseks võib olla vaja seda muuta.

Eelised: disaini kasutamise eelised, mis tulenevad nii lähteallikates antud infost kui enda teadmistest. Täiendan seda nimekirja tehtud eksperimentide tulemuste (vt peatükk 5) alusel. Juhul kui tegemist on eksperimendi tulemusel leitud eelisega, siis lisan viite peatükile 6.

Puudused: disaini kasutamise puudused/raskused/väljakutsed, mis tulenevad nii lähteallikates antud infost kui enda teadmistest. Täiendan seda nimekirja tehtud eksperimendi tulemuste (vt peatükk 5) alusel. Juhul kui tegemist on eksperimendi tulemusel leitud puudusega, siis lisan viite peatükile 6.

Variatsioonid: kui tehnilisest lahendusest leidub variatsioone, siis kommenteerin neid lühidalt ja lisan viiteid allikatele, kus nende kohta saab täpsemalt lugeda.

Näide: disaini rakendamine konkreetse probleemi lahendamiseks. Võrdluse huvides kasutan kõigis näidetes samale põhiolemitüübile (*Tellimus*) vastavat tabelit ja ühesugust seisundite hulka. Tellimuse puhul oluliseks peetavad nimelised omadused (atribuudid) on tellimuse number ja kohaletoimetamise aadress. Näite atribuutide ja seosetüüpide hulka on teadlikult piiratud, et keskenduda näites kõige olulisemale – seisundite esitamisele. Võimalikud seisundid leidsin kolme suure e-poe platvormi Shopify [37], WooCommerce [38] ja BigCommerce [39] süsteemides kasutusel olevate tellimuse seisundiklassifikaatorite väärtuste alusel. Antud näite puhul võivad tellimused olla seisundis: ootel, töötlemisel, välja saadetud, kohale toimetatud ja tühistatud. Joonis 2 esitab antud näite kohta seisundidiagrammi. Tegemist on lihtsustatud mudeliga ja reaalsete süsteemide korral võib see olla palju keerulisem. Näiteks on reaalse süsteemi loomisel vaja otsustada, mis juhtub siis kui peale tellimuse töötlemise alustamist kaob selle täitmise võime või tellimus jääb mingil põhjusel kliendile üleandmata. Läksin teadlikult mudeli lihtsustamise teed, et näidet mitte liiga keeruliseks ja seega võimalikele lugejatele halvasti jälgitavaks muuta.



Joonis 2. Tellimuse seisundidiagramm.

Näide lähtub järgmistest nõuetest (vt ka Tabel 1):

- iga tellimus on korraga täpselt ühes seisundis (nõuded 1.1 ja 1.2),
- iga tellimuse puhul on vaja teada ainult selle hetkeseisundit (nõue 2.1),
- iga tellimuse puhul on lisainfona seisundimuudatuse kohta vaja teada selle aega (nõue 3.2),
- andmebaasis ei ole vaja säilitada lisainfot seisundite kohta (nõue 5.1).

Näites esitan tabelites olevad andmed ning kasutan kõikide disainide puhul andmeid ühesuguste omadustega tellimuste kohta. Kasutan samateemalist näidet (kuid muidugi ka palju suuremate andmehulkadega) ka hilisemates eksperimentides (vt peatükk 5).

Strateegia: disaini realisatsioon konkreetses andmebaasisüsteemis, milleks antud töös on PostgreSQL (11). Esitan disaini eelmises punktis kirjeldatud näite kohta. Seega vastab see disaini põhilahendusele, mitte variatsioonidele. Lahenduse esitan Enterprise Architect CASE vahendis koostatud andmebaasi diagrammina, lisades vajadusel juurde tekstilisi kommentaare (kui visuaalselt ei ole võimalik kogu infot välja tuua). Näiteks toon kommentaarides välja CHECK kitsenduste loogikaavaldised, trigerite lähtekoodi ja indeksite loomise laused. Paraku ei võimalda Enterprise Architect esitada domeene. Diagrammil esitatakse kitsendused nii nagu need oleksid otse tabeli küljes, kuid

andmebaasis võtan ühesuguste omadustega veergude kirjeldamiseks kasutusele domeeni, millega saab selle kitsenduse ühekordselt siduda [40].

Nõuded: lahenduse sobivus äriprotsessidest tulenevatele nõuetele. Sobivuse hindamiseks kasutan erinevaid allikaid ning enda teadmisi.

Järgnevalt esitan nõuded, mis on jaotatud kuude erinevasse klassi. Iga kataloogis esitatava disaini kohta antakse ülevaade kuidas tuleks läheneda antud disainile, milliseid muudatusi teha selleks, et olla vastavuses Tabel 1 kirjeldatud nõuetega. Nõuetele sobivuse kirjelduse paremaks illustreerimiseks kasutan seal tellimuse näidet.

Tabel 1. Nõuete kirjeldused.

Id	Nõude klass	Id	Täpsem nõue
1	Olemi samaaegsete seisundite hulk	1.1	Olem peab olema kindlasti mingis seisundis
		1.2	Olem saab olla korraga maksimaalselt ühes seisundis
		1.3	Olem saab olla korraga mitmes seisundis
2	Seisundimuudatuste ajalugu	2.1	Olemi puhul tuleb teada ainult selle hetkeseisundit
		2.2	Olemi puhul tuleb teada selle seisundite ajalugu
3	Lisainfo vajadus seisundimuudatuste kohta	3.1	Olemi puhul huvitab ainult seisundi omamise fakt
		3.2	Olemi puhul on vaja lisainfot seisundimuudatuse kohta (millal see seisund omandati, kes selle muudatuse algatas jne)
4	Seisundimuudatuste lubatavuse kontrollimine *	4.1	Ei ole vaja olekutabelit, st lubatavad seisundimuudatused tuleb sisse kodeerida kontrollprogrammi ja neid ei ole võimalik dünaamiliselt andmebaasi kasutajate poolt muuta. Seega kui lisandub või kaob lubatud üleminek kahe seisundi vahel, siis tuleb muuta programmi.
		4.2	On vaja olekutabelit, mis kirjeldab kõikvõimalikud lubatud seisundimuudatused ja mis võimaldab teha lubatavate üleminekute hulgas jooksvalt muutusi ilma selleks kontrollprogrammi ümberkirjutamata.
5	Lisainfo seisundite kohta	5.1	Andmebaasis ei ole vaja säilitada lisainfot seisundite kohta
		5.2	Andmebaasis on vaja säilitada lisainfot seisundite kohta (näiteks nimetus erinevates keeltes, kehtivus, kommentaar)
6	Seisundite hulga muutus ajas	6.1	Tekib uus võimalik seisund, milles olem võib viidida
		6.2	Kaob seisund, milles olem võib viibida

* Seisundimuudatuste lubatavuse reeglid on leitavad põhiolemitüüpide seisundidiagrammidelt. Näiteks olemitüübi *Tellimus* seisundidiagramm kirjeldab kõikvõimalikud tellimuste seisundid ja lubatud seisundi üleminekud (Joonis 2). Kui diagrammil puudub üleminek seisundist *Ootel* seisundisse *Välja saadetud*, siis see väljendab, et see üleminek pole lubatud. Selliste reeglite andmebaasi tasemel jõustamiseks ei saa tänapäeva SQL-andmebaasisüsteemides kasutada deklaratiivseid kitsendusi. Selle asemel tuleb luua kas andmebaasi trigerid või teostada kontroll andmebaasiserveris talletatud rutiinides, mille kaudu registreeritakse andmebaasis seisundimuudatuste tulemused. Võimalus on seda kontrolli teha ka rakenduses või siis ka üldse mitte kontrollida. Pean kontrollimist siiski soovitavaks, sest nagu ka arstiteaduses kehtib ka siin põhimõte – probleeme ennetada on kokkuvõttes odavam ja lihtsam kui neid tagantjärgi lahendada.

Kitsenduste jõustamisest kirjutades lähtun edaspidi järgnevast eelistuse järjekorrast:

1. Deklaratiivne andmebaasi kitsendus (SQL-andmebaasis PRIMARY KEY, UNIQUE, NOT NULL, FOREIGN KEY, CHECK kitsendused. Deklaratiivne tähendab, et süsteemile öeldakse, mida on vaja kontrollida, mitte kuidas e milliste käskude järjestusega seda teha).

2. Andmebaasi triger (imperatiivne viis kitsenduse jõustamiseks, mis siiski tagab, et kitsendusest ei saa mööda minna. Imperatiivne tähendab, et trigeris kirjeldatakse samm-sammult ära, kuidas kontrolli teha.).

3. Jõustamine andmebaasiserveris talletatud rutiinis, mis on mõeldud andmebaasis andmete muutmiseks.

4. Jõustamine rakenduses.

5. Mittejõustamine.

Eeldan, et kui kitsendust saab jõustada tasemel n, siis saab seda teha ka tasemel n+1. Toon välja eelistuste mõttes esimese (võimalikult väike n) viisi, mida kasutades saab kitsendust jõustada.

4.2 Korduvad osad

Mustri kirjelduse osaks on alati ka probleemipüstitus ning disaini valikut mõjutavad jõud. Kuna need korduvad kõigis disainides, siis tõstan need „sulgude ette“, et mitte asjatult teksti dubleerida.

Probleem: Andmebaasis on vaja talletada andmeid põhiolemite seisundite kohta.

Jõud: Iga põhiolemi korral peab olema andmebaasist päringuga leida selle hetkeseisund.

Andmebaasi peab olema lihtne muuta nii, et

- lubada ühel olemil korraga mitu seisundit,
- hakata säilitama seisundimuudatuste ajalugu, sh erinevat lisainfot seisundimuudatuste kohta,
- hakata säilitama erinevat lisainfot hetkeseisundisse toonud seisundimuudatuse kohta,
- jõustada olekutabeli abil ülemineku kitsendused, mis piiravad andmebaasi tasemel lubatud seisundite üleminekuid,
- säilitada lisainfot seisundite kohta,
- tulla toime seisundite hulga muutusega ajas.

Andmebaasis ei tohi olla kontrollimatut andmete liiasust, sest selle tulemusena võidakse andmebaasis registreerida vastuolulised väited. Siiski andmebaasis võib olla kontrollitud andmete liiasus. Kontrollitud andmete liiasuse puhul on küll andmebaasist võimalik tuletada üks ja sama väide rohkem kui ühel viisil, kuid andmebaasi kasutaja on liiasusest teadlik ja andmebaasisüsteem hoolitseb andmete muutmisel automaatselt, et väidetesse ei tekiks vastuolu.

4.3 Seisundiklassifikaatori tabel

Nimi inglise keeles: (Separate) Table for status classifications

Alternatiivsed ingliskeelsed nimed: *Look-up table, Dependent table, Reference table, Domain table*

Allikad:

- „SQL Antipatterns: Avoiding the Pitfalls of Database Programming“ [19]
- Raamatuseeria „The Data Model Resource Book“ [16] [17] [11]
- „Mõned disainimustrid klassfikaatorite esitamiseks SQL andmebaasides“ [21]
- „Enterprise Model Patterns: Describing the World (UML version)“ [18]
- „Mustrid“ [20]

Lahendus: Igale põhiolemitüübile vastava tabeli kohta luuakse omaette seisundiklassifikaatori tabel, kus on registreeritud kõikvõimalikud seisundid, milles vastavat tüüpi põhiolemid võivad olla. Tabelis, kus on selle klassifikaatori poolt iseloomustatavate põhiolemite andmed, on klassifikaatori tabelile viitav välisvõti (välisvõtme veerg/veerud + kitsendus). Välisvõtme veerg/veerud peavad olema kohustuslikud (ei tohi lubada NULLe). Tagamaks, et klassifikaatori koodi muutmisel seisundiklassifikaatori tabelis kantakse muudatus üle põhiolemitüübile vastavasse tabelisse, kasutatakse välisvõtme kitsendusega seotud kompenseeriva tegevuse määrangut ON UPDATE CASCADE. Sama välisvõtme määrang ON DELETE NO ACTION tagab, et juhul kui üritatakse kustutada seisundiklassifikaatori väärtust (rida seisundiklassifikaatori tabelist), siis see ei õnnestu, kui sellega on seotud vähemalt ühe põhiolemi andmed (rida põhiolemitüübile vastavas tabelis). Selleks, et määrata esimene seisund, millesse iga põhiolem loomise järel vaikimisi läheb, tuleb defineerida vaikimisi väärtus eelnimetatud välisvõtme veergudele. Vaikimisi väärtuseks on selle seisundi kood, millesse põhiolem loomise järel vaikimisi läheb.

Eelised:

- Välditakse olukorda, kus põhiolemitüübile vastavas tabelis on seisundiklassifikaatori välisvõtme veeru asemel veerg, milles lubatud väärtuste hulk on piiratud CHECK kitsendustega või loendustüüpi kuuluvate väärtustega (nt tabelis *Tellimus* on veerg *tellimuse_hetkeseisund*, milles lubatud tekstilised väärtused (nt „ootel“, „töötlemisel“) on määratud sellele veerule loodud CHECK kitsendustega). Sellise lahenduse probleeme kirjeldatakse SQL antimustris „31 maitset“ [19].

- Ei toimu seisundite metaandmete (nt nimi) dubleerimist põhiolemitüübile vastava tabeli erinevates ridades, misläbi vähendatakse andmete liiasust ja andmete muutmise anomaaliaid. [23]

- Lihtne leida ja kasutajaliideses esitada võimalikke seisundite nimesid ja muid metaandmeid, sest neid andmeid tuleb küsida seisundiklassifikaatori tabelist. (vt jaotis 5.6.1)

- Erinevate põhiolemitüüpide seisundiklassifikaatorite korral võib koguda erinevaid metaandmeid – erinevates seisundiklassifikaatorite tabelites võib olla erinev veergude hulk.

- Puudub vajadus seisundite nimesid ja muid metaandmeid rakendusse sisse kodeerida, mistõttu nende andmete muutmine ei nõua programmeerija sekkumist. (vt jaotis 5.6.3)

- Uue seisundi väärtuse lisandumisel ei ole vaja luua andmebaasi uut tabelit, või lisada olemasolevasse tabelisse uut veergu, vaid saab lihtsalt lisada uue rea olemasolevasse (seisundiklassifikaatori) tabelisse [21]. See omakorda tähendab, et uue seisundi väärtuse kasutuselevõtuks ei pea pöörduma andmebaasi administraatori poole (kes käivitaks CREATE või ALTER lause) või juhul kui väärtuseid saab hallata läbi spetsiaalse rakenduse, siis ei pea rakendusele andma andmebaasi struktuuri muutmise õiguseid. Esimene oleks vähepaindlik ja teine oleks väheturvaline. Seisundi lisamise SQL kood on (PostgreSQL 11 näitel) võrreldes teiste töös esitatavate disainidega kõige lihtsam (vt peatükk 6).

- Tänu ON DELETE NO ACTION määrangule välisvõtme kitsenduses ei ole võimalik kogemata kustutada seisundiklassifikaatori tabelist väärtust, mida kasutatakse vähemalt ühe olemi seisundi iseloomustamiseks. [19] Lisaks on seisundi eemaldamiseks mõeldud SQL kood (PostgreSQL 11 näitel) võrreldes teiste töös esitatavate disainidega üks kõige lihtsamatest (vt peatükk 6).

- Arendajate foorumipostitus [29] soovitab, et kui on juba teada, et olem võib olla samaaegselt ainult ühes seisundis, siis tuleks luua põhiolemitüübile vastavas tabelis ainult üks veerg seisundiväärtuste jaoks – st disain sobib loomulikul viisil sellise kitsenduse jaoks.

- Andmebaasisüsteem salvestab tabelites kasutajatele esitatavad andmed sisemiselt lehekülgedel ehk plokkides. Juhul kui tabeli veergude arv läheb nii suureks, et üks rida ei mahu ära ühte plokki, siis võib, sõltuvalt andmebaasisüsteemist, rida jaotuda mitme ploki vahel ning andmete lugemine ja muutmine võtab rohkem aega [23]. Kuna seisundite metaandmeid ei dubleerita põhiolemitüübile vastavas tabelis, siis see peaks taolise olukorra tekkimise võimalust vähendama. Kuna hoides võimalike seisundite andmeid eraldi klassifikaatori tabelis on selle tabeli ridade arv suhteliselt väike, siis kulutatakse nende andmete salvestamisel ja lugemisel vähem andmebaasisüsteemi ressurssi [21] võrreldes olukorraga, kus need andmed oleksid põhiolemitüübile vastavas tabelis. Lisaks on (PostgreSQL 11 näitel) tegemist disainiga, mille puhul andmemahud on võrreldes kõigi teiste kataloogis olevate disainidega kõige väiksemad (vt peatükk 6).

- Päring, leidmaks mingis kindlas seisundis olemeid, tuleb teha ühe tabeli põhjal kasutades seisundi koode, st ei ole vaja pöörduda seisundiklassifikaatori tabeli poole. See omakorda suurendab (PostgreSQL 11 näitel) päringu täitmiskiirust (vt peatükk 6).

- Koondandmete otsimise päringud on (PostgreSQL 11 näitel) võrreldes teiste kataloogis esitatud disainidega vähem keerukad ja täitmine on kiire. (vt peatükk 6)

Puudused:

- Suureneb ühendamisoperatsioone nõudvate päringute hulk (päringutes võib olla vaja ühendada põhiolemitüübile vastav tabel ja seisundiklassifikaatori tabel), mis

võib vähendada päringute töökiirust ja suurendada keerukust. Võrdlusest teiste töös esitatavate disainidega selgus (PostgreSQL 11 näitel), et peamiselt mõjutab see päringut, kus peab leidma kõik olemid, mis pole ühes kindlas seisundis, ning tagastama iga sellise kohta hetkeseisundi nimetuse (vt peatükk 6).

- Klassifikaatoritele viitavate välisvõtmete väärtused ei pruugi olla lõppkasutajale arusaadavad, sest lõppkasutaja võib eelistada koodidele nimesid (näiteks 2 *vs.* „töötlemisel“) [21]. Lõppkasutajale arusaadavat seisundi nime peab alati otsima seisundiklassfikaatori tabelist. Lahenduseks on kasutada seisundite koodidena tekstilisi, lõppkasutajatele arusaadavaid väärtuseid. [27]

- Kui andmebaasis hakatakse registreerima uuele olemitüübile vastavaid andmeid, siis on vajalik uuendada andmebaasi struktuuri, lisades uue seisundiklassfikaatori tabeli ja põhiolemitüübile vastavasse tabelisse välisvõtme veeru/veerud ning kitsenduse. [21]

- Juhul kui muutuvad nõuded seisundite kohta kogutavate metaandmete kohta, siis tähendab see muudatusi kõikides seisundiklassifikaatorite tabelites (vt Tabel 2).

Variatsioonid:

- Kõikidel põhiolemitüüpidel on ühesugune hulk võimalikke seisundeid ning kasutusel on üks ühine seisundiklassifikaatori tabel [11]. Kõikides põhiolemitüüpidele vastavates tabelites on välisvõtme kaudu loodud seos selle tabeliga.
- Denormaliseerimise lahendus, kus seisundi nimetus ja võimalik et ka muud seisundite metaandmed on dubleeritud põhiolemitüübile vastavas tablis [23]. Samas säilib ka seisundiklassifikaatori tabel ning neid tabeleid siduv välisvõti.
- *Kõik klassifikaatorid ühises tabelis* disain [21], mis pakub luua ühise seisundiklassifikaatorite tabeli. Selles hoitakse igale põhiolemitüübile vajalike seisundiklassifikaatorite väärtuseid, kusjuures erinevatel põhiolemitüüpidel on erinev võimalik hulk seisundeid.
- Disaini *Kunstlik ühendaja* kohaselt [21] luuakse igale põhiolemitüübile eraldi seisundiklassifikaatori tabel, kuid see on seotud välisvõtme kaudu ühise kunstliku klassifikaatorite tabeliga. Ühises tabelis hoitakse klassifikaatorite metaandmeid (näiteks seisundi viimane muutja või muutmiskuupäev), mida kogutakse kõikide

klassifikaatorite korral. Seega kaob vajadus vastavaid veerge erinevatesse klassifikaatorite tabelitesse lisada.

Näide:

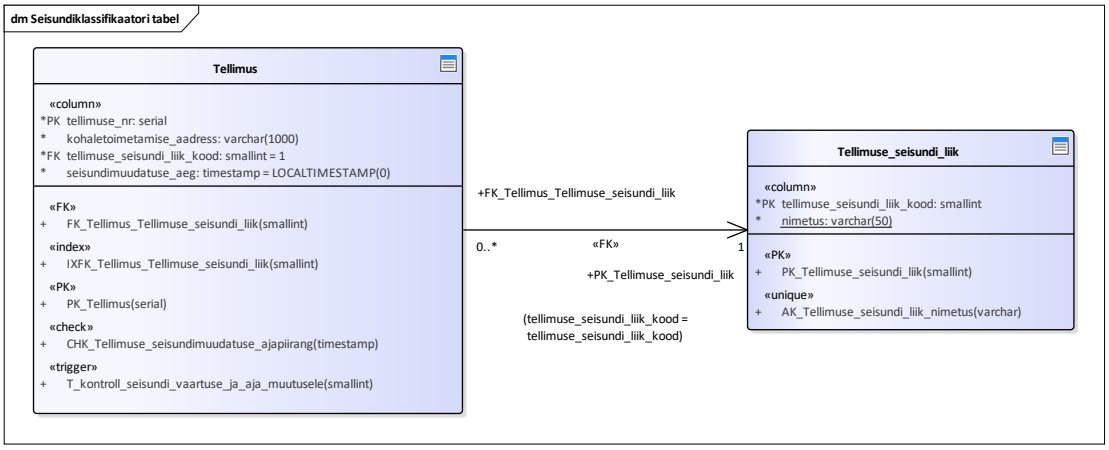
Põhiolemitüübi *Tellimus* alusel luuakse tabel *Tellimus*:

tellimuse_nr	**kohaletoimetamise_aadress**	**tellimuse_seisundi_liik_kood**	**seisundimuudatuse_aeg**
1	Narva mnt 5	1	03.03.2019 12:13:15
2	Ale 6	2	14.03.2019 12:02:11
3	Ristiku 51	3	14.03.2019 14:02:14
4	Akadeemia tee 21a	4	14.03.2019 14:55:32
5	Soo 46	5	12.03.2019 12:00:02

Luuakse seisundiklassifikaatori tabel *Tellimuse_seisundi_liik*:

tellimuse_seisundi_liik_kood	**nimetus**
1	Ootel
2	Töötlemisel
3	Välja saadetud
4	Kohale toimetatud
5	Tühistatud

Strateegia: Joonis 3 esitab *Seisundiklassifikaatori tabel* disaini realisatsiooni PostgreSQL andmebaasisüsteemis.



Joonis 3. Seisundiklassifikaatori disaini andmebaasi diagramm.

Välisvõtme veerule *tellimuse_seisundi_liik_kood* tabelis *Tellimus* loon ühedamisoperatsioonide töökiiruse parandamiseks indeksi (Joonis 4):

```
CREATE INDEX IXFK_Tellimus_Tellimuse_seisundi_liik ON Tellimus
(tellimuse_seisundi_liik_kood);
```

Joonis 4. Indeksi loomise lause.

Tagamaks jaotises 4.1 esitatud näite kirjelduses esitatud nõude (3.2) täidetuse loon trigeri (Joonis 5), mis tagab, et tellimuse seisundi väärtuse muutumisel muudetakse ka viimase seisundimuudatuse aega. Triger käivitub vaid siis, kui muudetakse hetkeseisundit nii, et uus väärtus erineb vanast. Kuna iga tellimus peab olema kohustuslikult mingis seisundis, siis pole tingimuses vaja arvestada olukorraga, et uue või vana väärtuse asemel on NULL:

```
CREATE OR REPLACE FUNCTION F_kontroll_seisundi_vaartuse_ja_aja_muutusele()
RETURNS TRIGGER AS $$
BEGIN
NEW.seisundimuudatuse_aeg = LOCALTIMESTAMP(0);
RETURN NEW;
END;$$ LANGUAGE plpgsql SECURITY DEFINER
SET search_path = public, pg_temp;

CREATE TRIGGER T_kontroll_seisundi_vaartuse_ja_aja_muutusele
BEFORE UPDATE OF tellimuse_seisundi_liik_kood ON Tellimus
FOR EACH ROW
WHEN (OLD.tellimuse_seisundi_liik_kood<>NEW.tellimuse_seisundi_liik_kood)
EXECUTE FUNCTION F_kontroll_seisundi_vaartuse_ja_aja_muutusele();
```

Joonis 5. Trigeri loomise laused.

Nõuded: Tabel 2 esitab lahendused kuidas olla vastavuses nõuete klassidest tulenevatele tingimustele (vt jaotis 4.1).

Tabel 2. Seisundiklassifikaatori tabel lahendused nõuetele.

	Nõude kirjeldus	**Lahendus**
1	Olem peab olema kindlasti mingis seisundis	Selles jaotises esitatud lahendus täidab antud nõuet. Kui võetakse kasutusele seisundite ajaloo tabel, siis tuleb tagada, et põhiolemile vastavale tabelis oleva reaga peab olema seotud vähemalt üks rida seisundite ajaloo tabelis. Seda kontrolli saab teha näiteks andmete andmebaasis registreerimiseks mõeldud rutiinides.
	Olem saab olla korraga maksimaalselt ühes seisundis	Tagatud andmebaasi struktuuri ja kitsendustega – tabeli *Tellimus* primaarvõti välistab korduvad read, st sama tellimus ei saa olla registreeritud korduvalt, kuid erinevate seisunditega ning iga tellimuse kohta registreeritakse täpselt üks seisund.
	Olem saab olla korraga mitmes seisundis	Tabeli *Tellimus* välisvõti (tellimuse_seisundi_liik_kood) tuleb eemaldada, st eemaldada tuleb nii veerg kui kitsendus. [19] Tuleb luua vahetabel, mis aitab siduda seisundiklassifikaatori ja põhiolemitüübi tabelites olevaid ridu. Vahetabelis on välisvõtmed (tellimuse_nr) ja (tellimuse_seisundi_liik_kood), mis viitavad vastavalt *Tellimus* ja *Tellimuse_seisundi_liik* tabelite primaarvõtmetele. Uue vahetabeli primaarvõtmeks oleks (tellimuse_nr, tellimuse_seisundi_liik_kood). Antud lahendust kirjeldab ka Silverstoni kolmanda taseme muster [11].
2	Olemi puhul tuleb teada ainult selle hetkeseisundit	Selles jaotises esitatud lahendus täidab antud nõuet.
	Olemi puhul tuleb teada selle seisundite ajalugu	Silverston [11] pakub kolmanda taseme mustris, et vahetabelit, mida kasutatakse selleks kui olem saab olla korraga mitmes seisundis, võib kasutada ka seisundite ajaloo hoidmiseks. Sellises tabelis tuleb primaarvõtmena kasutada surrogaatvõtit, sest primaarvõti / unikaalsuse kitsendus (tellimuse_nr, tellimuse_seisundi_liik_kood) jõustaks ilmselt liiga piirava reegli, mille kohaselt ei saa olem oma eluea jooksul olla korduvalt samas seisundis. Tabelisse tuleb lisada uusi veerge nagu *seisundimuudatuse_aeg* ning tabeli üheks kandidaatvõtmeks on (tellimuse_nr, seisundimuudatuse_aeg).
3	Olemi puhul huvitab ainult seisundi omamise fakt	Selles jaotises esitatud lahendus täidab antud nõuet.
	Olemi puhul on vaja lisainfot seisundimuudatuse kohta (millal see seisund omandati, kes selle muudatuse algatas jne)	Kui andmebaasis registreeritakse ainult hetkeseisund, siis tuleb vastavad veerud lisada põhiolemitüübile vastavasse tabelisse (antud juhul *Tellimus*). Kui andmebaasis registreeritakse seisundite ajalugu, siis tuleb vastavad veerud lisada seisundimuudatuste ajaloo tabelisse. Silverstoni [11] kolmanda taseme seisundi muster näeb ette, et iga põhiolemitüübi kohta tekib eraldi seisundimuudatuste tabel ning neljanda taseme

	Nõude kirjeldus	Lahendus
		seisundi muster näeb ette, et kõigile põhiolemitüüpidele tekib ühine seisundimuudatuste ajaloo tabel. Neljanda taseme mustri puudustena näen selle tabeli liigset suurust nii ridade kui veergude arvu poolest. Kui selles toimub samaaegselt palju andmemuudatusi ja andmete lugemisi, siis sellest võib saada töökiiruse mõttes pudelikael. Kui erinevate põhiolemitüüpide korral tuleb seisundimuudatuste kohta koguda erinevat liiki andmeid, siis viib see nimetatud vahetabelis suure hulga mittekohustuslike veergude (lubavad NULLe) tekkimiseni. Uue põhiolemitüübi lisandumisel tuleb muuta olemasoleva vahetabeli struktuuri (lisada sinna uus välisvõtme veerg/veerud + kitsendus), selle asemel, et lisada uus tabel.
4	Lubatavate seisundimuudatuste kontroll ilma olekutabelita	Lubatud seisundimuudatuste kontrollprogrammi tuleb sisse kirjutada seisundiklassifikaatorite koodid. Kui muutub klassifikaatori kood, siis tuleb muuta kontrollprogrammi. Kontrolli saab realiseerida trigrite abil.
	Lubatavate seisundimuudatuste kontroll olekutabeliga	Olekutabeli saab välisvõtmete kaudu siduda seisundiklassifikaatori tabeliga, st olekutabelis on lubatud kasutada vaid selliseid seisundiklassifikaatorite väärtuseid, mis on ka registreeritud seisundiklassifikaatori tabelis. Nendele välisvõtme kitsendustele saab määrata ON UPDATE CASCADE kompenseeriva tegevuse, st kui muutub seisundiklassifikaatori kood, siis muutub see ka automaatselt olekutabelis.Välisvõtme kitsendusele määratav ON DELETE CASCADE kompenseeriv tegevus tagab, et klassifikaatori väärtuse eemaldamisel eemaldatakse automaatselt ka sellega seotud üleminekud olekutabelist.
5	Andmebaasis ei ole vaja säilitada lisainfot seisundite kohta	Selles jaotises esitatud lahendus täidab antud nõuet.
	Andmebaasis on vaja säilitada lisainfot seisundite kohta (näiteks nimetus erinevates keeltes, kehtivus, kommentaar)	Lisada lisainfot sisaldavad veerud igasse seisundiklassifikaatori tabelisse või luua uus tabel, mis on seotud kõigi olemitüüpide seisundiklassifikaatorite tabelitega. Antud tabelis võib hoida infot näiteks kõigi seisundiklassifikaatorite väärtuste kehtivuse kohta.
6	Tekib uus võimalik seisund, milles olem võib viibida	Lisada uus võimalik seisund seisundiklassifikaatori tabelisse kasutades INSERT [41] lauset.
	Kaob seisund, milles olem võib viibida	Kustutada antud seisundi kohta käiv rida seisundiklassifikaatori tabelist kasutades selleks DELETE [42] lauset. Juhul kui mõni olemitüüp on hetkel antud seisundis, siis välisvõtme ON

	Nõude kirjeldus	Lahendus
		DELETE NO ACTION määrang ei luba seda seisundit klassifikaatori tabelist kustutada. Sellisel juhul võib luua uue tabeli, mis hoiab andmeid klassifikaatori väärtuste kehtivuse kohta ning on seejuures ühendatud klassifikaatori tabeli endaga. Teine võimalus on lisada vastav veerg seisundiklassifikaatori tabelisse ning muuta antud seisund mitteaktiivseks (st muuta selle seisundit).

4.4 Temporaalsed veerud

Nimi inglise keeles: Temporal columns

Alternatiivsed ingliskeelsed nimed: -

Allikad:

- „The Data Model Resource Book, Volume 3: Universal Patterns for Data Modeling“ [11]
- „How to design database for storing object states in mysql?“ [30]
- „SQL query to get status as of a given date“ [43]
- „How to handle status columns in designing tables?“ [29]

Lahendus: Põhiolemitüübile vastavas tabelis defineeritakse erinevat tüüpi sündmustele vastavad mittekohustuslikud (NULLe lubavad) temporaalsed (ajalised) veerud – iga sündmuse kohta üks. Väärtus sellises veerus näitab, et põhiolemiga on (mingil ajal) vastavat tüüpi sündmus toimunud. Sellest omakorda võib järeldada, et vastav olem on läinud mingisse seisundisse. Need veerud peaksid olema ajatempli (TIMESTAMP) tüüpi ning registreerima väärtuseid vähemalt sekundi täpsusega, sest nende andmete alusel määratakse, milline on olemi hetkeseisund. Algseisundile vastava veeru vaikimisi väärtuseks on avaldis, mis süsteemi kella alusel tagastab hetke ajatempli.

Eelised:

- Pakub lisainfot selle kohta, millal olem antud seisundi omandas. [30]
- Andmebaasi konspetuaalne skeem annab lisainfot võimalike sündmuste kohta, mis olemitega võib toimuda. Skeem on kasutajale lihtsamini mõistetav kui sündmuste hulk ei ole suur. [11]
- Päringute täitmiskiirused on (PostgreSQL 11 näitel) ühtlaselt head nii koondandmete leidmise päringute korral kui ka konkreetse olemi seisundi otsimisel. (vt peatükk 6)

- Väike andmemaht (PostgreSQL 11 näitel) kui tabeli veergude arv on kuni kaheksa (vt peatükk 6). Sellisel juhul ei kuluta PostgreSQL NULLide hoidmiseks üldse ruumi [44].

- Disaini realiseerimiseks vajalikud andmebaasikeele laused on (PostgreSQL 11 näitel) vähem keerukamad kui teiste kataloogis esitatavate disainide puhul. (vt peatükk 6)

- Seisundimuudatuseks läbiviidav operatsioon täidetakse (PostgreSQL 11 näitel) kiiremini kui teiste kataloogis esitatud disainide puhul, sest väärtust tuleb muuta ainult kahele tabeli veerule vastavas väljas ning see ei kutsu välja ühtegi muud lisategevust või kontrolli. (vt peatükk 6)

Puudused:

- Uue võimaliku seisundi lisandumine tähendab muudatust andmebaasi skeemis, sest selleks tuleb põhiolemitüübi tabelisse lisada uus veerg. Sarnaselt nõuab muudatusi tabelite struktuuris ka võimalike seisundite eemaldamine. Eelnev tingib omakorda vajaduse kirjutada ümber tabelile viitavaid andmekäitluskeele lauseid (vaadetes, rutiinides, rakenduses, testides). Muudatuste sisseviimine nõuab andmebaasi administraatori sekkumist (ei ole paindlik) või suuri õiguseid seisundite hulka haldavale rakendusele (ei ole turvaline) (vt jaotis 5.6.3).

- Võimaliku seisundite hulga vähendamine ja sellega seoses tabelist veeru või veergude eemaldamine võib põhjustada infokadu.

- Juhul kui võimalike seisundite hulk kasvab ajas, siis võib see mõjutada päringule vastuse saamiseks kuluvat aega. See võib juhtuda siis, kui tabeli veergude arv läheb nii suureks, et üks rida ei mahu ära ühte plokki. Sellisel juhul võib andmebaasisüsteem jaotada rea mitme plokki vahel ning andmete muutmiseks ja lugemiseks kulub rohkem aega. [23]

- Andmebaasisüsteemi maksimaalne lubatud veergude hulk (nt 1600 PostgreSQL korral [45]) seab piirangu maksimaalsele võimalike seisundite hulgale.

- Suur hulk seisundeid võib muuta raskemaks mõistmise, millised tabeli veerud on kasutusel olemi seisundite registreerimiseks. [11]

- Raske eristada temporaalseid veerge, mis on mõeldud seisundi registreerimiseks nendest, mida selleks otstarbeks ei kasutata.

- Ei võimalda liigitada seisundeid erinevatesse kategooriatesse. Näiteks tellimusel võib olla üks kategooria seisundeid, mis vastavad tellimuse töötlemis protsessile ja teine kategooria seisundeid, mis on rakendatavad tellimuse planeerimisele. [11]

- Seisundite nimed tuleb kodeerida sisse andmebaasirakendusse ja päringutesse. Nime muutmisel tuleb muudatusi teha mitmes kohas. (vt jaotis 5.6.1)

- Kui soovitakse andmebaasis registreerida reeglid seisundite vaheliste seoste kohta (olekutabel), siis tuleb andmebaasis mingil viisil registreerida vastavus seisundi nime ja selle esitamiseks kasutatava tabeli veeru vahel. [11] Kui veergude nimed muutuvad, siis tuleb muuta ka seda vastavust (see ei toimu automaatselt).

- Sündmuste esitamiseks mõeldud veerud peavad olema mittekohustuslikud (st tabelis võib olla palju NULLe). [21]

- Hetkeseisundi leidmiseks tuleb teha päring üle kõikide seisundite esitamiseks mõeldud veergude. Kui seisundite (veergude) hulk seal muutub, siis tuleb päring ümber kirjutada. (vt jaotis 5.6.1)

- Koondandmete leidmise päring on (PostgreSQL 11 näitel) keerukas. (vt peatükk 6)

Variatsioonid:

Tõeväärtustüüpi veerud põhiolemitüübi tabelis, mille kohta on eraldi disain kirjeldatud jaotises 1.1.

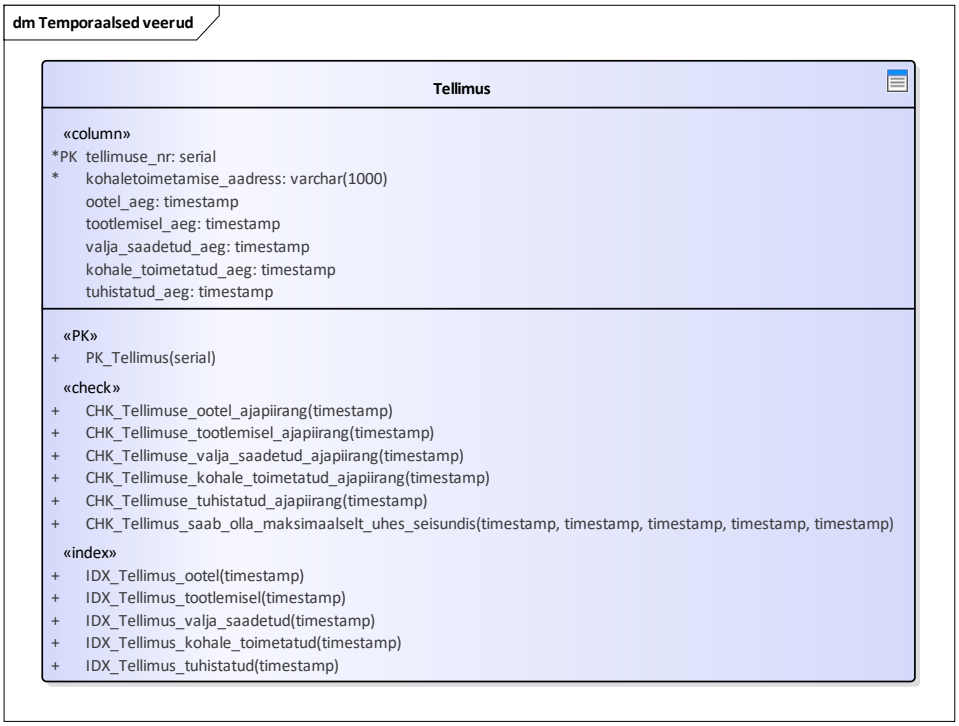
Näide:

Põhiolemitüübi *Tellimus* alusel luuakse tabel *Tellimus*:

tellimuse_nr	**kohale-toimetamise_aadress**	**ootel_aeg**	**töötlemisel_aeg**	**välja_saadetud_aeg**	**kohale_toimetatud_aeg**	**tühistatud_aeg**
1	Narva mnt 5	03.03.2019 12:13:15	NULL	NULL	NULL	NULL

tellimuse_nr	kohale-toimetamise_aadress	ootel_aeg	töötlemisel_aeg	välja_saadetud_aeg	kohale_toimetatud_aeg	tühistatud_aeg
2	Ale 6	NULL	14.03.2019 12:02:11	NULL	NULL	NULL
3	Ristiku 51	NULL	NULL	14.03.2019 14:02:14	NULL	NULL
4	Akadeemia tee 21a	NULL	NULL	NULL	14.03.2019 14:55:32	NULL
5	Soo 46	NULL	NULL	NULL	NULL	12.03.2019 12:00:02

Strateegia: Joonis 6 esitab *Temporaalsed veerud* disaini realisatsiooni PosgreSQL andmebaasisüsteemis.



Joonis 6. Temporaalsed veerud disaini andmebaasi diagramm.

Tagamaks jaotise 4.1 näite kirjelduses esitatud nõuete (1.1 ja 1.2) täidetavust loon CHECK kitsenduse (Joonis 7), mis kontrollib, et iga tellimus on alati täpselt ühes seisundis:

```
CREATE CONSTRAINT CHK_Tellimus_saab_olla_maksimaalselt_uhes_seisundis CHECK
(num_nonnulls(ootel_aeg,tootlemisel_aeg,kohale_toimetatud_aeg,valja_saadetud_
aeg,tuhistatud_aeg)=1);
```

Joonis 7. Kitsenduse loomise lause.

Samuti loon otsingute kiirendamiseks osalised indeksid mittekohustuslike seisundite veergudele [46] (Joonis 8):

```
CREATE INDEX IDX_Tellimus_ootel ON Tellimus(ootel_aeg) WHERE ootel_aeg IS NOT NULL;
CREATE INDEX IDX_Tellimus_tootlemisel ON Tellimus(tootlemisel_aeg) WHERE tootlemisel_aeg IS NOT NULL;
CREATE INDEX IDX_Tellimus_valja_saadetud ON Tellimus(valja_saadetud_aeg) WHERE valja_saadetud_aeg IS NOT NULL;
CREATE INDEX IDX_Tellimus_kohale_toimetatud ON Tellimus(kohale_toimetatud_aeg) WHERE kohale_toimetatud_aeg IS NOT NULL;
CREATE INDEX IDX_Tellimus_tuhistatud ON Tellimus(tuhistatud_aeg) WHERE tuhistatud_aeg IS NOT NULL;
```

Joonis 8. Osaliste indeksite loomise laused.

Lisaks kasutan iga seisundi veeru omaduste kirjeldamiseks domeeni [40] (Joonis 9), millega seotud CHECK kitsendus kontrollib, et sisestatav ajaline väärtus oleks suurem kui 31.12.1999 ja väiksem kui 01.01.2200 (diagrammil domeeni ei esitata):

```
CREATE DOMAIN D_ajapiirang TIMESTAMP WITHOUT TIME ZONE NULL CONSTRAINT chk_ajapiirang CHECK (VALUE>='2000-01-01' AND VALUE<'2200-01-01');
```

Joonis 9. Domeeni loomise lause.

Nõuded: Tabel 3 esitab lahendused kuidas olla vastavuses nõuete klassidest tulenevatele tingimustele (vt jaotis 4.1).

Tabel 3. Temporaalsed veerud lahendused nõuetele.

	Nõude kirjeldus	**Lahendus**
1	Olem peab olema kindlasti mingis seisundis	Luua CHECK kitsendus üle kõigi seisundite jaoks mõeldud veergude, mis kontrollib, et igal ajahetkel oleks igal olemil registreeritud väärtus vähemalt ühe seisundi kohta.
	Olem saab olla korraga maksimaalselt ühes seisundis	Andmebaasi tasemel saab kitsendusena jõustada juhul, kui ei säilitata seisundite ajalugu ning iga tabelis oleva rea korral kontrollitakse CHECK kitsendusega, et selles on väärtus määratud täpselt ühes seisundile vastavas veerus. Kui säilitatakse seisundite ajalugu, siis seda, millises seisundis konkreetne olem parajasti viibib, peavad interpreteerima andmebaasi kasutatavad programmid kasutades andmebaasis olevaid andmeid. Hetkeseisundi leidmiseks tuleb luua päring, mis võrdleb konkreetse põhiolemi korral omavahel ajalisi väärtuseid kõigis seisundi infot hoidvates veergudes. Päring leiab seisundi selle alusel, milline ajaline väärtus on kõige suurem. Seisundi nime määramiseks saab kasutada CASE avalist. Hetkeseisundi ja sellesse mineku aja leidmiseks saab kasutada GREATEST funktsiooni [47].

	Nõude kirjeldus	**Lahendus**
	Olem saab olla korraga mitmes seisundis	Selles jaotises esitatud lahendus täidab antud nõuet. Seda, millises seisundis konkreetne olem parajasti viibib, peavad interpreteerima andmebaasi kasutavad programmid kasutades andmebaasis olevaid andmeid.
2	Olemi puhul tuleb teada ainult selle hetkeseisundit	Sündmustele vastavad veerud peavad olema mittekohustuslikud. Põhiolemi seisundimuudatuse korral tuleb selle kohta käivast reast kustutada eelmistele sündmustele vastavad väärtused. CHECK kitsendusega tuleb kontrollida, et seisunditele vastavate veergude kohta on registreeritud ainult üks väärtus.
	Olemi puhul tuleb teada selle seisundite ajalugu	Algseisundile vastav veerg peab olema kohustuslik ning ülejäänud seisunditele vastavad veerud on mittekohustuslikud. Selles jaotises esitatud lahendus täidab antud nõuet selle klausliga, et ei ole võimalik säilitada infot selle kohta, et sama olem oli samas seisundis korduvalt. Näiteks kui sama tellimus oli seisundis ootel, läks seisundisse töötlemisel ja siis uuesti seisundisse ootel, siis sellisel juhul kirjutatakse andmebaasis esimene ootele mineku aeg üle ning fakt selle toimumise kohta kaob andmebaasist.
3	Olemi puhul huvitab ainult seisundi omamise fakt	Selles jaotises esitatud lahendus täidab antud nõuet.
	Olemi puhul on vaja lisainfot seisundimuudatuse kohta (millal see seisund omandati, kes selle muudatuse algatas jne)	Selles jaotises esitatud lahendus võimaldab registreerida lisainfot selle kohta, millal see seisund omandati. Juhul kui on vajalik veel lisainfot, siis tuleb luua uued veerud selles samas tabelis ning CHECK kitsendused, mis tagaksid, et kui seisundisse viivale sündmusele vastavas veerus registreeritakse väärtus, siis tuleb registreerida andmed ka osades või kõigis nendes veergudes. Selliste kitsenduste loogikaavaldis peab olema kujul NOT (A) OR B. See kitsendus jõustab reegli, et kui on täidetud tingimus A, siis peab olema täidetud ka tingimus B. Näiteks: NOT (kinnitamise_aeg IS NOT NULL) OR (kinnitaja IS NOT NULL) – kui kinnitamise aeg on määratud, siis peab ka kinnitaja olema määratud. Eraldi seisundimuudatuste ajaloo tabeliga on see probleem, et info seisundimuudatuste kohta oleks sellisel juhul jagatud erinevate tabelite vahel – seisundimuudatuse aeg ühes tabelis ning täiendavad andmed seisundimuudatuse kohta (nt muutja) teises tabelis.

	Nõude kirjeldus	**Lahendus**
4	Lubatavate seisundimuudatuste kontroll ilma olekutabelita	Kuna kõigi võimalike seisundite jaoks on loodud temporaalsed (ajalisi väärtuseid sisaldavad) veerud põhiolemitüübi tabelis, siis on võimalik kasutada kontrolliks selle tabeliga seotud trigerit.
	Lubatavate seisundimuudatuste kontroll olekutabeliga	Olekutabelis on vaja viidata põhiolemitüübi tabelis sündmuste registreerimiseks kasutatavate veergude nimedele.
5	Andmebaasis ei ole vaja säilitada lisainfot seisundite kohta	Selles jaotises esitatud lahendus täidab antud nõuet.
	Andmebaasis on vaja säilitada lisainfot seisundite kohta (näiteks nimetus erinevates keeltes, kehtivus, kommentaar)	Säilitamaks lisainfot seisundite kohta tuleks luua uus tabel. Üks võimalus oleks see tabel ülesehitada nii, et seal oleksid erinevate parameetrite väärtused. Näiteks võib luua tabelisse järgnevad veerud: *seisund_nimi*, *parameeter* ja *väärtus*. Näiteks üks rida selles tabelis sisaldab vastavalt väärtuseid „ootel“, „nimetus ENG“ ja „pending“. Selleks, et kuvada põhiolemi hetkeseisundi nimi inglise keeles, tuleks leida põhiolemitüübi tabelist, selle hetkeseisundi nimetus eesti keeles. Peale seisundi nimetuse leidmist tuleks parameetrite tabelist vastavalt leitud seisundi nimetusele otsida üles rida, mille parameetri kirjeldus on „nimetus ENG“ ning tagastada sõna veerust *väärtus*. Parameetrite tabeli omaduseks on, et sama olemitüübi erinevate seisundite korral võib seal olla registreeritud erinevale hulgale parameetritele vastavad väärtused. Veel üks lahendus on luua seisundiklassifikaatori tabel (vt jaotis 4.3) puhtalt metaandmete hoidmise jaoks.
6	Tekib uus võimalik seisund, milles olem võib viibida	Selleks tuleb lisada uus ajalist väärtust lubav veerg põhiolemitüübi tabelisse, kasutades selleks ALTER [48] lauset.
	Kaob seisund, milles olem võib viibida	Selleks tuleb kustutada veerg põhiolemitüübi tabelist, kasutades selleks ALTER [48] lauset.

4.5 Tõeväärtustüüpi veerud

Nimi inglise keeles: Columns with the Boolean type

Alternatiivsed ingliskeelsed nimed: *Create Multiple Columns*

Allikad:

- „The Data Model Resource Book, Volume 3: Universal Patterns for Data Modeling“ [11]
- „SQL Antipatterns: Avoiding the Pitfalls of Database Programming“ [19]
- „How to handle status columns in designing tables?“ [29]
- „Database Systems: An Application-Oriented Approach (Complete Version)“ [49]
- „Picking datatype for STATUS fields“ [50]

Lahendus: Põhiolemitüübile vastavas tabelis defineeritakse erinevatele seisunditele vastavad kohustuslikud (NOT NULL) tõeväärtustüüpi (Boolean) veerud – iga seisundi kohta üks. Väärtus sellises veerus näitab, et põhiolem on vastavas seisundis. Algseisundile vastavale veerule tuleb defineerida vaikimisi väärtus TRUE.

Eelised:

- Andmebaasi konspetuaalne skeem annab lisainfot võimalike seisundite kohta, milles olemid võivad viibida. Skeem on kasutajale lihtsamini mõistetav kui seisundite hulk ei ole suur. [11]
- Võrreldes disainiga *Temporaalsed veerud* (vt jaotis 1.1) ei teki andmebaasi mittekohustuslikke veerge. Tõeväärtustüüpi veerud peavad olema kohustuslikud (NOT NULL kitsendusega) ning kui konkreetne olem pole vastavas seisundis, siis on sellele veerule vastavas väljas väärtus FALSE.

- Päringute täitmiskiirused on (PostgreSQL 11 näitel) head nii konkreetse olemi ja selle seisundi otsimisel kui ka koondandmete leidmise päringu puhul. (vt peatükk 6)

- Tõeväärtustüüpi väärtuste salvestamine nõuab andmebaasis vähe salvestusruumi (näiteks üks bait ühe väärtuse kohta PostgreSQLis). [51]

Puudused:

- Uue võimaliku seisundi lisandumine tähendab muudatust andmebaasi skeemis, sest selleks tuleb põhiolemitüübi tabelisse lisada uus veerg. Sarnaselt nõuab muudatusi tabelite struktuuris ka võimalike seisundite eemaldamine. Eelnev tingib omakorda vajaduse kirjutada ümber tabelile viitavaid andmekäitluskeele lauseid (vaadetes, rutiinides, rakenduses, testides). Muudatuste sisseviimine nõuab andmebaasi administraatori sekkumist (ei ole paindlik) või suuri õiguseid seisundite hulka haldavale rakendusele (ei ole turvaline). Ka on antud operatsioonidele vastav SQL kood (PostgreSQL 11 näitel) seetõttu üsna keerukas. (vt peatükk 6)

- Juhul kui võimalike seisundite hulk kasvab ajas, siis võib see mõjutada päringule vastuse saamiseks kuluvat aega. See võib juhtuda siis, kui tabeli veergude arv läheb nii suureks, et üks rida ei mahu ära ühte plokki. Sellisel juhul võib andmebaasisüsteem jaotada rea mitme plokki vahel ning andmete muutmiseks ja lugemiseks kulub rohkem aega. [23]

- Andmebaasisüsteemi maksimaalne lubatud veergude hulk (nt 1600 PostgreSQL korral [45]) seab piirangu maksimaalsele võimalike seisundite hulgale.

- Suur hulk seisundeid võib muuta raskemaks mõistmise, millised tabeli veerud on kasutusel olemi seisundite registreerimiseks. [11]

- Raske eristada tõeväärtustüüpi veerge, mis on mõeldud seisundi registreerimiseks nendest, mida selleks otstarbeks ei kasutata.

- Ei võimalda liigitada seisundeid erinevatesse kategooriatesse. Näiteks tellimusel võib olla üks kategooria seisundeid, mis vastavad tellimuse töötlemis protsessile ja teine kategooria seisundeid, mis on rakendatavad tellimuse planeerimisele. [11]

- Seisundite nimed tuleb kodeerida sisse andmebaasirakendusse ja päringutesse. Nime muutmisel tuleb muudatusi teha mitmes kohas. (vt jaotis 5.6.1)

- Kui soovitakse andmebaasis registreerida reeglid seisundite vaheliste seoste kohta (olekutabel), siis tuleb andmebaasis mingil viisil registreerida vastavus seisundi nime ja selle esitamiseks kasutatava tabeli veeru vahel. [11] Kui veergude nimed muutuvad, siis tuleb muuta ka seda vastavust (see ei toimu automaatselt).

- Hetkeseisundi leidmiseks tuleb teha päring üle kõikide seisundite esitamiseks mõeldud veergude. Kui seisundite (veergude) hulk seal muutub, siis tuleb päring ümber kirjutada. (vt jaotis 5.6.1)

- Koondandmete otsimise päring on (PostgreSQL 11 näitel) keerukas. (vt peatükk 6)

- Kuigi tõeväärtuste hoidmiseks kulub vähe salvestusruumi tähendab veergude kohustuslikkus, et igas rea väljas peab olema väärtus ning kokkuvõtes võib (PostgreSQL 11 näitel) see tähendada suuremat salvestusruumi kasutamist kui NULLe lubav temporaalset tüüpi veergudega disain. (vt peatükk 6)

Variatsioonid:

- Kui andmebaasisüsteem tõeväärtustüüpe ei toeta, siis tuleb kasutada täisarvulisi väärtuseid 1 (kodeerib TRUE) ja 0 (kodeerib FALSE) või sõnalisi väärtuseid nagu näiteks Y (kodeerib TRUE) ja N (kodeerib FALSE).

- *Temporaalsed veerud* põhiolemitüübi tabelis, mille kohta on eraldi disain kirjeldatud jaotises 4.4.

Näide:

Põhiolemitüübi *Tellimus* alusel luuakse tabel *Tellimus*:

tellimuse_ nr	**kohale-toimetamise_ aadress**	**on_ ootel**	**on_ töötlemisel**	**on_välja_ saadetud**	**on_ kohale_ toimetatud**	**on_ tühistatud**	**seisundi-muudatuse _aeg**
1	Narva mnt 5	TRUE	FALSE	FALSE	FALSE	FALSE	03.03.2019 12:13:15
2	Ale 6	FALSE	TRUE	FALSE	FALSE	FALSE	14.03.2019 12:02:11

tellimuse_ nr	kohale-toimetamise_ aadress	on_ ootel	on_ töötlemisel	on_välja_ saadetud	on_ kohale_ toimetatud	on_ tühistatud	seisundi-muudatuse _aeg
3	Ristiku 51	FALSE	FALSE	TRUE	FALSE	FALSE	14.03.2019 14:02:14
4	Akadeemia tee 21a	FALSE	FALSE	FALSE	TRUE	FALSE	14.03.2019 14:55:32
5	Soo 46	TRUE	FALSE	FALSE	FALSE	TRUE	12.03.2019 12:00:02

Strateegia: Joonis 10 esitab *Tõeväärtustüüpi veerud* disaini realisatsiooni PosgreSQL andmebaasisüsteemis.

Joonis 10. Tõeväärtustüüpi veerud disaini andmebaasi diagramm.

Tagamaks 4.1 näite kirjelduses esitatud nõuete (1.1 ja 1.2) täidetavust loon CHECK kitsenduse (Joonis 11), mis kontrollib, et iga tellimus on alati täpselt ühes seisundis:

```
CREATE CONSTRAINT CHK_Tellimus_saab_olla_maksimaalselt_uhes_seisundis CHECK
(((on_ootel=true)::int + (on_tootlemisel=true)::int +
(on_valja_saadetud=true)::int + (on_kohale_toimetatud=true)::int +
(on_tuhistatud=true)::int )=1);
```

Joonis 11. Kitsenduse loomise lause.

Lisaks loon ka trigeri (Joonis 12), mis tagab (vastavalt nõudele 3.2), et tellimuse seisundi väärtuse muutumisel muudetakse ka seisundimuudatuse aega:

```
CREATE OR REPLACE FUNCTION F_kontroll_seisundi_vaartuse_ja_aja_muutusele()
RETURNS TRIGGER AS $$
BEGIN
NEW.seisundimuudatuse_aeg = LOCALTIMESTAMP(0);
RETURN NEW;
END;$$ LANGUAGE plpgsql SECURITY DEFINER
SET search_path = public, pg_temp;

CREATE TRIGGER T_kontroll_seisundi_vaartuse_ja_aja_muutusele
BEFORE UPDATE OF
on_ootel,on_tootlemisel,on_valja_saadetud,on_kohale_toimetatud,on_tuhistatud
ON Tellimus
FOR EACH ROW
WHEN (NEW.on_ootel<>OLD.on_ootel OR NEW.on_tootlemisel<>OLD.on_tootlemisel OR
NEW.on_valja_saadetud<>OLD.on_valja_saadetud OR
NEW.on_kohale_toimetatud<>OLD.on_kohale_toimetatud OR
NEW.on_tuhistatud<>OLD.on_tuhistatud)
EXECUTE FUNCTION F_kontroll_seisundi_vaartuse_ja_aja_muutusele();
```

Joonis 12. Trigeri loomise laused.

Samuti loon otsingute kiirendamiseks osalised indeksid kohustuslikele sündmuste veergudele [46] (Joonis 13), sest väärtus TRUE on suhteliselt väikeses hulgas ridades:

```
CREATE INDEX IDX_Tellimus_ootel ON Tellimus(on_ootel) WHERE on_ootel = TRUE;
CREATE INDEX IDX_Tellimus_tootlemisel ON Tellimus(on_tootlemisel) WHERE
on_tootlemisel = TRUE;
CREATE INDEX IDX_Tellimus_valja_saadetud ON Tellimus(on_valja_saadetud) WHERE
on_valja_saadetud = TRUE;
CREATE INDEX IDX_Tellimus_kohale_toimetatud ON Tellimus(on_kohale_toimetatud)
WHERE on_kohale_toimetatud = TRUE;
CREATE INDEX IDX_Tellimus_tuhistatud ON Tellimus(on_tuhistatud) WHERE
on_tuhistatud = TRUE;
```

Joonis 13. Osaliste indeksite loomise laused.

Nõuded: Tabel 4 esitab lahendused kuidas olla vastavuses nõuete klassidest tulenevatele tingimustele (vt jaotis 4.1).

Tabel 4. Tõeväärtustüüpi veerud lahendused nõuetele.

	Nõude kirjeldus	**Lahendus**
1	Olem peab olema kindlasti mingis seisundis	Luua CHECK kitsendus üle kõigi seisundite jaoks mõeldud veergude, mis kontrollib, et igal ajahetkel oleks igal olemil registreeritud TRUE vähemalt ühe seisundi kohta.
	Olem saab olla korraga maksimaalselt ühes seisundis	Luua CHECK kitsendus üle kõigi seisundite jaoks mõeldud veergude, mis kontrollib, et igal ajahetkel oleks igal olemil registreeritud väärtus TRUE täpselt ühe seisundi kohta.
	Olem saab olla korraga mitmes seisundis	Selles jaotises esitatud lahendus täidab antud nõuet.

	Nõude kirjeldus	**Lahendus**
2	Olemi puhul tuleb teada ainult selle hetkeseisundit	Selles jaotises esitatud lahendus täidab antud nõuet.
	Olemi puhul tuleb teada selle seisundite ajalugu	Võimalik on täiendada põhiolemitüübile vastavat tabelit temporaalsete veergudega (vt jaotis 4.4) või võtta kasutusele seisundimuudatuste ajaloo tabel (vt jaotis 4.3). Esimesel juhul ei ole võimalik säilitada infot selle kohta, et sama olem oli samas seisundis korduvalt. Teine eeldaks ajaloo säilitamise tarbeks ka seisundiklassifikaatorite tabeli loomist. Sellisel juhul on andmed põhiolemitüübi võimalike seisundite hulga kohta esitatud kahes kohas – põhiolemitüübile vastavas tabelis veergudena ning klassifikaatori tabelis ridadena. Selle hulga muutumisel tuleb muudatus teha mõlemas kohas.
3	Olemi puhul huvitab ainult seisundi omamise fakt	Selles jaotises esitatud lahendus täidab antud nõuet.
	Olemi puhul on vaja lisainfot seisundimuudatuse kohta (millal see seisund omandati, kes selle muudatuse algatas jne)	Võimaluseks on luua eraldi seisundimuudatuste ajaloo tabel ning lisada sinna vajalikud veerud. Mis puudutab vastavate veergude lisamist põhiolemitüübile vastavasse tabelisse, siis see on võimalik, kuid eeldab mitmesuguste reeglite realiseerimist. Näited: - Kui olem lahkub seisundist s, siis tuleb kustutada väärtused veergudest, mis sisaldavad lisainfot seisundi s omandamise kohta. - Kui olem jõuab seisundisse s, siis tuleb registreerida väärtused veergudes, mis sisaldavad lisainfot seisundi s omandamise kohta. Lisaks peavad kõik need lisainfo veerud olema mittekohustuslikud, st tabelis on palju NULLe.
4	Lubatavate seisundimuudatuste kontroll ilma olekutabelita	Kuna kõigi võimalike seisundiväärtuste jaoks on loodud tõeväärtustüüpi veerud põhiolemitüübi tabelis, siis on võimalik kasutada kontrolliks selle tabeliga seotud trigerit.
	Lubatavate seisundimuudatuste kontroll olekutabeliga	Olekutabelis on vaja viidata põhiolemitüübi tabelis seisundite registreerimiseks kasutatavate veergude nimedele.
5	Andmebaasis ei ole vaja säilitada lisainfot seisundite kohta	Selles jaotises esitatud lahendus täidab antud nõuet.
	Andmebaasis on vaja säilitada lisainfot seisundite	Säilitamaks lisainfot seisundite kohta tuleks luua uus tabel. Üks võimalus oleks see tabel ülesehitada nii, et seal

	Nõude kirjeldus	**Lahendus**
	kohta (näiteks nimetus erinevates keeltes, kehtivus, kommentaar)	oleksid erinevate parameetrite väärtused. Näiteks võib luua tabelisse järgnevad veerud: *seisund_nimi*, *parameeter* ja *väärtus*. (vt jaotis 4.4) Veel üks lahendus on luua seisundiklassifikaatori tabel (vt jaotis 4.3) puhtalt metaandmete hoidmise jaoks.
6	Tekib uus võimalik seisund, milles olem võib viibida	Selleks tuleb lisada tõeväärtustüüpi väärtust lubav veerg põhiolemitüübi tabelisse, kasutades selleks ALTER [48] lauset.
	Kaob seisund, milles olem võib viibida	Selleks tuleb kustutada veerg põhiolemitüübi tabelist, kasutades selleks ALTER [48] lauset.

4.6 Vektorkodeerimine

Nimi inglise keeles: Vectorcoding

Alternatiivsed ingliskeelsed nimed: *Multiple propositions as a Data Value, Format Comma-Separated Lists*

Allikad:

- „SQL Antipatterns: Avoiding the Pitfalls of Database Programming“ [19]
- „How to handle status columns in designing tables?“ [29]
- „Mitme väite ühe andmeväärtusena esitamise eelised ja puudused SQL-andmebaasides“ [25]

Lahendus: Põhiolemitüübile vastavas tabelis luuakse kohustuslik (NOT NULL) tekstitüüpi veerg. Iga sellises veerus olev string e sõne kasutab vektorkodeerimist ehk iga väärtus selles veerus esitab mitu väidet reaalse maailma olemite kohta. String võib olla ülesehitatud selliselt, et 1 mingis positsioonis tähendab, et olem on parajasti mingis seisundis ning 0 tähendab, et ei ole.

Eelised:

- Võrreldes disainiga *Temporaalsed veerud* ei teki andmebaasi mittekohustuslikke veerge.
- Uue seisundi lisandumine või olemasoleva eemaldamine ei nõua uue veeru või tabeli lisamist. Küll aga on vaja muuta programme (sh SQL andmekäitluskeele laused, andmebaasiserveris talletatud rutiinid, rakendus), mis peavad neid väärtuseid interpreteerima. (vt jaotis 5.6.3)
- Eeldusel, et erinevaid võimalikke seisundeid on vähe ja seisundite kodeerimiseks kasutatakse võimalikult lihtsat koodi, siis seisundit esitav väärtus võtab kettal ja mälus vähe ruumi. (vt peatükk 6)
- Päringud, millega otsitakse olemile vastavat seisundit, on (PostgreSQL 11 näitel) kiired ning lihtsad. (vt peatükk 6)

- Seisundimuudatuseks vajalik SQL kood pole (PostgreSQL 11 näitel) keeruline ja see operatsioon viiakse võrreldes teiste disainidega kiirelt läbi. (vt peatükk 6)

Puudused:

- Seoses uute seisundite lisamisega võib olla vaja suurendada vektorite hoidmiseks mõeldud väljas maksimaalset lubatud väärtuse pikkust. [19] Sellisel juhul tuleb ka muuta vektorkoodi kodeerimise ja dekodeerimise funktsioone. Selleks, et muudatust oleks võimalikult lihtne teha, võiks andmebaasi realiseerida nii, et vektorkoodi hoidval veerul on väljapikkus suur. Lisaks on veerul kitsendus, mis kontrollib, et väljas olev väärtus ei ületa vektorkoodile ettenähtud maksimaalset pikkust. Kui on vaja hakata kodeerima uut seisundit, siis tuleb kustutada ja uuesti luua see kitsendus, kuid ei ole vaja muuta veeru väljapikkust.

- Andmebaasi kontseptuaalne skeem (tabeli struktuur) ei anna lisainfot võimalike seisundite kohta, milles olemid võivad viibida.

- Kodeeritud andmete kasutamine otsingu tingimuses, kodeeritud andmete alusel tulemuse sorteerimine ja seisundi nime esitamine päringu tulemuses (et see näiteks rakenduses kuvada) nõuab vektrorkoodi dekodeerimise funktsiooni loomist ja kasutamist. [25]

- Nii deklaratiivselt kui imperatiivselt (trigeritega) jõustatud kitsenduste realiseerimiseks läheb vaja vektorkoodi dekodeerimise funktsiooni. [25]

- Kodeeritud andmete struktuuri muutmiseks on vaja muuta kodeerimise funktsiooni. (vt jaotis 5.6.3)

- Ei võimalda liigitada seisundeid erinevatesse kategooriatesse. Näiteks tellimusel võib olla üks kategooria seisundeid, mis vastavad tellimuse töötlemis protsessile ja teine kategooria seisundeid, mis on rakendatavad tellimuse planeerimisele. [11]

- Seisundite nimed tuleb kodeerida sisse andmebaasirakendusse ja päringutesse juhul kui ei looda eelnevalt kirjeldatud funktsiooni. Nime muutmisel tuleb muudatusi teha mitmes kohas, mis suurendab ka lausete keerukust.

- Kui soovitakse andmebaasis registreerida reeglid seisundite vaheliste seoste kohta (olekutabel), siis tuleb andmebaasis mingil viisil registreerida vastavus seisundi nime ja selle esitamiseks kasutatava vektorkoodi positsiooni vahel. [11] Kui veergude nimed muutuvad, siis tuleb muuta ka seda vastavust (see ei toimu automaatselt).

- Tüüpilised päringud on sellised, et andmebaasisüsteem ei kasutaks veerule loodud „tavalisi“ (B-puu) indekseid. Selle tulemuseks on tabeli täielik läbiskaneerimine päringu täitmiseks [29]. Veeru põhjal otsingute tegemise kiirendamine nõuab funktsioonil põhinevate indeksite loomist [25].

Variatsioonid:

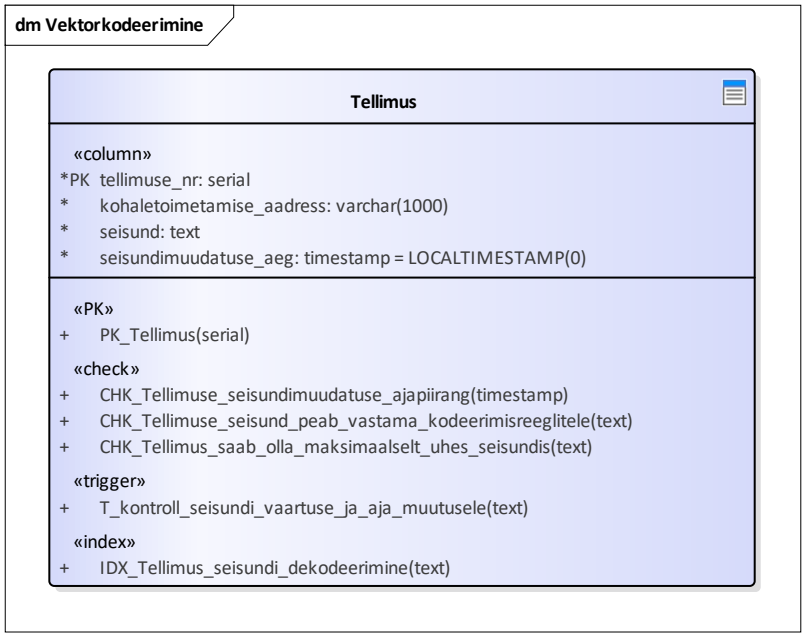
Disainid, mille puhul vektorkoodiga veeru asemel on kasutusel massiivi tüüpi veerg, JSON tüüpi veerg või JSONB tüüpi veerg. PostgreSQL andmebaasisüsteemis oleks selleks otstarbeks võimalik kasutada ka XML tüüpi veergu. [25]

Näide:

Põhiolemitüübi *Tellimus* alusel luuakse tabel *Tellimus*:

tellimuse_nr	**kohaletoimetamise_aadress**	**seisund**	**seisundimuudatuse_aeg**
1	Narva mnt 5	10000	03.03.2019 12:13:15
2	Ale 6	01000	14.03.2019 12:02:11
3	Ristiku 51	00100	14.03.2019 14:02:14
4	Akadeemia tee 21a	00010	14.03.2019 14:55:32
5	Soo 46	00001	12.03.2019 12:00:02

Strateegia: Joonis 14 esitab *Vektorkodeerimine* disaini realisatsiooni PosgreSQL andmebaasisüsteemis.



Joonis 14. Vektorkodeerimine disaini andmebaasi diagramm.

Tagamaks 4.1 näite kirjelduses esitatud nõuete (1.1 ja 1.2) täidetavust loon CHECK kitsenduse (Joonis 15), mis kontrollib, et iga tellimus on alati täpselt ühes seisundis:

```
CREATE CONSTRAINT CHK_Tellimus_saab_olla_maksimaalselt_uhes_seisundis CHECK
(seisund LIKE '%1%'AND seisund NOT LIKE '%1%1%');
```

Joonis 15. Kitsenduse loomise lause.

Samuti loon CHECK kitsenduse (Joonis 16), mis kontrollib, et iga tellimuse seisundi väärtus vastaks kodeerimisreeglitele:

```
CREATE CONSTRAINT CHK_Tellimuse_seisund_peab_vastama_kodeerimisreeglitele
CHECK (seisund~'^([0-1]{5})$');
```

Joonis 16. Kitsenduse loomise lause.

Lisaks loon ka trigeri (Joonis 17), mis tagab (vastavalt nõudele 3.2), et tellimuse seisundi väärtuse muutumisel muudetakse ka seisundimuudatuse aega:

```
CREATE OR REPLACE FUNCTION F_kontroll_seisundi_vaartuse_ja_aja_muutusele()
RETURNS TRIGGER AS $$
BEGIN
NEW.seisundimuudatuse_aeg = LOCALTIMESTAMP(0);
RETURN NEW;
END;$$ LANGUAGE plpgsql SECURITY DEFINER
SET search_path = public, pg_temp;

CREATE TRIGGER T_kontroll_seisundi_vaartuse_ja_aja_muutusele
BEFORE UPDATE OF seisund ON Tellimus
FOR EACH ROW
WHEN (OLD.seisund<>NEW.seisund)
EXECUTE FUNCTION F_kontroll_seisundi_vaartuse_ja_aja_muutusele();
```

Joonis 17. Trigeri loomise laused.

Selleks, et kiirendada veerupõhiste seisundi väärtuste otsinguid, loon funktsiooni (Joonis 18), mis dekodeerib vektrokoodid neile vastavateks sõnalisteks seisundi esitusteks. Samuti loon otsingute kiirendamiseks sellel funktsioonil põhineva indeksi (Joonis 19). Samuti loon kodeerimise funktsiooni (Joonis 20), mida saab näiteks kasutada seisundimuudatuste tegemiseks. Need funktsioonid tagavad, et muutes vektorkoodi ülesehitust ei ole vaja ümberkirjutada seda koodi kasutavaid SQL lauseid.

```
CREATE OR REPLACE FUNCTION F_tellimuse_seisundi_dekodeerimine(seisund text)
RETURNS TEXT AS $$
SELECT CASE
       WHEN seisund = '10000' THEN 'Ootel'
       WHEN seisund = '01000' THEN 'Töötlemisel'
       WHEN seisund = '00100' THEN 'Välja saadetud'
       WHEN seisund = '00010' THEN 'Kohale toimetatud'
       WHEN seisund = '00001' THEN 'Tühistatud'
       ELSE 'Teadmata'
END AS seisundi_tulem;
$$ LANGUAGE sql IMMUTABLE STRICT;
```

Joonis 18. Dekodeerimise funktsiooni loomise lause.

```
CREATE INDEX IDX_tellimus_seisundi_dekodeerimine ON Tellimus
(f_tellimuse_seisundi_dekodeerimine(seisund));
```

Joonis 19. Funktsioonil põhineva indeksi loomise lause.

```
CREATE OR REPLACE FUNCTION F_tellimuse_seisundi_kodeerimine(seisund text)
RETURNS TEXT AS $$
SELECT CASE
    WHEN seisund = 'Ootel' THEN '10000'
    WHEN seisund = 'Töötlemisel' THEN '01000'
    WHEN seisund = 'Välja saadetud' THEN '00100'
    WHEN seisund = 'Kohale toimetatud' THEN '00010'
    WHEN seisund = 'Tühistatud' THEN '00001'
END AS seisundi_tulem;
$$ LANGUAGE sql IMMUTABLE STRICT;
```

Joonis 20. Kodeerimise funktsiooni loomise lause.

Nõuded: Tabel 5 esitab lahendused kuidas olla vastavuses nõuete klassidest tulenevatele tingimustele (vaata peatükk 4.1).

Tabel 5. Vektokodeerimine lahendused nõuetele.

	Nõude kirjeldus	**Lahendus**
1	Olem peab olema kindlasti mingis seisundis	Luua CHECK kitsendus vektorkoodi sisaldavale veerule, mis kontrollib, et igal ajahetkel oleks igal olemil registreeritud väärtus 1 vähemalt ühe seisundi kohta. Näide: CHECK (seisund LIKE '%1%')
	Olem saab olla korraga maksimaalselt ühes seisundis	Luua CHECK kitsendus vektorkoodi sisaldavale veerule, mis kontrollib, et igal ajahetkel oleks igal olemil registreeritud väärtus 1 täpselt ühe seisundi kohta.

	Nõude kirjeldus	**Lahendus**
		CHECK kitsenduse loogikaavaldises tuleb kasutada regulaaravaldist või LIKE predikaati, mis kontrollib, et vektorkoodis on täpselt üks 1. Näide: CHECK (seisund LIKE '%1%' AND seisund NOT LIKE '%1%1%')
	Olem saab olla korraga mitmes seisundis	Selles jaotises esitatud lahendus täidab antud nõuet.
2	Olemi puhul tuleb teada ainult selle hetkeseisundit	Selles jaotises esitatud lahendus täidab antud nõuet.
	Olemi puhul tuleb teada selle seisundite ajalugu	Võimalik on täiendada põhiolemitüübile vastavat tabelit temporaalsete veergudega (vt jaotis 4.4) või võtta kasutusele seisundimuudatuste ajaloo tabel (vt jaotis 4.3). Esimesel juhul ei ole võimalik säilitada infot selle kohta, et sama olem oli samas seisundis korduvalt. Teine eeldaks ajaloo säilitamise tarbeks ka seisundiklassifikaatorite tabeli loomist. Sellisel juhul on andmed põhiolemitüübi võimalike seisundite hulga kohta esitatud mitmes kohas – kodeerimise ja dekodeerimise funktsioonides lähtekoodi osana ning klassifikaatori tabelis ridadena. Selle hulga muutumisel tuleb muudatus teha mitmes kohas.
3	Olemi puhul huvitab ainult seisundi omamise fakt	Selles jaotises esitatud lahendus täidab antud nõuet.
	Olemi puhul on vaja lisainfot seisundimuudatuse kohta (millal see seisund omandati, kes selle muudatuse algatas jne)	Võimaluseks on luua eraldi seisundimuudatuste ajaloo tabel ning lisada sinna vajalikud veerud. Teiseks, keeruliseks ja mitte-eelistatud võimaluseks, on tekstilise vektorkoodi asemel hakata kasutama mitmemõõtmelist massiivi – üks massiiv esitab hetkeseisundit, teises on seisundi omandamise aeg, kolmandas seisundi muutja jne [25]. Probleemideks on näiteks keerukad päringud, deklaratiivsete kitsenduste jõustamise võimatus ja ka see, et mitte kõik andmebaasisüsteemid ei toeta massiivitüüpi veerge.
4	Lubatavate seisundimuudatuste kontroll ilma olekutabelita	Võimalik on kasutada kontrolliks tabeliga seotud trigerit. Trigeris on vaja kirjeldada lubatud vektorkoodi üleminekud (näiteks, et on lubatud üleminek 01000 => 00100).
	Lubatavate seisundimuudatuste kontroll olekutabeliga	Olekutabelis on vajab kirjeldada lubatud vektorkoodi üleminekud (näiteks, et on lubatud üleminek 01000 => 00100).

	Nõude kirjeldus	**Lahendus**
5	Andmebaasis ei ole vaja säilitada lisainfot seisundite kohta	Selles jaotises esitatud lahendus täidab antud nõuet.
	Andmebaasis on vaja säilitada lisainfot seisundite kohta (näiteks nimetus erinevates keeltes, kehtivus, kommentaar)	Säilitamaks lisainfot seisundite kohta tuleks luua uus tabel. Üks võimalus oleks see tabel ülesehitada nii, et seal oleksid erinevate parameetrite väärtused. Näiteks võib luua tabelisse järgnevad veerud: *seisund_nimi*, *parameeter* ja *väärtus*. (vt jaotis 4.4) Veel üks lahendus on luua seisundiklassifikaatori tabel (vt jaotis 4.3) puhtalt metaandmete hoidmise jaoks.
6	Tekib uus võimalik seisund, milles olem võib viibida	Võib tekkida vajadus suurendada vektorkoodi registreerimiseks mõeldud välja pikkust. Tuleb muuta kodeerimise ja dekodeerimise funktsioone. Kõik tabelis olevad vektorkoodid tuleb asendada uuega.
	Kaob seisund, milles olem võib viibida	Tuleb muuta kodeerimise ja dekodeerimise funktsioone ning vähendada vektorkoodi registreerimiseks mõeldud välja pikkust. Kõik tabelis olevad vektorkoodid tuleb asendada uuega.

4.7 Seisundite esitamine põhiolemitüübi alamtüüpidena

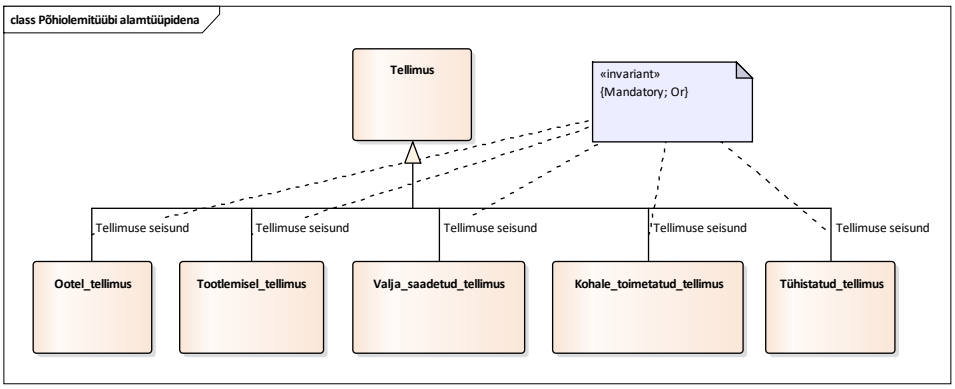
Nimi inglise keeles: Presenting states as subtypes of main types

Alternatiivsed ingliskeelsed nimed: Class Table Inheritance, Once Class One Table, Vertical Split

Allikad:

- „Disainimustrid üldistusseoste realiseerimiseks“ [24]
- „State pattern“ [52]
- „SQL Antipatterns: Avoiding the Pitfalls of Database Programming“ [19]

Lahendus: Kontseptuaalselt modelleeritakse (Joonis 21) seda kasutades üldistuste hulka (*generalization set*), mille puhul ülatüüpi olemeid liigitatakse seisundite alusel. Seega näiteks kontseptuaalses andmemudelis on *Tellimus* ülatüüp ning *Ootel_tellimus* ja *Töötlemisel_tellimus* selle alamtüübid. Andmebaasis realiseeritakse see nii, et ülatüübi ning iga alamtüübi kohta luuakse eraldi tabel. Igas alamtüübile vastavas tabelis on välisvõti, mis viitab ülatüübi alusel loodud tabeli primaarvõtmele. Selle välisvõtme puhul kasutatakse ON DELETE CASCADE kompenseerivat tegevust. See välisvõti on ühtlasi ka tabeli primaarvõti. Kui olem liigub ühest seisundist teise, siis tuleb selle kohta käiv rida kustutada vanale seisundile vastavast tabelist ja lisada rida uuele seisundile vastavasse tabelisse.



Joonis 21. Seisundite esitamine põhiolemitüübi alamtüüpidena konseptuaalne andmemudel.

Eelised:

- Andmebaasi konspetuaalne skeem annab lisainfot võimalike seisundite kohta, milles olemid võivad viibida. Skeem on kasutajale lihtsamini mõistetav kui seisundite hulk ei ole suur. [11]

- Võrreldes disainiga *Temporaalsed veerud* (vt jaotis 1.1) ei teki andmebaasi mittekohustuslikke veerge.

- Võimaldab erinevates seisundites olevate olemite kohta hoida erinevat lisainfot – vastavad veerud lisatakse seisundi alusel loodud tabelisse. Kui kõigi vastavas seisundis olevate olemite puhul on nende andmete registreerimine kohustuslik, siis saab andmebaasis need veerud deklareerida kohustuslikuks (NOT NULL).

- Uue seisundi lisandumine või olemasoleva eemaldamine nõuab olemasoleva tabeli struktuuri muutmise asemel uue tabeli lisamist või eemaldamist. See on andmebaasi kasutuse mõttes vähem-invasiivne (takistusi ja tõrkeid tekitav) muudatus. (vt jaotis 5.6.3)

- Lihtne teha päringut, mis leiab kõik mingis seisundis põhiolemid. Kui päring peab iga sellise põhiolemi kohta leidma vaid unikaalse identifikaatori, siis piisab andmete lugemisest konkreetsele seisundile vastavast tabelist. Samuti on (PostgreSQL 11 näitel) antud päringu täitmiskiirus ja keerukus võrreldes teiste töös esitatavate disainidega parem (vt peatükk 6).

Puudused:

- Võib juhtuda olukord, et olemi hetkeseisund on registreerimata – on rida põhiolemitüübile vastavas tabelis, kuid pole ühtegi vastavat rida seisunditele vastavates tabelites. Lahenduseks on luua kitsenduste triger, mis kontrollib, et vähemalt ühes alamtabelis peab eksisteerima rida põhitabelis registeeritud olemile [53].

- Päring, mis peab leidma kõikide olemite hetkeseisundi on keerukas (vajab ühendamise ja ühendi leidmise operatsioone) ja sellest tulenevalt ka potensiaalselt halva jõudlusega. Sama kehtib ka konkreetse olemi hetkeseisundi määramise

päringu kohta. Päringute kirjutamise lihtsustamiseks võib kasutada vaadet, kuid see ei paranda jõudlust.

- Olemi seisundi muutumise peegeldamine andmebaasis nõuab rea kustutamist ühest tabelist ning lisamist teise tabelisse, st muudatus on mitmeosaline ja võtab seetõttu rohkem aega. Muudatus tuleb läbi viia ühe tehinguna e transaktsioonina, et see oleks loogiline tervik, mida ei tehta osaliselt. Selline muudatus on (PostgreSQL 11 näitel) ka aeganõudev (vt peatükk 6).

- Keeruline on jõustada kitsendust, et iga olem saab olla korraga ainult ühes seisundis, st olemi identifikaator võib olla samal ajal vaid ühele seisundile vastavas tabelis. Üldiste kitsenduste (ASSERTION) toe puudumise tõttu ei saa tänapäeva SQL-andmebaasisüsteemis seda kitsendust realiseerida deklaratiivselt. PostgreSQL näitel tuleb selleks igale seisundile vastavale tabelile luua eraldi triger, mis suurendab antud disaini realiseerimiseks vajalike koodiridade arvu ning muudab raskemaks ka uue seisundi lisamise või olemasoleva eemaldamise. (vt peatükk 6)

- Erinevate seisundimuudatuste ja ülemineku reeglite realiseerimine on keeruline, sest kontrollid tuleb teha üle mitme tabeli. Antud kontrollide tegemine vähendab andmemuudatuste jõudlust.

- Seisundite nimed tuleb kodeerida sisse andmebaasirakendusse ja päringutesse. Nime muutmisel tuleb muudatusi teha mitmes kohas. (vt jaotis 5.6.1)

- Kui soovitakse andmebaasis registreerida reeglid seisundite vaheliste seoste kohta (olekutabel), siis tuleb andmebaasis mingil viisil registreerida vastavus seisundi nime ja selle esitamiseks kasutatava tabeli vahel. [11] Kui tabelite nimed muutuvad, siis tuleb muuta ka seda vastavust (see ei toimu automaatselt).

- Andmemaht on (PostgreSQL 11 näitel) võrreldes teiste disainidega suur, sest ühele olemile vastavaid andmeid leidub samaaegselt vähemalt kahes tabelis. (vt peatükk 6)

- Ei võimalda liigitada seisundeid erinevatesse kategooriatesse.

Variatsioonid:

- Lõputöö [24] esitab alternatiivseid võimalusi, kuidas SQL-andmebaasis realiseerida kontseptuaalses andmemudelis väljendatud üldistusi.
- Mitte kustutada seisundi üleminekul s1=>s2 rida tabelist, mis vastab seisundile s1.

Näide:

Põhiolemitüübi *Tellimus* alusel luuakse tabel *Tellimus*:

tellimuse_nr	**kohaletoimetamise_aadress**
1	Narva mnt 5
2	Ale 6
3	Ristiku 51
4	Akadeemia tee 21a
5	Soo 46

Luuakse järgmise struktuuriga tabel nimega *Ootel_tellimus*:

tellimuse_nr	**seisundimuudatuse_aeg**
1	03.03.2019 12:13:15

Luuakse järgmise struktuuriga tabel nimega *Töötlemisel_tellimus*:

tellimuse_nr	**seisundimuudatuse_aeg**
2	14.03.2019 12:02:11

Luuakse järgmise struktuuriga tabel nimega *Välja_saadetud_tellimus*:

tellimuse_nr	**seisundimuudatuse_aeg**
3	14.03.2019 14:02:14

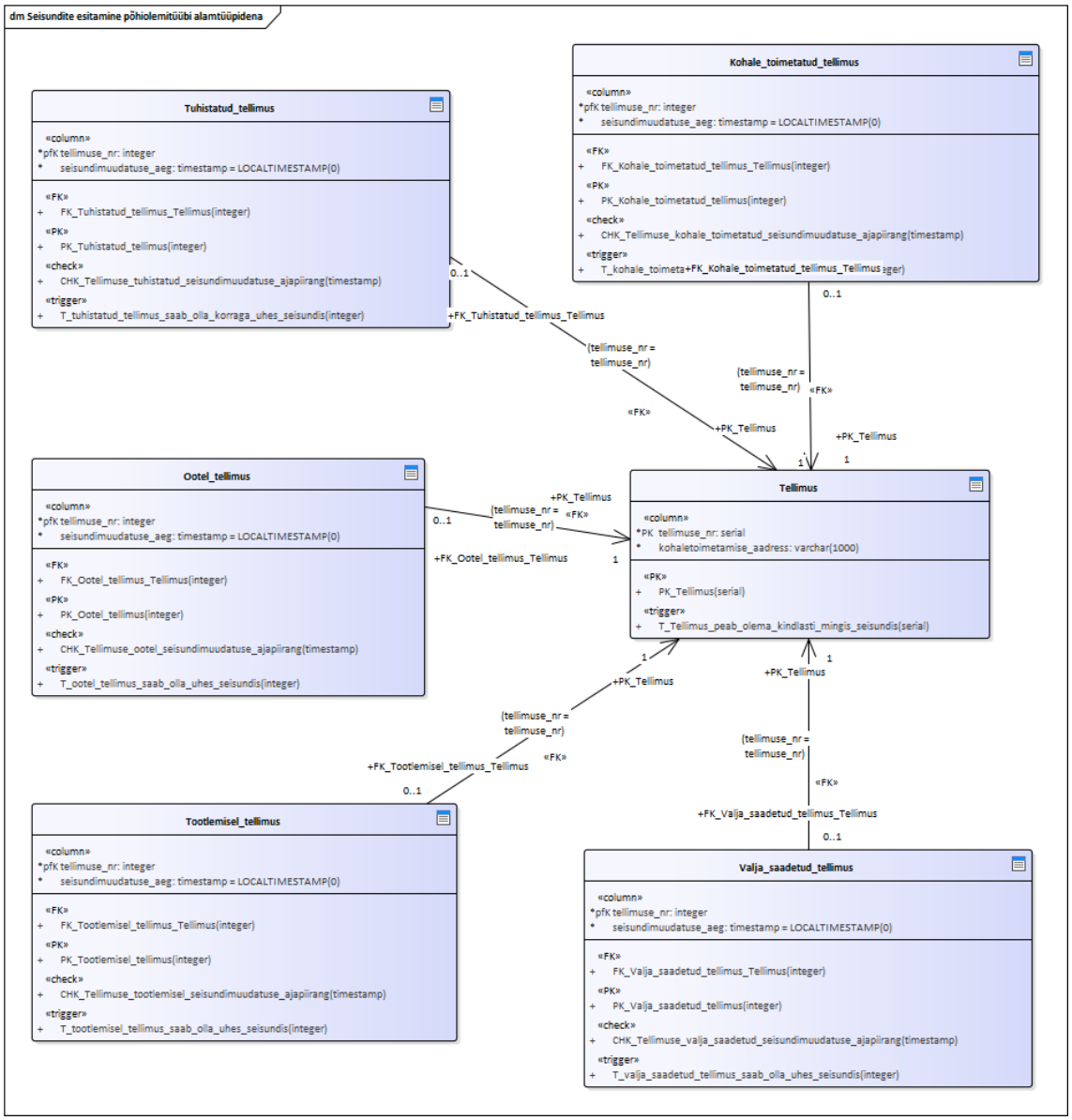
Luuakse järgmise struktuuriga tabel nimega *Kohale_toimetatud_tellimus*:

tellimuse_nr	**seisundimuudatuse_aeg**
4	14.03.2019 14:55:32

Luuakse järgmise struktuuriga tabel nimega *Tühistatud_tellimus*:

tellimuse_nr	**seisundimuudatuse_aeg**
5	12.03.2019 12:00:02

Strateegia: Joonis 22 esitab *Seisundite esitamine põhiolemitüübi alamtüüpidena* disaini realisatsiooni PosgreSQL andmebaasisüsteemis.



Joonis 22. Seisundite esitamine põhiolemitüübi alamtüüpidena disaini andmebaasi diagramm.

Tagamaks, et näite kirjelduses esitatud nõue (1.1) oleks täidetud, loon andmebaasis kitsenduste trigeri tabelile *Tellimus*, mis kontrollib, et olem peab olla kindlasti mingis seisundis (Joonis 23):

```
CREATE OR REPLACE FUNCTION F_Tellimus_peab_olema_kindlasti_mingis_seisundis()
RETURNS TRIGGER AS $$
DECLARE
arv integer;
BEGIN
SELECT INTO arv COUNT(*)
FROM
(SELECT Ootel_tellimus.tellimuse_nr FROM Ootel_tellimus WHERE tellimuse_nr =
NEW.tellimuse_nr
UNION
SELECT Tuhistatud_tellimus.tellimuse_nr FROM Tuhistatud_tellimus WHERE
tellimuse_nr = NEW.tellimuse_nr
UNION
SELECT Tootlemisel_tellimus.tellimuse_nr FROM Tootlemisel_Tellimus WHERE
tellimuse_nr = NEW.tellimuse_nr
UNION
SELECT Valja_saadetud_tellimus.tellimuse_nr FROM Valja_saadetud_tellimus
WHERE tellimuse_nr = NEW.tellimuse_nr
UNION
SELECT Kohale_toimetatud_tellimus.tellimuse_nr FROM
Kohale_toimetatud_tellimus WHERE tellimuse_nr = NEW.tellimuse_nr) a;
IF arv<1 THEN
RAISE EXCEPTION 'Tegevus ebaõnnestus - tellimus peab olema kindlasti mingis
seisundis';
END IF;
RETURN NEW;
END;$$ LANGUAGE plpgsql SECURITY DEFINER
SET search_path = public, pg_temp;

CREATE CONSTRAINT TRIGGER T_Tellimus_peab_olema_kindlasti_mingis_seisundis
AFTER INSERT OR UPDATE OF tellimuse_nr ON tellimus
INITIALLY DEFERRED
FOR EACH ROW
EXECUTE FUNCTION f_Tellimus_peab_olema_kindlasti_mingis_seisundis();
```

Joonis 23. Kitsenduste trigeri loomise laused.

Tagamaks, et näite kirjelduses esitatud nõue (1.2) oleks täidetud, loon andmebaasis trigeri igale seisundi tabelile, mis kontrollib, et iga tellimus saab korraga olla maksimaalselt ühes seisundis (Joonis 24):

```
CREATE OR REPLACE FUNCTION
F_ootel_tellimus_saab_olla_korraga_uhes_seisundis() RETURNS TRIGGER AS $$
DECLARE
arv integer;
t_tellimuse_nr Tellimus.tellimuse_nr%TYPE;
BEGIN
SELECT tellimuse_nr INTO t_tellimuse_nr FROM Tellimus WHERE tellimuse_nr =
NEW.tellimuse_nr FOR UPDATE;
SELECT INTO arv COUNT(*)
FROM (SELECT Tootlemisel_tellimus.tellimuse_nr FROM Tootlemisel_Tellimus
UNION
SELECT Tuhistatud_tellimus.tellimuse_nr FROM Tuhistatud_tellimus
UNION
SELECT Valja_saadetud_tellimus.tellimuse_nr FROM Valja_saadetud_tellimus
UNION
SELECT Kohale_toimetatud_tellimus.tellimuse_nr FROM
Kohale_toimetatud_tellimus) a
WHERE a.tellimuse_nr = NEW.tellimuse_nr;
IF arv>0 THEN
RAISE EXCEPTION 'Tegevus ebaõnnestus - tellimus saab olla korraga ainult ühes
seisundis';
END IF;
RETURN NEW;
END;$$ LANGUAGE plpgsql SECURITY DEFINER
SET search_path = public, pg_temp;

CREATE TRIGGER T_ootel_tellimus_saab_olla_uhes_seisundis ON ootel_tellimus
BEFORE INSERT OR UPDATE ON ootel_tellimus
FOR EACH ROW
EXECUTE FUNCTION F_ootel_tellimus_saab_olla_korraga_uhes_seisundis();
```

Joonis 24. Trigeri loomise laused.

Lisaks kasutan igas tabelis veeru *seisundimuudatuse_aeg* omaduste kirjeldamiseks domeeni [40] (Joonis 9), millega seotud CHECK kitsendus kontrollib, et sisestatav ajaline väärtus oleks suurem kui 31.12.1999 ja väiksem kui 01.01.2200.

Nõuded: Tabel 6 esitab lahendused kuidas olla vastavuses nõuete klassidest tulenevatele tingimustele (vt jaotis 4.1).

Tabel 6. Seisundite esitamine põhiolemitüübi alamtüüpidena lahendused nõuetele.

	Nõude kirjeldus	**Lahendus**
1	Olem peab olema kindlasti mingis seisundis	Selliste kontrollide andmebaasi tasemel deklaratiivseks jõustamiseks puudub tänapäeva SQL-andmebaasisüsteemides võimalus, sest need ei toeta üldiste kitsenduse (ASSERTION) loomist ja alampäringuid CHECK kitsendustes. [54] Lahenduseks on kasutada trigereid, mis tagavad, et iga olemi identifikaator

	Nõude kirjeldus	**Lahendus**
		on registreeritud vähemalt ühele seisundile vastavas tabelis.
	Olem saab olla korraga maksimaalselt ühes seisundis	Nõuab trigerite loomist tagamaks, et iga olemi identifikaator on registreeritud vaid ühele seisundile vastavas tabelis.
	Olem saab olla korraga mitmes seisundis	Selles jaotises esitatud lahendus täidab antud nõuet.
2	Olemi puhul tuleb teada ainult selle hetkeseisundit	Selles jaotises esitatud lahendus täidab antud nõuet.
	Olemi puhul tuleb teada selle seisundite ajalugu	Lahenduseks on muuta disaini nii, et olemi üleminekul seisundist s1 seisundisse s2 ei kustutata rida seisundile s1 vastavast tabelist. Probleem on, et kui sama olem jõuab ma elukaare jooksul samasse seisundisse korduvalt, siis säilib info kõige viimase seisundimuudatuse kohta.
3	Olemi puhul huvitab ainult seisundi omamise fakt	Selles jaotises esitatud lahendus täidab antud nõuet.
	Olemi puhul on vaja lisainfot seisundimuudatuse kohta (millal see seisund omandati, kes selle muudatuse algatas jne)	Selleks tuleb luua vaid soovitud lisainfot talletavad veerud seisundile vastavas tabelis.
4	Lubatavate seisundimuudatuste kontroll ilma olekutabelita	Seisundimuudatuste ja ülemineku reeglite kirjeldamiseks ning kontrollimiseks tuleb luua trigerid, mis kontrollivad uue rea lisamisel kõigi tabelite üleselt, et kas see on lubatud tegevus või mitte.
	Lubatavate seisundimuudatuste kontroll olekutabeliga	Olekutabelis on vaja viidata seisundite registreerimiseks kasutatavate tabelite nimedele.
5	Andmebaasis ei ole vaja säilitada lisainfot seisundite kohta	Selles jaotises esitatud lahendus täidab antud nõuet.
	Andmebaasis on vaja säilitada lisainfot seisundite kohta (näiteks nimetus erinevates keeltes, kehtivus, kommentaar)	Säilitamaks lisainfot seisundite kohta tuleks luua uus tabel. Üks võimalus oleks see tabel ülesehitada nii, et seal oleksid erinevate parameetrite väärtused. Näiteks võib luua tabelisse järgnevad veerud: *seisund_nimi*, *parameeter* ja *väärtus*. (vt jaotis 4.4) Veel üks lahendus on luua seisundiklassifikaatori tabel (vt jaotis 4.3) puhtalt metaandmete hoidmise jaoks.
6	Tekib uus võimalik seisund, milles olem võib viibida	Tuleb luua uus tabel kasutades selleks CREATE TABLE [55] lause.

	Nõude kirjeldus	**Lahendus**
	Kaob seisund, milles olem võib viibida	Tuleb kustutada tabel kasutades selleks DROP TABLE [56] lauset.

4.8 Seisundite tuletamine olemite ja seoste kohta registreeritud andmetest

Nimi inglise keeles: Derive status values from the data about entitis and their relationships

Alternatiivsed ingliskeelsed nimed: No special means to register status

Allikad:

- „Dynamically update „Status“ column after „X“ amount of time?“ [57]
- „SQL set column with derived value“ [58]
- „Specify Computed Columns in a Table“ [59]

Lahendus: Eraldi ühtlustatud mehanismi seisundi registreerimiseks ei kasutata. Põhiolemi hetkeseisund tuleb tuletada põhiolemi, sellega seotud olemite ja nende seoste kohta registreeritud andmetest. Andmebaasi mõttes võivad need andmed olla erinevates tabelites. Nende andmete esitamiseks võib kuid ei pruugi olla kasutatud temporaalseid veege. Näiteks sellest, et tellimus on täidetud saan aru kui tellimusega on seotud kohaletoimetamine, millele on määratud kohaletoimetamise aeg.

Eelised:

- Andmeid seisundite kohta ei dubleeria, mistõttu ei saa registreerida vastuolulisi väiteid. Näiteks pole võimalik olukord, et seisundiklassifikaatori alusel on tellimus kohaletoimetamata, samas kui andmebaasis on registreeritud kohaletoimetamise aeg.
- Seisundi kohta info andmebaasis talletamiseks pole vaja spetsiaalseid tabeleid ja veerge (nt klassifikaatori tabel, vektorkoodi sisaldav veerg), mis lihtsustab mingil määral andmebaasi struktuuri ning vähendab andmebaasi mahtu.
- Ärireeglitest tulenevalt võib olla seisundi muudatuseks vajalik operatsioon väga lihtne ja kiire, sest antud disainis pole loodud eraldi veergu või tabelit seisundi väärtuste hoidmiseks. Sama võib öelda ka uue seisundi lisamise või olemasoleva

eemaldamise kohta, sest võimalik, et tuleb teha muudatus ainult ärireeglites. (vt peatükk 6)

Puudused:

- Süsteemianalüüs peab selgeks tegema, milliste olemitüübi kohta registreeritud andmetele vastab milline seisund. Tulemused saab dokumenteerida ärireeglitena. Andmebaasi inimkasutajal on andmetel peale vaadates keeruline mõista, millises seisundis on hetkel mingi põhiolem. Selleks tuleb lugeda ärireeglite kirjeldust erinevate seisundite tuletamise kohta – milliste andmete registreerituses võib järeldada millise olemi seisundi.

- Võib juhtuda olukord, et olemi hetkeseisund on registreerimata – on rida põhiolemitüübile vastavas tabelis, kuid pole ühtegi vastavat rida seisunditele vastavates tabelites. Turul olevad SQL-andmebaasisüsteemid ei toeta paraku praegu üldiste kitsenduste (ASSERTION) objekte ja alampäringuid CHECK kitsendustes, mis võimaldaks seda andmebaasis deklareeritud kitsenduste abil vältida.

- Keeruline päring (PostgreSQL 11 näitel), et leida iga põhiolemi kohta selle hetkeseisund. Sama käib ka koondandmete leidmise päringu kohta. Uue võimaliku seisundi lisandumisel tuleb neid päringuid täiendada. (vt peatükk 6)

- Kuna nii hetkeseisundi kui ka seisunditele vastavate koondandmete leidmiseks tuleb andmeid lugeda mitmest tabelist, siis see suurendab päringute täitmiseks kuluvat aega. (vt peatükk 6)

- Ei võimalda liigitada seisundeid erinevatesse kategooriatesse.

- Andmebaasi kasutajal puudub ülevaade, millised on kõikvõimalikud seisundid, milles mingit tüüpi olem võib viibida.

- Raske realiseeria olekutabelit, sest andmed seisundite kohta on erinevate tabelite erinevates veergudes. Olekutabelis tuleb viidata nendele tabelitele ja veergudele.

- Keeruline päring, et leida seisundimuudatuste ajalugu ja näidata selle kronoloogiat. Uue võimaliku seisundi lisandumisel tuleb seda päringut täiendada.

- Seisundite nimed tuleb kodeerida sisse andmebaasirakendusse ja päringutesse. Nime muutmisel tuleb muudatusi teha mitmes kohas. (vt jaotis 5.6.1)
- Antud disaini puhul on (PostgreSQL 11 näitel) andmemaht suurem teistest töös esitatud disainidest. (vt peatükk 6)

Variatsioonid:

- Temporaalsete veergude kasutamine, milles olevate väärtuste põhjal tuletatakse sarnaselt selles jaotises esitatud disainile info konkreetse olemi seisundi kohta. Antud lahendus on välja toodud jaotises 4.4.
- Erinevatele seisunditele vastavad erinevad tabelid nagu kirjeldatakse jaotises 4.7.
- Päringute, mis tagastavad vastavalt ärireeglitele põhiolemite hetkeseisundi, realiseerimine vaadetena. [57] See võimaldab hetkeseisundi päringut lihtsamalt kirja panna, kuid ei aita parandada nende päringute töökiirust [60].

Näide:

Põhiolemitüübi *Tellimus* alusel luuakse tabel *Tellimus*:

tellimuse_nr	**esitamise_aeg**	**kohaletoimetamise_aadress**
1	03.03.2019 12:13:15	Narva mnt 5
2	08.03.2019 16:40:11	Ale 6
3	10.03.2019 09:15:27	Ristiku 51
4	10.03.2019 15:32:01	Akadeemia tee 21a
5	12.03.2019 11:59:59	Soo 46

Luuakse järgmise struktuuriga tabel *Arve:*

tellimuse_nr	**tasumise_aeg**
1	NULL
2	14.03.2019 12:02:11
3	12.03.2019 14:43:11
4	11.03.2019 09:00:01

tellimuse_nr	tasumise_aeg
5	NULL

Luuakse järgmise struktuuriga tabel *Tellimuse_tühistamine*:

tellimuse_nr	tühistamise_aeg
5	12.03.2019 12:00:02

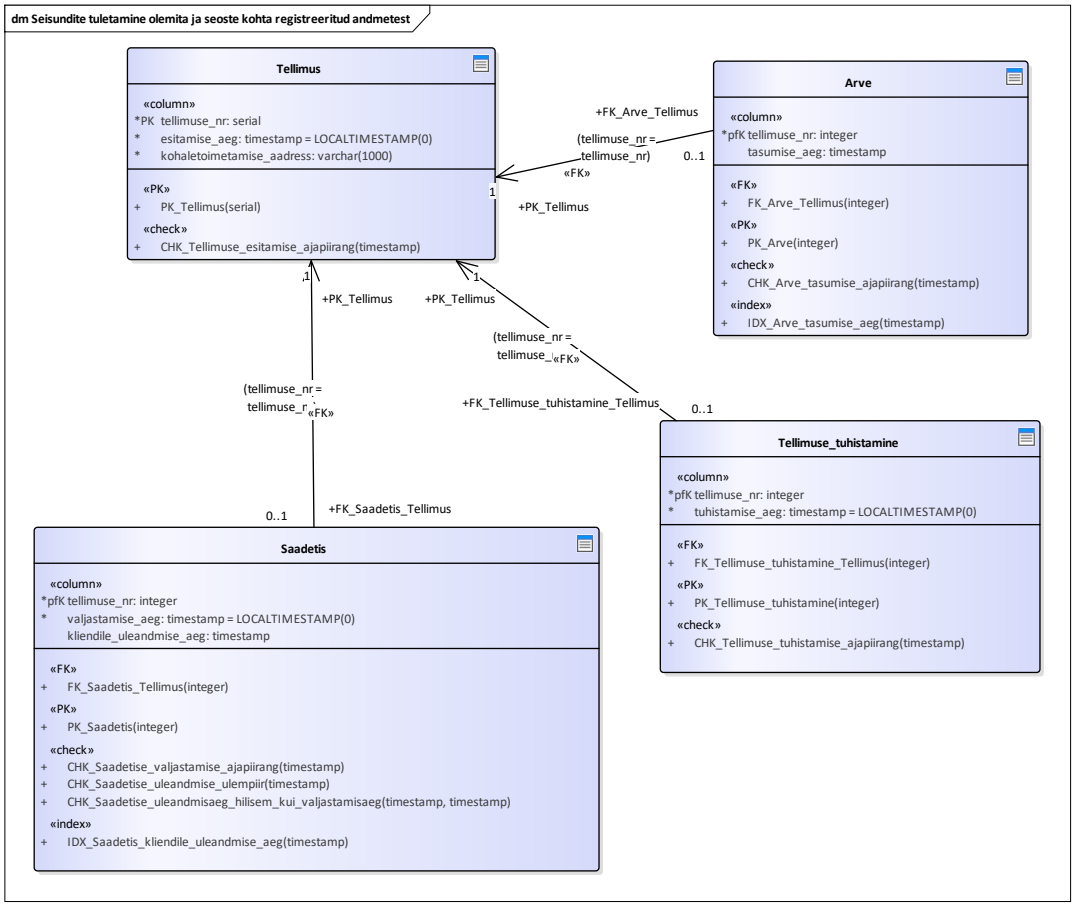
Luuakse järgmise struktuuriga tabel *Saadetis*:

tellimuse_nr	väljastamise_aeg	kliendile_üleandmise_aeg
3	14.03.2019 14:02:14	NULL
4	13.03.2019 17:04:11	14.03.2019 14:55:32

Lisaks kirjeldatakse järgnevad ärireeglid:

1. Tellimus on ootel, kui (tellimusega pole seotud arvet või tellimusega seotud arvel puudub tasumise_aeg) ja tellimusega pole seotud tühistamist.
2. Tellimus on tühistatud, kui tellimusega on seotud tühistamine.
3. Tellimus on töötlemisel, kui tellimusega on seotud tasumise ajaga arve ja sellega pole seotud saadetist.
4. Tellimus on väljastamisel, kui tellimusega on seotud saadetis, millel kliendile üleandmise aeg puudub.
5. Tellimus on kohale toimetatud, kui tellimusega on seotud saadetis, millel on kliendile üleandmise aeg määratud.

Strateegia: Joonis 25 esitab *Seisundite tuletamine olemite ja seoste kohta registreeritud andmetest* disaini realisatsiooni PostgreSQL andmebaasisüsteemis.



Joonis 25. Seisundite tuletamine olemite ja seoste kohta registreeritud andmetest disaini andmebaasi diagramm.

Kiirendamaks otsinguid loon osalise indeksi igale mittekohustuslikule sündmuse veerule [46] (Joonis 26):

```
CREATE INDEX IDX_Arve_tasumise_aeg ON Arve(tasumise_aeg) WHERE tasumise_aeg
IS NOT NULL;
CREATE INDEX IDX_Saadetis_kliendile_uleandmise_aeg ON
Saadetis(kliendile_uleandmise_aeg) WHERE kliendile_uleandmise_aeg IS NOT
NULL;
```

Joonis 26. Osaliste indeksite loomise laused.

Lisaks kasutan igas tabelis ajalisi väärtusi hoidvate veergude omaduste kirjeldamiseks domeeni [28] (Joonis 9), millega seotud CHECK kitsendus kontrollib, et sisestatav ajaline väärtus oleks suurem kui 31.12.1999 ja väiksem kui 01.01.2200:

Nõuded: Tabel 7 esitab lahendused kuidas olla vastavuses nõuete klassidest tulenevatele tingimustele (vt jaotis 4.1).

Tabel 7. Seisundite tuletamine olemite ja seoste kohta registeeritud andmetest lahendused nõuetele.

	Nõude kirjeldus	**Lahendus**
1	Olem peab olema kindlasti mingis seisundis	Selliste kontrollide andmebaasi tasemel deklaratiivseks jõustamiseks puudub tänapäeva SQL-andmebaasisüsteemides võimalus, sest need ei toeta üldiste kitsenduse (ASSERTION) loomist ja alampäringuid CHECK kitsendustes. [54] Lahenduseks on kasutada trigereid.
	Olem saab olla korraga maksimaalselt ühes seisundis	Seda, millises seisundis konkreetne olem parajasti viibib, peavad interpreteerima andmebaasi kasutavad programmid kasutades andmebaasis olevaid andmeid. Kontroll selle kohta, et andmed seisundi kohta vastaksid ärireeglitele eeldab suure tõenäosusega andmete lugemist erinevatest tabelitest. Näiteks võib kontrollida, et kui tellimus on täidetud, siis peab olema registreeritud ka selle tellimuse alusel moodustatud saadetis. Selliste kontrollide andmebaasi tasemel deklaratiivseks jõustamiseks puudub tänapäeva SQL-andmebaasisüsteemides võimalus. Lahenduseks on kasutada trigereid.
	Olem saab olla korraga mitmes seisundis	Selles jaotises esitatud lahendus täidab antud nõuet.
2	Olemi puhul tuleb teada ainult selle hetkeseisundit	Tuleb teha päring üle erinevate tabelite. Võimalik lahendus kasutab ühendi leidmise (UNION) päringut, kus iga võimaliku seisundi kohta on alampäring, millega leitakse olemid, mis hetkel selles seisundis viibivad + iga olemi kohta sellesse seisundisse minemise aeg. Lisakeerukusena tuleb igas alampäringus kontrollida, et olem ei viibi parjasti mõnes teises seisundis.
	Olemi puhul tuleb teada selle seisundite ajalugu	Eeldab, et erinevates tabelites põhiolemi kohta registreeritud sündmuste juures on registreeritud ka sündmuse toimumise aeg. Võimalik lahendus kasutab ühendi leidmise (UNION) päringut, kus iga võimaliku seisundi kohta on alampäring, millega leitakse olemid, mis kunagi on selles seisundis viibinud + iga olemi kohta sellesse seisundisse minemise aeg.
3	Olemi puhul huvitab ainult seisundi omamise fakt	Selles jaotises esitatud lahendus täidab antud nõuet.
	Olemi puhul on vaja lisainfot seisundimuudatuse kohta (millal see seisund omandati, kes selle muudatuse algatas jne)	Nende andmete kogumiseks vajalikud veerud tuleb vastavalt vajadusele lisada tabelitesse, milles registreeritakse andmed mingi konkreetse seisundi omamise kohta. Näiteks tabelis *Tühistamine* on veerud *põhjus* ja *tühistaja*. Viimane nendest on välisvõtme veerg,

	Nõude kirjeldus	**Lahendus**
		milles iga väärtus viitab tühistamise korralduse andnud töötajale, kelle andmed on tabelis *Töötaja*.
4	Lubatavate seisundimuudatuste kontroll ilma olekutabelita	Seda võivad teha andmete registreerimiseks mõeldud andmebaasiserveris talletatud rutiinid. Näiteks enne kui registreeritakse tellimuse kohaletoimetamine kontrollitakse, kas tellimus on välja saadetud.
	Lubatavate seisundimuudatuste kontroll olekutabeliga	Raske realiseeria olekutabelit, sest andmed seisundite kohta on erinevate tabelite erinevates veergudes. Olekutabelis tuleb viidata nendele tabelitele ja veergudele.
5	Andmebaasis ei ole vaja säilitada lisainfot seisundite kohta	Selles jaotises esitatud lahendus täidab antud nõuet.
	Andmebaasis on vaja säilitada lisainfot seisundite kohta (näiteks nimetus erinevates keeltes, kehtivus, kommentaar)	Säilitamaks lisainfot seisundite kohta tuleks luua uus tabel. Üks võimalus oleks see tabel ülesehitada nii, et seal oleksid erinevate parameetrite väärtused. Näiteks võib luua tabelisse järgnevad veerud: *seisund_nimi*, *parameeter* ja *väärtus*. (vt jaotis 4.4) Veel üks lahendus on luua seisundiklassifikaatori tabel (vt jaotis 4.3) puhtalt metaandmete hoidmise jaoks.
6	Tekib uus võimalik seisund, milles olem võib viibida	Tuleb luua uusi tabeleid või veerge. Lisaks tuleb teha täiendusi nii ärireeglites kui ka vastavalt sellele päringutes, mis realiseerivad seisundi kindlakstegemist.
	Kaob seisund, milles olem võib viibida	Selleks tuleb teha muudatusi nii ärireeglites kui ka vastavalt sellele päringutes, mis realiseerivad seisundi kindlaks tegemist. Kui koos äriprotsesside ümberkorraldamisega otsustatakse loobuda mingite andmete kogumisest, siis tuleb andmebaasist eemaldada tabeleid või veerge ja see omakorda viib infokaoni.

5 Eksperiment

Selles peatükis kirjeldan detailselt käesolevas töös läbiviidavat eksperimenti võrdlemaks omavahel peatükis 4 kirja pandud disaine.

5.1 Eksperimendi eesmärk

Käesolevasse magistritöösse allikmaterjali otsides selgus, et leidub küllaltki palju kirjandust (vt peatükk 3), mis puudutab põhiolemite seisundite kohta andmebaasis andmete hoidmist. Siiski ei leidu allikat, mis koondaks erinevaid lahendusi ja võdleks neid omavahel erinevatest aspektides, nagu näiteks päringute ja andmemuudatuste kiirus ja keerukus. Seetõttu teen töö osana praktilise eksperimendi, et uurida nende leitud lahenduste headust. Seega annab minu töö infosüsteemi arendajatele võimaluse toetada oma valikute põhjendamisel mitte ainult disainide kataloogis välja toodud teooriale, vaid ka praktilistele katsetustele. Selleks viin kataloogis kirjeldatud disainide võrdlemiseks läbi järgnevad testid:

- Mõõdan tabelite ja nendele loodud indeksite salvestusruumi kasutust erinevate andmehulkade korral.
- Mõõdan nii konkreetse olemi seisundi otsimise, olemite nimekirja koostamise kui ka koondandmete päringu täitmise kiirust erinevate andmehulkade korral.
- Mõõdan konkreetse olemi seisundi muutmise kiirust erinevate andmehulkade korral.
- Mõõdan eelnimetatud andmekäitluse lausete keerukust.
- Mõõdan uue seisundi lisamise ja olemasoleva seisundi eemaldamiseks vajalike lausete keerukust. Sellised operatsioonid on vajalikud seoses muudatustega äriprotsessides ning sellest tulenevate muudatustega infosüsteemi tarkvaras.
- Mõõdan erinevate disainide realiseerimiseks vajalike andmebaasikeele lausete keerukust.

Esitan mõõtmiste tulemused peatükis 6. Analüüsin mõõtmise tulemusi ning täiendan tehtud järelduste (vt peatükk 7) põhjal disainide kataloogis esitatud eeliseid ja puuduseid.

5.2 Eksperimendi kirjeldus

Eksperimendis loon disainide kataloogis kirjeldatud näite (vt jaotis 4.1) alusel iga kataloogis kirjeldatud disaini kohta samas andmebaasis eraldi skeemid. Nimetan need järgnevalt:

- *seisundiklassifikaator* (1)
- *temporaalsed_veerud* (2)
- *toevaartustuupi_veerud* (3)
- *vektorkodeerimine* (4)
- *pohiolemituubi_alamtuupidena* (5)
- *seisundite_tuletamine* (6)

Selleks, et eksperimendi kirjeldus ja tulemused oleksid paremini jälgitavad, kasutatan disaini nimetuste asemel töö eksperimendi osas disainidele viidates numbreid, mis on esitatud ülalolevas listis iga skeemi nime järgi.

Võrdlen eksperimendis disaine põhiolemitüübi *Tellimus* näitel (vt jaotis 4.1), mida kasutasin ka läbiva näitena kataloogis (vt peatükk 4)

Selleks, et eksperimendi tulemused oleks võrreldavad, kasutan kõikides andmebaasidisainides andmeid samasuguste omadustega ning hetkeseisundiga tellimuste kohta ning seejuures võtan arvesse ka erinevaid andmehulki. Testandmete genereerimise protsessist kirjutan lähemalt jaotises 5.5. Saamaks teada tabeli salvestusruumi kasutust baitides kasutan PostgreSQL'i süsteemi-defineeritud funktsiooni *pg_total_relation_size(regclass)* [61]. Antud funktsioon arvutab ja tagastab tabeli, tabelitega seotud indeksite ning TOAST (*The Oversized-Attribute Storage Technique*) andmete andmemahu baitides [23]. Selleks, et mõõtmistulemused oleks paremini jälgitavad, kasutatan lisaks veel *pg_size_pretty()* funktsiooni, mis teisendab tulemused asjakohasesse suurusjärku (bytes, kB, MB, GB või TB) [61]. Summeerin saadud tulemused iga disaini ja sellele vastava andmehulga lõikes. Kõik mõõtmised on sellised, et mida väiksem tulemus, seda parem.

Päringute ja andmemuudatus operatsioonide ning andmebaasi realiseerimiseks vajalike lausete keerukuse hindamiseks kasutan koodiridade arvu metoodikat (*Source Lines of Code, SLOC*). See on tarkvaramõõdik, mida kasutatakse arvutiprogrammi suuruse mõõtmiseks, loendades lähtekoodi ridade arvu. [62] Leidmaks füüsilist koodiridade arvu, mis tähendab, et koodiridade hulka ei loeta kommentaare ega tühje ridu, kasutan tasuta programmi LocMetrics [63]. Selleks, et oleks võimalik võrrelda kõiki töös kasutatavaid SQL lauseid, vormindan need järgmiste reeglite kohaselt:

- Peamised võtmesõnad algavad uuelt realt (SELECT, FROM, WHERE, ORDER BY, ALTER, UPDATE, DELETE, UNION, JOIN, CREATE).
- SELECT, UPDATE ja ALTER klauslis algab iga järgneva veeru või muudatuse nimetus uuelt realt.
- Iga tabeli nimi FROM (ka JOIN ning UNION) klausis algab uuelt realt.
- Iga WHERE alamtingimus algab uuelt realt.
- CREATE TABLE klauslis algab iga loodava veeru nimetus uuelt realt (ka esimese veeru nimetus).

Selliselt formaaditud SELECT lause näide on *Lisas 2*.

Päringute ja andmemuudatus operatsioonide täitmiskiiruse mõõtmiseks kasutan PostgreSQL'i lauset EXPLAIN ANALYZE [64]. Antud lause tagastab andmekäitluskeele lause täitmisplaani, plaani koostamiseks kulunud aja ning plaani täitmiseks kulunud aja. Andmemuudatuste korral näidatakse ka trigerite täitmiseks kulunud aega. Lõpliku täitmise aja leidmiseks liidan kõik need ajad kokku. Lisaks käivitan kõiki päringuid ja andmemuudatus operatsioone viis korda tagamaks, et ühekordsed riist- või tarkvara kõrvalekalded ning süsteemi „soojenemine" (lause esimese täitmise järel täitmisplaani koostamine ning meeldejätmine, andmete muutmällu lugemine), mis võivad mõjutada lausete täitmise kiirust, ei mõjutaks liigselt tulemusi. [23] Saadud tulemuste põhjal arvutan geomeetrilise keskmise, milleks kasutan Exceli funktsiooni *GEOMEAN()* [65]. Eelistan geomeetrilist keskmist aritmeetilisele keskmisele, sest lause korduval täitmisel jätab süsteem üldjuhul esimese korra järel meelde selle täitmisplaani ning lause täitmiseks loetakse andmed muutmällu [15]. Seega on mõõtmiste tulemused üksteisest sõltuvad [66].

Kõik testid, mõõtmaks andmemuudatus operatsioonide ja päringute täitmise kiirust, viin läbi kolme erineva andmehulgaga :

- andmehulk 1 – 500 000 tellimust,
- andmehulk 2 – 1 miljon tellimust,
- andmehulk 3 – 2 miljon tellimust.

Selle eesmärgiks on hinnata kas täitmisaja kasvamise e töökiiruse vähenemise ja andmehulga suurenemise vahel on lineaarne sõltuvus või mitte [67]. Seetõttu leian nende mõõtmistulemuste alusel Pearsoni korrelatsiooni koefitsendi [68], mille arvutamiseks kasutan Excelis olevat funktsiooni *PEARSON()* [69]. Seetõttu loon ka andmebaasis iga disaini kohta kolm eraldi skeemi – üks iga andmehulga kohta.

Kogu töös kasutatav SQL-kood on lisatud formaaditud kujul GitHub'i, mis on internetis kättesaadav aadressil https://github.com/susannapeek1/seisundid_SQL. [70]

5.3 Kasutatavad andmebaasisüsteemid

Valisin käesolevas töös kasutatavaks andmebaasisüsteemiks PostgreSQL 11. Valikul lähtusin süsteemi võimalusterohkusest, enda eelnevatest teadmistest ja selle süsteemi populaarsusest. 2019. aasta mais oli see andmebaasisüsteemide populaarsuse indeksis neljandal kohal [3]. PostgreSQL korral tasub mainimist ka see, et tegemist on avatud lähtekoodiga ja tasuta kasutatava andmebaasisüsteemiga, mistõttu selle tundmine ja kasutamine võib aidata olulisel määral kulusi kokku hoida [15].

Eksperimendi läbiviimiseks kasutan Tallinna Tehnikaülikooli serverit *apex.ttu.ee*, kus on PostgreSQL (11) andmebaasisüsteem. Tehnilised andmed sellele on järgmised: virtuaalmasin QEMU Virtual CPU version, 811 GB HDD, 40 GB RAM, 16 virtuaalset CPU'd, CentOS 6.10.

5.4 Andmebaasi realiseerimine

Realiseerin iga kataloogis esitatud disaini alusel tabelite ja kitsenduste loomiseks vajalikud CREATE laused. Seejärel käivitan need andmebaasis nimega *seisundite_esitamine*.

Kõikide skeemide loomise kood on küllaltki mahukas, seega on need lisatud GitHubi salve, mille aadressiks on https://github.com/susannapeek1/seisundid_SQL. Täpsemalt leiab need iga disaini alamkataloogis olevast kaustast, mille nimi sisaldab tähte „C“.

PostgreSQL loob primaarvõtmete ja unikaalsuse kitsenduste alusel automaatselt indeksid [15], mistõttu ei sisalda loodavate andmebaaside SQL kood vastavaid ridu.

5.5 Testandmed

Selleks, et viia läbi eksperimente, mis võrdlevad kõigi kataloogis esitatud disainide andmemahtusid, andmemuudatuste ja päringute täitmiskiirust, on vajalik genereerida testandmed. Selgitamaks välja disaini omaduste sõltuvust andmehulga muutustest teen kõiki mõõtmisi kolme erineva andmehulgaga [67]. Valisin antud töös nendeks hulkadeks 500 000 (1), 1 000 000 (2) ja 2 000 000 (3) rida (iga järgmine on eelmisest kaks korda suurem). Tegin ridade arvu valiku, et näha kuidas mõjutavad just suured andmehulgad testitavate operatsioonide käitumist iga disaini korral. Seeläbi saavad arendajad tugineda selle magistritöö tulemusele ka suure prognoositava andmehulgaga süsteemide puhul. Sellise hulga tellimuste hoidmine andmebaasis on mõne suurema e-kaubanduse ettevõte puhul täiesti reaalne. Lisaks, et testide tulemusi oleks võimalik omavahel võrrelda, kasutatan iga disaini korral andmeid samasuguste omadustega ning hetkeseisundiga tellimuste kohta. Vastavalt eksperimendis testitavatele operatsioonidele (vt jaotis 5.6) ei loo ma ühtegi seisundis tühistatud tellimust.

Testandmete genereerimiseks kasutan PostgreSQL funktsioone *random(), floor()* [71] ja *generate_series()* [72]. Tagamaks, et kõigis skeemides on andmed ühesuguste tellimuste kohta, genereerin need ainult *seisundiklassifikaator* kõige suurema andmehulgaga skeemi tabeli *Tellimus* jaoks. Enne tellimuste registreerimist väärtustan seisundiklassifikaatori [41] (Joonis 27):

```
INSERT INTO tellimuse_seisundi_liik (tellimuse_seisundi_liik_kood, nimetus)
VALUES (1,'Ootel'), (2,'Töötlemisel'), (3,'Välja saadetud'), (4,'Kohale
toimetatud'), (5,'Tühistatud');
```

Joonis 27. Klassifikaatori väärtuste registreerimise lause.

Tellimuste andmete genereerimiseks kasutan lauset Joonis 28. Süsteemi-defineeritud funktsioon *generate_series()* võimaldab määrata soovitava ridade arvu. Avaldis *floor(random()*(5-1+1) + 1)::int* genereerib juhusliku täisarvu vahemikus 1 ja 4 (otspunktid kaasa arvatud). Kuna lõin tabeli *Tellimus* veerule *seisundimuudatuse_aeg*

reataseme trigeri, mis käivitub uue tellimuse lisandumisel või olemasoleva tellimuse seisundimuutuse korral, siis ei ole selles SQL koodis avaldist antud veergu väärtuste genereerimiseks:

```
INSERT INTO tellimus (kohaletoimetamise_aadress,tellimuse_seisundi_liik_kood)
SELECT
md5(random()::text),
floor(random()*(5-1+1) + 1)::int
FROM generate_series(1,2000000);
```

Joonis 28. Tellimuste andmete genereerimise lause.

Suurima andmehulgaga *seisundiklassifikaator* skeemist kannan andmed INSERT INTO lauseid kasutades edasi teistesse skeemidesse [41]. Selleks, et ka *toevaartustuupi_veerud* ja *vektorkodeerimine* skeemide puhul oleks veeru *seisundimuudatuse_aeg* väärtused identsed genereeritavate andmetega, luuakse eelnevalt kirjeldatud triger alles peale andmete tabelitesse importimist. Kuna skeemi *seisundite_tuletamine* tabelid sisaldavad olemite kohta rohkem infot, siis järgitakse selle tabelitesse andmete importimisel ühtset põhimõtet. Nimelt oletatan, et tellimuse seisund on muutunud iga päev ehk kui see on näiteks seisundis *Kohale toimetatud*, siis selle *kliendile_uleandmise_aeg* tabelis *Saadetis* on võrdne *seisundimuudatuse_aeg* väärtusega skeemis *seisundiklassifikaator*. Samas tabelis oleva veeru *valjastamise_aeg* väärtus leitakse lahutades *seisundimuudatuse_aeg* väljas olevast väärtusest üks päev. Lisaks täidetakse ka tabeli *Arve* veerg *tasumise_aeg* lahutades *seisundimuudatuse_aeg* väljas olevast väärtusest kaks päeva ja tabelis *Tellimus* olev veerg *esitamise_aeg* lahutades *seisundimuudatuse_aeg* väljas olevast väärtusest kolm päeva. Sarnast lahutamise ideed kasutatakse ka kõigi selliste tellimuste andmete importimisel, mis on kas seisundis *Töötlemisel*, *Välja saadetud* või *Kohale toimetatud*. Seisundis *Ootel* tellimuste jaoks sellist arvutust teha pole vaja, sest tellimus on selles seisundis kui *Tellimus* tabelis olev veerg *esitamise_aeg* (mis on võrdne *seisundiklassifikaator* veerus *seisundimuudatuse_aeg* oleva väärtusega) on täidetud. Tellimusel võib kuid ei pruugi olla seotud arve. Seetõttu kasutan LIMIT [73] klauslit, et lisada osa juhuslikult leitud andmetest ka tabelisse *Arve*.

Väiksemate mahtudega skeemide jaoks kustutan peale andmete lisamist vajalik hulk ridu tabelitest kasutades selleks DELETE [42] lauset. Kui kõikidesse skeemidesse on andmed edukalt kantud, siis käivitan lause VACUUM ANALYZE, mis eemaldab loogiliselt tabelist kustutatud read ka füüsilisest salvestusest ning värskendab statistikat [74].

Kogu töös kasutatav SQL-kood testandmete genereerimiseks on leitav GitHub'ist aadressilt https://github.com/susannapeek1/seisundid_SQL. Täpsemalt on need leitavad alamkaustast nimega *Testandmed*. [70]

5.6 Eksperimendis testitavad operatsioonid

Eksperimendi käigus viin läbi andmete lugemise ja muutmise operatsioone, et katsetada nende töökiirust ja määrata vastava koodi keerukus. Nimetatud operatsioonid on valitud nii, et neeed oleksid iseloomulikud operatiivandmete andmebaasile ja vajaliku seda kasutavale onlain tehingutöötluse tarkvarale. Näiteks teatud seisundis tellimuste (näiteks töötlemisel tellimused) nimekiri võib olla aluseks nimekirjavormile, kust saab valida tellimusi, mille seisundit soovitakse muuta (näiteks väljasaadetuks).

Samuti viin läbi operatsioonid uue seisundi kasutuselevõtuks ja seisundi eemaldamiseks, et hinnata selleks vajalike lausete koodi keerukust. PostgreSQL võimaldab koondada andmekirjelduskeele lauseid transaktsiooni, kus need täidetakse loogilise tervikuna – kas täidetakse kõik või jäävad kõik täitmata. Seega kõigi mitu lauset hõlmavate lahenduste korral on laused koondatud transaktsiooni plokki.

5.6.1 Andmete lugemine

Järgnevalt esitan kõikide eksperimendis kasutatavate andmete otsimise ülesannete kirjelduse. Lisaks toon välja ülesande lahendamiseks mõeldud SQL laused igale kataloogis esitatud disaini kohta.

Teatavasti saab SQLis ühte ja sama ülesannet lahendada paljudel erinevatel viisidel [75]. Oma eksperimendis kasutasin iga ülesande puhul ühte lahendust. Töö tulemuse mõttes on see piirang, sest näiteks töökiiruse mõttes võib leidud paremaid lahendusi kui katsetan antud töös. Parema töökiiruse huvides lähtusin lahenduste valikul põhimõttest, et võimalusel mitte sama tabeli poole lauses mitu korda pöörduda ning mitte lasta süsteemil lugeda lause täitmiseks liigselt andmeid. Samuti vältisin lahendustes NOT IN + mittekorreleeruv alampäring kasutamist ja eelistasin selle asemel NOT EXISTS + korreleeruv alampäring kasutamist. Selle põhjuseks on NOT IN halb töökiirus PostgreSQLis [76]. WITH klausli ja ühise tabeli avaldise kasutamisel katsetasin ka alternatiive, veendumaks, et see ei põhjusta töökiiruse kadu nagu kirjeldab [77].

Esimese päringuga (S1) leian kõik ühes kindlas seisundis tellimused. Päringu (Tabel 8) tulemusel tagastatakse kasvavas järjekorras kõik seisundis *Töötlemisel* olevate tellimuste numbrid.

Tabel 8. Andmete lugemiseks S1 kasutatavad päringud.

Disain	**S1**
1	`SELECT Tellimus.tellimuse_nr FROM Tellimus WHERE Tellimus.tellimuse_seisundi_liik_kood = 2 ORDER BY Tellimus.tellimuse_nr;`
2	`SELECT Tellimus.tellimuse_nr FROM Tellimus WHERE Tellimus.tootlemisel_aeg IS NOT NULL ORDER BY Tellimus.tellimuse_nr;`
3	`SELECT Tellimus.tellimuse_nr FROM Tellimus WHERE Tellimus.on_tootlemisel = TRUE ORDER BY Tellimus.tellimuse_nr;`
4	`SELECT Tellimus.tellimuse_nr FROM Tellimus WHERE F_tellimuse_seisundi_dekodeerimine (Tellimus.seisund) = 'Töötlemisel' ORDER BY Tellimus.tellimuse_nr;`
5	`SELECT Tootlemisel_tellimus.tellimuse_nr FROM Tootlemisel_tellimus ORDER BY Tootlemisel_tellimus.tellimuse_nr;`
6	`SELECT Tellimus.tellimuse_nr FROM Tellimus INNER JOIN Arve ON Tellimus.tellimuse_nr = Arve.tellimuse_nr WHERE Arve.tasumise_aeg IS NOT NULL AND NOT EXISTS (SELECT * FROM Saadetis WHERE Tellimus.tellimuse_nr = Saadetis.tellimuse_nr) ORDER BY Tellimus.tellimuse_nr;`

Teise päringuga (S2) leian kõik tellimused, mis pole ühes kindlas seisundis. Päringu (Tabel 9) tulemusel tagastatakse kõigi selliste tellimuste numbrid ja nende hetkeseisundi nimetus, mis ei ole seisundis *Ootel*. Tulemus sorteeritakse tellimuse numbrite alusel kasvavalt.

Tabel 9. Andmete lugemiseks S2 kasutatavad päringud.

Disain	**S2**
1	`SELECT Tellimus.tellimuse_nr,Tellimuse_seisundi_liik.nimetus AS hetkeseisund FROM Tellimus INNER JOIN Tellimuse_seisundi_liik ON Tellimus.tellimuse_seisundi_liik_kood = Tellimuse_seisundi_liik.tellimuse_seisundi_liik_kood WHERE Tellimus.tellimuse_seisundi_liik_kood <> 1 ORDER BY Tellimus.tellimuse_nr;`
2	`SELECT Tellimus.tellimuse_nr, CASE WHEN Tellimus.tootlemisel_aeg IS NOT NULL THEN 'Töötlemisel' WHEN Tellimus.valja_saadetud_aeg IS NOT NULL THEN 'Välja saadetud' WHEN Tellimus.kohale_toimetatud_aeg IS NOT NULL THEN 'Kohale toimetatud' WHEN Tellimus.tuhistatud_aeg IS NOT NULL THEN 'Tühistatud' ELSE 'Teadmata' END AS hetkeseisund FROM Tellimus WHERE Tellimus.ootel_aeg IS NULL ORDER BY Tellimus.tellimuse_nr;`

Disain	S2
3	SELECT Tellimus.tellimuse_nr, CASE WHEN Tellimus.on_tootlemisel = TRUE THEN 'Töötlemisel' WHEN Tellimus.on_valja_saadetud = TRUE THEN 'Välja saadetud' WHEN Tellimus.on_kohale_toimetatud = TRUE THEN 'Kohale toimetatud' WHEN Tellimus.on_tuhistatud = TRUE THEN 'Tühistatud' ELSE 'Teadmata' END AS hetkeseisund FROM Tellimus WHERE Tellimus.on_ootel = FALSE ORDER BY Tellimus.tellimuse_nr;
4	SELECT Tellimus.tellimuse_nr, F_tellimuse_seisundi_dekodeerimine (seisund) AS hetkeseisund FROM Tellimus WHERE F_tellimuse_seisundi_dekodeerimine (Tellimus.seisund) <> 'Ootel' ORDER BY tellimus.tellimuse_nr;
5	SELECT A.tellimuse_nr, A.hetkeseisund FROM (SELECT Tootlemisel_tellimus.tellimuse_nr,'Töötlemisel' AS hetkeseisund FROM Tootlemisel_tellimus UNION SELECT Tuhistatud_tellimus.tellimuse_nr,'Tühistatud' AS hetkeseisund FROM Tuhistatud_tellimus UNION SELECT Valja_saadetud_tellimus.tellimuse_nr,'Välja saadetud' AS hetkeseisund FROM Valja_saadetud_tellimus UNION SELECT Kohale_toimetatud_tellimus.tellimuse_nr,'Kohale toimetatud' AS hetkeseisund FROM Kohale_toimetatud_tellimus) A;
6	SELECT A.tellimuse_nr, A.hetkeseisund FROM (SELECT tellimuse_nr, 'Töötlemisel' AS hetkeseisund FROM Tellimus WHERE Tellimus.tellimuse_nr IN (SELECT Arve.tellimuse_nr FROM Arve WHERE Arve.tasumise_aeg IS NOT NULL) AND NOT EXISTS (SELECT * FROM Saadetis WHERE Tellimus.tellimuse_nr = Saadetis.tellimuse_nr) UNION SELECT tellimuse_nr,'Välja saadetud' AS hetkeseisund FROM Tellimus WHERE Tellimus.tellimuse_nr IN (SELECT Saadetis.tellimuse_nr FROM Saadetis WHERE Saadetis.kliendile_uleandmise_aeg IS NULL) UNION SELECT tellimuse_nr,'Kohale toimetatud' AS hetkeseisund FROM Tellimus WHERE Tellimus.tellimuse_nr IN (SELECT Saadetis.tellimuse_nr FROM Saadetis WHERE Saadetis.kliendile_uleandmise_aeg IS NOT NULL) UNION SELECT tellimuse_nr,'Tühistatud' AS hetkeseisund FROM Tellimus WHERE Tellimus.tellimuse_nr IN (SELECT Tellimuse_tuhistamine.tellimuse_nr FROM Tellimuse_tuhistamine)) A ORDER BY A.tellimuse_nr;

Kolmanda päringuga (S3) leian ühe konkreetse tellimuse hetkeseisundi. Päringu (Tabel 10) tulemusel tagastatakse tellimuse nr 200000 hetkeseisundi nimetus.

Tabel 10. Andmete lugemiseks S3 kasutatavad päringud.

Disain	**S3**
1	SELECT Tellimuse_seisundi_liik.nimetus AS hetkeseisund FROM Tellimus INNER JOIN Tellimuse_seisundi_liik ON Tellimus.tellimuse_seisundi_liik_kood = Tellimuse_seisundi_liik.tellimuse_seisundi_liik_kood WHERE Tellimus.tellimuse_nr=200000;
2	SELECT CASE WHEN Tellimus.ootel_aeg IS NOT NULL THEN 'Ootel' WHEN Tellimus.tootlemisel_aeg IS NOT NULL THEN 'Töötlemisel' WHEN Tellimus.valja_saadetud_aeg IS NOT NULL THEN 'Välja saadetud' WHEN Tellimus.kohale_toimetatud_aeg IS NOT NULL THEN 'Kohale toimetatud' WHEN Tellimus.tuhistatud_aeg IS NOT NULL THEN 'Tühistatud' ELSE 'Teadmata' END AS hetkeseisund FROM Tellimus WHERE Tellimus.tellimuse_nr=200000;
3	SELECT CASE WHEN Tellimus.on_ootel = TRUE THEN 'Ootel' WHEN Tellimus.on_tootlemisel = TRUE THEN 'Töötlemisel' WHEN Tellimus.on_valja_saadetud = TRUE THEN 'Välja saadetud' WHEN Tellimus.on_kohale_toimetatud = TRUE THEN 'Kohale toimetatud' WHEN Tellimus.on_tuhistatud = TRUE THEN 'Tühistatud' ELSE 'Teadmata' END AS hetkeseisund FROM Tellimus WHERE Tellimus.tellimuse_nr=200000;
4	SELECT F_tellimuse_seisundi_dekodeerimine (Tellimus.seisund) AS hetkeseisund FROM Tellimus WHERE Tellimus.tellimuse_nr=200000;
5	SELECT A.hetkeseisund FROM (SELECT Ootel_tellimus.tellimuse_nr,'Ootel' AS hetkeseisund FROM Ootel_tellimus UNION SELECT Tootlemisel_tellimus.tellimuse_nr,'Töötlemisel' AS hetkeseisund FROM Tootlemisel_tellimus UNION SELECT Tuhistatud_tellimus.tellimuse_nr,'Tühistatud' AS hetkeseisund FROM Tuhistatud_tellimus UNION SELECT Valja_saadetud_tellimus.tellimuse_nr,'Välja saadetud' AS hetkeseisund FROM Valja_saadetud_tellimus UNION SELECT Kohale_toimetatud_tellimus.tellimuse_nr,'Kohale toimetatud' AS hetkeseisund FROM Kohale_toimetatud_tellimus) A WHERE A.tellimuse_nr=200000;

Disain	S3
6	SELECT CASE WHEN (Tellimus.tellimuse_nr IN (SELECT Arve.tellimuse_nr FROM Arve WHERE Arve.tasumise_aeg IS NULL) OR NOT EXISTS (SELECT * FROM Arve WHERE Tellimus.tellimuse_nr = Arve.tellimuse_nr)) AND NOT EXISTS (SELECT * FROM Tellimuse_tuhistamine WHERE Tellimus.tellimuse_nr = Tellimuse_tuhistamine.tellimuse_nr) THEN 'Ootel' WHEN Tellimus.tellimuse_nr IN (SELECT Arve.tellimuse_nr FROM Arve WHERE Arve.tasumise_aeg IS NOT NULL) AND NOT EXISTS (SELECT * FROM Saadetis WHERE Tellimus.tellimuse_nr = Saadetis.tellimuse_nr) THEN 'Töötlemisel' WHEN Tellimus.tellimuse_nr IN (SELECT Saadetis.tellimuse_nr FROM Saadetis WHERE Saadetis.kliendile_uleandmise_aeg IS NULL) THEN 'Välja saadetud' WHEN Tellimus.tellimuse_nr IN (SELECT Saadetis.tellimuse_nr FROM Saadetis WHERE Saadetis.kliendile_uleandmise_aeg IS NOT NULL) THEN 'Kohale toimetatud' WHEN Tellimus.tellimuse_nr IN (SELECT Tellimuse_tuhistamine.tellimuse_nr FROM tellimuse_tuhistamine) THEN 'Tühistatud' END AS hetkeseisund FROM Tellimus WHERE Tellimus.tellimuse_nr=200000;

Märkus disaini 5 päringu kohta – PostgreSQL (11) oskab seda täita nii, et kõigepealt otsitakse igast tabelist rida kus on tellimus 200000 ja saadud tulemused pannakse ühendi abil kokku, mitte ei leita kõigepealt suurt ühendit, mida järgnevalt hakatakse tingimuse alusel piirama.

Neljanda päringuga (S4) leian kõik sellised seisundid, milles pole ühtegi tellimust. Seega tagastatakse päringu (Tabel 11) tulemusel nende seisundite nimetused.

Tabel 11. Andmete lugemiseks S4 kasutatavad päringud.

Disain	S4
1	SELECT Tellimuse_seisundi_liik.nimetus AS seisund FROM Tellimuse_seisundi_liik WHERE NOT EXISTS (SELECT * FROM Tellimus WHERE Tellimuse_seisundi_liik.tellimuse_seisundi_liik_kood = Tellimus.tellimuse_seisundi_liik_kood);

Disain	S4
2	SELECT 'Ootel' AS seisund WHERE NOT EXISTS (SELECT * FROM Tellimus WHERE Tellimus.ootel_aeg IS NOT NULL) UNION SELECT 'Töötlemisel' AS seisund WHERE NOT EXISTS (SELECT * FROM Tellimus WHERE Tellimus.tootlemisel_aeg IS NOT NULL) UNION SELECT 'Välja saadetud' AS seisund WHERE NOT EXISTS (SELECT * FROM Tellimus WHERE Tellimus.valja_saadetud_aeg IS NOT NULL) UNION SELECT 'Kohale toimetatud' AS seisund WHERE NOT EXISTS (SELECT * FROM Tellimus WHERE Tellimus.kohale_toimetatud_aeg IS NOT NULL) UNION SELECT 'Tühistatud' AS seisund WHERE NOT EXISTS (SELECT * FROM Tellimus WHERE Tellimus.tuhistatud_aeg IS NOT NULL);
3	SELECT 'Ootel' AS seisund WHERE NOT EXISTS (SELECT * FROM Tellimus WHERE Tellimus.on_ootel = TRUE) UNION SELECT 'Töötlemisel' AS seisund WHERE NOT EXISTS (SELECT * FROM Tellimus WHERE Tellimus.on_tootlemisel = TRUE) UNION SELECT 'Välja saadetud' AS seisund WHERE NOT EXISTS (SELECT * FROM Tellimus WHERE Tellimus.on_valja_saadetud = TRUE) UNION SELECT 'Kohale toimetatud' AS seisund WHERE NOT EXISTS (SELECT * FROM Tellimus WHERE Tellimus.on_kohale_toimetatud = TRUE) UNION SELECT 'Tühistatud' AS seisund WHERE NOT EXISTS (SELECT * FROM Tellimus WHERE Tellimus.on_tuhistatud = TRUE);

Disain	S4
4	SELECT 'Ootel' AS seisund WHERE NOT EXISTS (SELECT * FROM Tellimus WHERE F_tellimuse_seisundi_dekodeerimine (Tellimus.seisund) = 'Ootel') UNION SELECT 'Töötlemisel' AS seisund WHERE NOT EXISTS (SELECT * FROM Tellimus WHERE F_tellimuse_seisundi_dekodeerimine(Tellimus.seisund) = 'Töötlemisel') UNION SELECT 'Välja saadetud' AS seisund WHERE NOT EXISTS (SELECT * FROM Tellimus WHERE F_tellimuse_seisundi_dekodeerimine (Tellimus.seisund) = 'Välja saadetud') UNION SELECT 'Kohale toimetatud' AS seisund WHERE NOT EXISTS (SELECT * FROM Tellimus WHERE F_tellimuse_seisundi_dekodeerimine (Tellimus.seisund) = 'Kohale toimetatud') UNION SELECT 'Tühistatud' AS seisund WHERE NOT EXISTS (SELECT * FROM Tellimus WHERE F_tellimuse_seisundi_dekodeerimine (Tellimus.seisund) = 'Tühistatud');
5	SELECT 'Ootel' AS seisund WHERE NOT EXISTS (SELECT * FROM Ootel_tellimus) UNION SELECT 'Töötlemisel' AS seisund WHERE NOT EXISTS (SELECT * FROM Tootlemisel_tellimus) UNION SELECT 'Välja saadetud' AS seisund WHERE NOT EXISTS (SELECT * FROM Valja_saadetud_tellimus) UNION SELECT 'Kohale toimetatud' AS seisund WHERE NOT EXISTS (SELECT * FROM Kohale_toimetatud_tellimus) UNION SELECT 'Tühistatud' AS seisund WHERE NOT EXISTS (SELECT * FROM Tuhistatud_tellimus);

Disain	S4
6	SELECT 'Ootel' AS seisund WHERE NOT EXISTS (SELECT * FROM Tellimus WHERE (Tellimus.tellimuse_nr IN (SELECT Arve.tellimuse_nr FROM Arve WHERE Arve.tasumise_aeg IS NULL) OR NOT EXISTS (SELECT * FROM Arve WHERE Tellimus.tellimuse_nr = Arve.tellimuse_nr)) AND NOT EXISTS (SELECT * FROM Tellimuse_tuhistamine WHERE Tellimus.tellimuse_nr = Tellimuse_tuhistamine.tellimuse_nr)) UNION SELECT 'Töötlemisel' AS seisund WHERE NOT EXISTS (SELECT * FROM Tellimus WHERE (Tellimus.tellimuse_nr IN (SELECT Arve.tellimuse_nr FROM Arve WHERE Arve.tasumise_aeg IS NOT NULL) AND NOT EXISTS (SELECT * FROM Saadetis WHERE Tellimus.tellimuse_nr = Saadetis.tellimuse_nr))) UNION SELECT 'Välja saadetud' AS seisund WHERE NOT EXISTS (SELECT * FROM Tellimus WHERE (Tellimus.tellimuse_nr IN (SELECT Saadetis.tellimuse_nr FROM Saadetis WHERE Saadetis.kliendile_uleandmise_aeg IS NULL))) UNION SELECT 'Kohale toimetatud' AS seisund WHERE NOT EXISTS (SELECT * FROM Tellimus WHERE (Tellimus.tellimuse_nr IN (SELECT Saadetis.tellimuse_nr FROM Saadetis WHERE Saadetis.kliendile_uleandmise_aeg IS NOT NULL))) UNION SELECT 'Tühistatud' AS seisund WHERE NOT EXISTS (SELECT * FROM Tellimus WHERE (Tellimus.tellimuse_nr IN (SELECT Tellimuse_tuhistamine.tellimuse_nr FROM Tellimuse_tuhistamine)));

Paljud ülesande (S4) lahendused kasutavad PostgreSQL võimalust kirjutada SELECT lause, kus on SELECT ja WHERE klausel, kuid ei ole FROM klauslit.

Viienda päringuga (S5) leian iga seisundi kohta selles olevate tellimuste arv. Kui selles seisundis pole ühtegi tellimust, siis on tagastatavaks väärtuseks 0. Päringu (Tabel 12) tulemusel tagastatakse iga seisundi kohta nimetus ning selles seisundis olevate tellimuste arv.

Tabel 12. Andmete lugemiseks S5 kasutatavad päringud.

Disain	S5
1	SELECT Tellimuse_seisundi_liik.nimetus AS seisund, COALESCE (A.tellimuste_arv, 0) AS tellimuste_arv FROM Tellimuse_seisundi_liik LEFT JOIN (SELECT Tellimus.tellimuse_seisundi_liik_kood, COUNT(*) AS tellimuste_arv FROM Tellimus GROUP BY Tellimus.tellimuse_seisundi_liik_kood) a ON Tellimuse_seisundi_liik.tellimuse_seisundi_liik_kood = A.tellimuse_seisundi_liik_kood;

Disain	S5
2	WITH Seisundite_arvud AS (SELECT COUNT(*) FILTER (WHERE ootel_aeg IS NOT NULL) AS ootel_tellimuste_arv, COUNT(*) FILTER (WHERE tootlemisel_aeg IS NOT NULL) AS tootlemisel_tellimuste_arv, COUNT(*) FILTER (WHERE valja_saadetud_aeg IS NOT NULL) AS valja_saadetud_tellimuste_arv, COUNT(*) FILTER (WHERE kohale_toimetatud_aeg IS NOT NULL) AS kohale_toimetatud_tellimuste_arv, COUNT(*) FILTER (WHERE tuhistatud_aeg IS NOT NULL) AS tuhistatud_tellimuste_arv FROM Tellimus) SELECT 'Ootel' AS seisund, ootel_tellimuste_arv FROM Seisundite_arvud UNION SELECT 'Töötlemisel' AS seisund, tootlemisel_tellimuste_arv FROM Seisundite_arvud UNION SELECT 'Välja saadetud' AS seisund, valja_saadetud_tellimuste_arv FROM Seisundite_arvud UNION SELECT 'Kohale toimetatud' AS seisund, kohale_toimetatud_tellimuste_arv FROM Seisundite_arvud UNION SELECT 'Tühistatud' AS seisund, tuhistatud_tellimuste_arv FROM Seisundite_arvud;
3	WITH Seisundite_arvud AS (SELECT COUNT(*) FILTER (WHERE on_ootel = TRUE) AS ootel_tellimuste_arv, COUNT(*) FILTER (WHERE on_tootlemisel = TRUE) AS tootlemisel_tellimuste_arv, COUNT(*) FILTER (WHERE on_valja_saadetud = TRUE) AS valja_saadetud_tellimuste_arv, COUNT(*) FILTER (WHERE on_kohale_toimetatud = TRUE) AS kohale_toimetatud_tellimuste_arv, COUNT(*) FILTER (WHERE on_tuhistatud = TRUE) AS tuhistatud_tellimuste_arv FROM Tellimus) SELECT 'Ootel' AS seisund, ootel_tellimuste_arv FROM Seisundite_arvud UNION SELECT 'Töötlemisel' AS seisund, tootlemisel_tellimuste_arv FROM Seisundite_arvud UNION SELECT 'Välja saadetud' AS seisund, valja_saadetud_tellimuste_arv FROM Seisundite_arvud UNION SELECT 'Kohale toimetatud' AS seisund, kohale_toimetatud_tellimuste_arv FROM Seisundite_arvud UNION SELECT 'Tühistatud' AS seisund, tuhistatud_tellimuste_arv FROM Seisundite_arvud;
4	SELECT COUNT(*) AS tellimuste_arv, 'Ootel' AS seisund FROM Tellimus WHERE F_tellimuse_seisundi_dekodeerimine (Tellimus.seisund) = 'Ootel' UNION

Disain	S5
	SELECT COUNT(*) AS tellimuste_arv, 'Töötlemisel' AS seisund FROM Tellimus WHERE F_tellimuse_seisundi_dekodeerimine (Tellimus.seisund) = 'Töötlemisel' UNION SELECT COUNT(*) AS tellimuste_arv, 'Välja saadetud' AS seisund FROM Tellimus WHERE F_tellimuse_seisundi_dekodeerimine (Tellimus.seisund) = 'Välja saadetud' UNION SELECT COUNT(*) AS tellimuste_arv, 'Kohale toimetatud' AS seisund FROM Tellimus WHERE F_tellimuse_seisundi_dekodeerimine (Tellimus.seisund) = 'Kohale toimetatud' UNION SELECT COUNT(*) AS tellimuste_arv, 'Tühistatud' AS seisund FROM Tellimus WHERE F_tellimuse_seisundi_dekodeerimine (Tellimus.seisund) = 'Tühistatud';
5	SELECT A.seisund, A.tellimuste_arv FROM (SELECT COUNT(*) AS tellimuste_arv,'Ootel' AS seisund FROM Ootel_tellimus UNION SELECT COUNT(*) AS tellimuste_arv,'Töötlemisel' AS seisund FROM Tootlemisel_tellimus UNION SELECT COUNT(*) AS tellimuste_arv,'Tühistatud' AS seisund FROM Tuhistatud_tellimus UNION SELECT COUNT(*) AS tellimuste_arv,'Välja saadetud' AS seisund FROM Valja_saadetud_tellimus UNION SELECT COUNT(*) AS tellimuste_arv,'Kohale toimetatud' AS seisund FROM Kohale_toimetatud_tellimus) A;
6	SELECT A.seisund, A.tellimuste_arv FROM (SELECT COUNT(*) AS tellimuste_arv,'Ootel' AS seisund FROM Tellimus WHERE (Tellimus.tellimuse_nr IN (SELECT Arve.tellimuse_nr FROM Arve WHERE Arve.tasumise_aeg IS NULL) OR NOT EXISTS (SELECT * FROM Arve WHERE Tellimus.tellimuse_nr = Arve.tellimuse_nr)) AND NOT EXISTS (SELECT * FROM Tellimuse_tuhistamine WHERE Tellimus.tellimuse_nr = Tellimuse_tuhistamine.tellimuse_nr) UNION SELECT COUNT(*) AS tellimuste_arv,'Töötlemisel' AS seisund FROM Tellimus WHERE Tellimus.tellimuse_nr IN (SELECT Arve.tellimuse_nr FROM Arve WHERE arve.tasumise_aeg IS NOT NULL) AND NOT EXISTS (SELECT * FROM saadetis WHERE Tellimus.tellimuse_nr = Saadetis.tellimuse_nr) UNION SELECT COUNT(*) AS tellimuste_arv,'Välja saadetud' AS seisund FROM Tellimus WHERE Tellimus.tellimuse_nr IN (SELECT Saadetis.tellimuse_nr FROM Saadetis WHERE Saadetis.kliendile_uleandmise_aeg IS NULL) UNION SELECT COUNT(*) AS tellimuste_arv,'Kohale toimetatud' AS seisund FROM Tellimus WHERE Tellimus.tellimuse_nr IN (SELECT Saadetis.tellimuse_nr FROM Saadetis WHERE Saadetis.kliendile_uleandmise_aeg IS NOT NULL) UNION SELECT COUNT(*) AS tellimuste_arv,'Tühistatud' AS seisund FROM Tellimus WHERE Tellimus.tellimuse_nr IN (SELECT Tellimuse_tuhistamine.tellimuse_nr FROM Tellimuse_tuhistamine)) A;

Huvitava tähelepanekuna on disaini 1 korral Joonis 29 esitatud päring (mis on kompaktsem) umbes kuus korda aeglasem kui minu kasutatav päring ning Joonis 30 esitatud päring (mis on ka kompaktsem) kaks kuni kolm korda aeglasem kui minu kasutatav päring. Joonis 31 esitatud päring disaini 4 korral on peaaegu kümme korda aeglasem kui minu kasutatav päring.

```
SELECT Tellimuse_seisundi_liik.nimetus AS seisund,
Count(Tellimus.tellimuse_seisundi_liik_kood) AS tellimuste_arv
FROM Tellimuse_seisundi_liik LEFT JOIN Tellimus USING(tellimuse_seisundi_liik_kood)
GROUP BY Tellimuse_seisundi_liik.nimetus;
```

Joonis 29. Alternatiivne koondandmete leidmise päring disaini nr 1 jaoks.

```
SELECT Tellimuse_seisundi_liik.nimetus AS seisund, (SELECT Count(*) AS c FROM
Tellimus WHERE Tellimus.tellimuse_seisundi_liik_kood=
Tellimuse_seisundi_liik.tellimuse_seisundi_liik_kood) AS tellimuste_arv
FROM Tellimuse_seisundi_liik;
```

Joonis 30. Alternatiivne koondandmete leidmise päring disaini nr 1 jaoks.

```
WITH Seisundite_arvud AS (
SELECT COUNT(*) FILTER (WHERE F_tellimuse_seisundi_dekodeerimine (Tellimus.seisund) =
'Ootel') AS ootel_tellimuste_arv,
COUNT(*) FILTER (WHERE F_tellimuse_seisundi_dekodeerimine (Tellimus.seisund) =
'Töötlemisel') AS tootlemisel_tellimuste_arv,
COUNT(*) FILTER (WHERE F_tellimuse_seisundi_dekodeerimine (Tellimus.seisund) = 'Välja
saadetud') AS valja_saadetud_tellimuste_arv,
COUNT(*) FILTER (WHERE F_tellimuse_seisundi_dekodeerimine (Tellimus.seisund) =
'Kohale toimetatud') AS kohale_toimetatud_tellimuste_arv,
COUNT(*) FILTER (WHERE F_tellimuse_seisundi_dekodeerimine (Tellimus.seisund) =
'Tühistatud') AS tuhistatud_tellimuste_arv
FROM Tellimus)
SELECT 'Ootel' AS seisund, ootel_tellimuste_arv
FROM Seisundite_arvud
UNION
SELECT 'Töötlemisel' AS seisund, tootlemisel_tellimuste_arv
FROM Seisundite_arvud
UNION
SELECT 'Välja saadetud' AS seisund, valja_saadetud_tellimuste_arv
FROM Seisundite_arvud
UNION
SELECT 'Kohale toimetatud' AS seisund, kohale_toimetatud_tellimuste_arv
FROM Seisundite_arvud
UNION
SELECT 'Tühistatud' AS seisund, tuhistatud_tellimuste_arv
FROM Seisundite_arvud;
```

Joonis 31. Alternatiivvne koondandmete leidmise päring disaini nr 4 jaoks.

5.6.2 Andmete muutmine

Järgnevalt esitan eksperimendis andmemuudatuse testiks kasutatava operatsiooni kirjelduse ja nende läbi viimiseks vajalikud SQL laused (Tabel 13) igale kataloogis esitatud disainile.

Täpsemalt mõtlen andmemuudatus operatsiooni (U) all konkreetse tellimuse seisundi muutmist. Selleks viin tellimuse nr 290550 seisundist *Välja saadetud (3)* lõppseisundisse *Kohale toimetatud (4).*

Tabel 13. Andmete muutmiseks kasutatavad päringud.

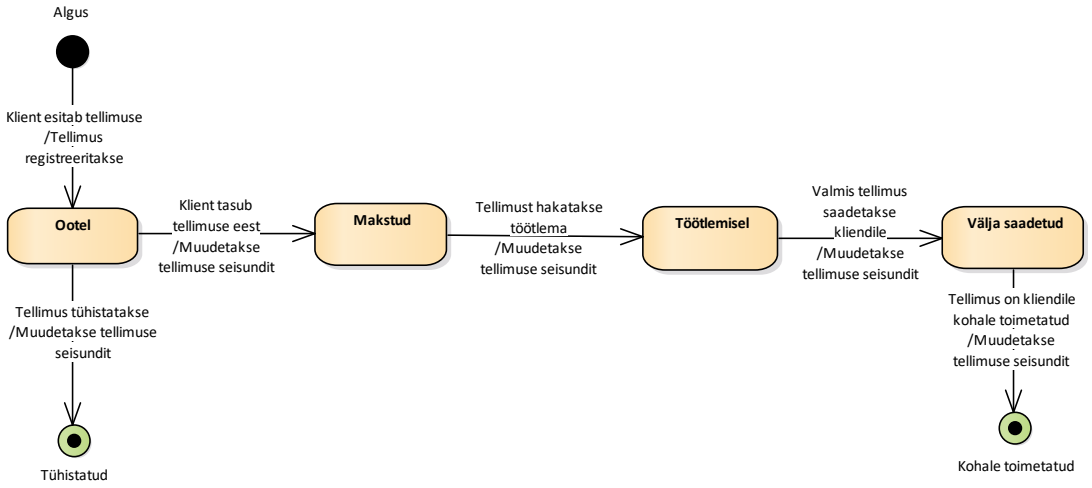
Disain	**U**
1	`UPDATE Tellimus SET tellimuse_seisundi_liik_kood = 4 WHERE tellimuse_nr=290550;`
2	`UPDATE Tellimus SET valja_saadetud_aeg=NULL, kohale_toimetatud_aeg=LOCALTIMESTAMP(0) WHERE tellimuse_nr=290550;`
3	`UPDATE Tellimus SET on_valja_saadetud=FALSE, on_kohale_toimetatud=TRUE WHERE tellimuse_nr=290550;`
4	`UPDATE Tellimus SET seisund=F_tellimuse_seisundi_kodeerimine('Välja saadetud') WHERE tellimuse_nr=290550;`

5	`START TRANSACTION;`
	`DELETE FROM Valja_saadetud_tellimus WHERE tellimuse_nr=290550;`
	`INSERT INTO Kohale_toimetatud_tellimus (tellimuse_nr) VALUES (290550);`
	`COMMIT;`

6	`UPDATE Saadetis SET kliendile_uleandmise_aeg=LOCALTIMESTAMP(0) WHERE tellimuse_nr=290550;`

5.6.3 Uue seisundi lisamine

Järgnevalt esitan eksperimendis seisundi lisamise (I) ülesande kirjelduse igale kataloogis esitatud disainile. Täpsemalt vaatlen, mis tegevused tuleb läbi viia juhul kui infosüsteemi jaoks huvipakkuvate tellimuse seisundite (Joonis 2) hulka lisandub seisund *Makstud*. Käsitlen muudatusi tabelite struktuuris, kitsendustes ja funktsioonides, kuid mitte vaadetes ja päringutes. Joonis 32 esitab seisundidiagrammi, mis kirjeldab antud muudatuse tulemust. Seejuures peab arvesse võtma ka disainide kirjelduses esitatud nõudeid (vt jaotis 4.1) ning tegema sellest tulenevalt muudatusi ka olemasolevates kontrollides.



Joonis 32. Tellimuste elutsükleid kirjeldav seisundidiagramm peale uue seisundi lisamist.

Kuna kõigile disainidele vastava SQL koodi lisamine antud töösse on liiga pikk, siis on need lisatud GitHubi salve, mis on leitav aadressilt https://github.com/susannapeek1/seisundid_SQL. Täpsemalt leiab antud testiks kasutatavad laused iga disaini alamkataloogis olevast kaustast, mille nimi sisaldab tähte „I“. [70]

Seisundiklassifikaator (1) disaini puhul tuleb lisada uus rida tabelisse *tellimuse_seisundi_liik*, kasutades selleks INSERT INTO [64] lauset. Tabel hoiab andmeid kõigi võimalike seisundikoodide ja neile vastavate nimetuste kohta.

Temporaalsed veerud (2) disaini puhul tuleb tabelisse *Tellimus* lisada TIMESTAMP tüüpi veerg (*makstud_aeg*). Lisaks tuleb täiendada olemasolevat CHECK kitsendust, mis tagab, et iga tellimus on täpselt ühes seisundis, lisades kitsenduse loogikaavaldisse uue veeru nime. Antud muudatuse viin läbi kasutades ALTER [48] lauset, millega kustutan kitsenduse vana versiooni ja loon uue. Samuti tuleb luua otsingute kiirendamiseks uuele veerule osaline indeks, mis on loodud ka teistele seisunditele vastavatele veergudele.

Tõeväärtustüüpi (3) disaini puhul tuleb tabelisse *Tellimus* lisada tõeväärtustüüpi veerg (*on_makstud*). Kuna tabelis võib olla andmeid, siis kõigepealt lisan mittekohustusliku veeru, siis määran, et kõikides ridades on selles veerus väärtus FALSE ning lõpuks muudan veeru kohustuslikuks. Lisaks tuleb täiendada olemasolevat CHECK kitsendust, mis tagab, et iga tellimus on täpselt ühes seisundis, lisades kitsenduse loogikaavaldisse uue veeru nime. Antud muudatuse viin läbi kasutades ALTER [48] lauset, millega

kustutan kitsenduse vana versiooni ja loon uue. Samuti tuleb luua otsingute kiirendamiseks uuele veerule osaline indeks, mis on loodud ka teistele seisunditele vastavatele veergudele. Lisaks tuleb muuta seisundimuudatuse aja registreerimiseks mõeldud trigerit – tuleb muuta veergude hulka, milles andmete muutmine trigeri käivitab ja WHEN klauslis olevat tingimust. Selleks tuleb kustutada olemasolev triger ja luua uus.

Vektrokodeerimine (4) disaini puhul tuleb teha muudatus seisundi väärtuste kodeerimiseks ja dekodeerimiseks kasutatavates funktsioonides. Selleks asendan olemasolevad funktsioonid uutega, kus arvestatakse uue võimaliku koodiga – 000001. Muudatuse jaoks kasutan CREATE OR REPLACE FUNCTION lauset. Samuti tuleb muuta CHECK kitsendust, mis kontrollib vektorkoodi lubatud pikkust. Antud muudatuse viin läbi kasutades ALTER [48] lauseid – kõigepealt tuleb kitsendus kustutada ja siis uuesti luua. Kuid enne selle uuesti loomist tuleb teha olemasolevate andmete seisundi väärtustes muudatus, et kõigile olemitele jääks siiski kehtima sama seisund, mis enne vektorkoodi struktuuri muutust. Selleks koostan UPDATE [78] lauseid, mis järgivad Tabel 14 olevaid reegleid.

Tabel 14. Uue Seisundi lisamise jaoks uued seisundiväärtused Vektorkodeerimine disainile.

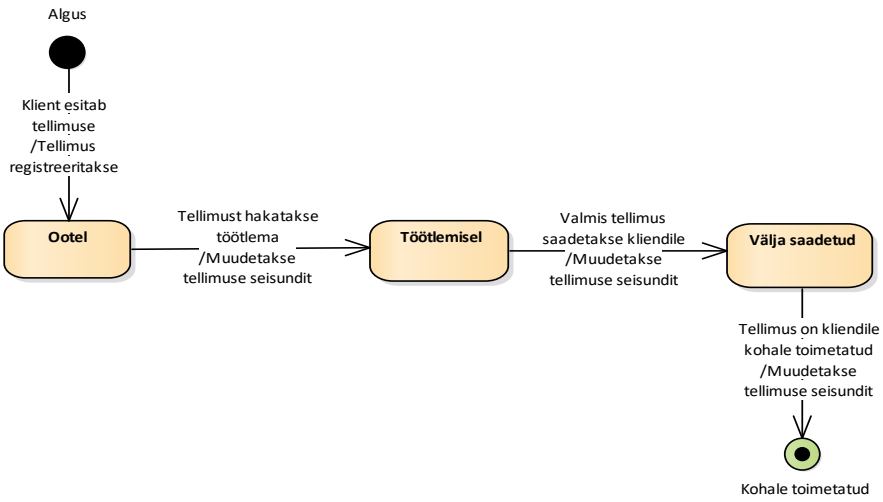
Seisundi nimetus	**Vana vektorkood**	**Uus vektorkood**
Ootel	10000	100000
Töötlemisel	01000	010000
Välja saadetud	00100	001000
Kohale toimetatud	00010	000100
Tühistatud	00001	000010
Makstud	-	000001

Seisundite esitamine põhiolemitüübi alamtüüpidena (5) disaini puhul tuleb luua uus tabel *Makstud_tellimus* kasutades selleks CREATE TABLE [55] lauset. Lisaks tuleb luua sarnaselt teistele alamtüüpidele vastavatele tabelitele uue tabeliga seotud triger, mis kontrollib, et tellimus saab olla korraga vaid ühes seisundis. Samuti tuleb täiendada ka teistele seisunditele vastavate tabelitega seotud samasisulisi trigereid. Lisaks tuleb täiendada tabeliga *Tellimus* seotud trigerit, mis tagab, et iga tellimus peab kindlasti olema mingis seisundis. Olemasolevate trigerite muutmisel on vaja asendada trigerite funktsioonid uutega, kus arvestatakse lisandunud tabeliga.

Seisundite tuletamine olemite ja seoste kohta registreeritud andmetest (6) disaini puhul tuleb viia läbi muudatus kõigepealt ärireeglite kirjelduses. Näiteks tuleb kirja panna reegel, et tellimus on seisundis makstud kui sellele tellimusele loodud arve on makstud. Samuti tuleb kirja panna reegel, et tellimus on seisundis töötlemisel, kui sellele eksisteerib rida uues loodavas tabelis, mis hoiab andmeid tööks kulunud aja kohta (alguse aeg ja lõpetamise aeg). Kui vastav muudatus reeglites on tehtud, siis tuleb luua see eelnevalt kirjeldatud uus tabel *Tootlemise*l kasutades selleks CREATE TABLE [55] lauset.

5.6.4 Seisundi eemaldamine

Järgnevalt esitan eksperimendis seisundi eemaldamise (D) ülesande kirjelduse igale kataloogis esitatud disainile. Täpsemalt vaatlen, mis tegevused tuleb läbi viia juhul kui seoses äriprotsesside ümberkorraldamisega ei paku infosüsteemile enam huvi olemasolev seisund *Tühistatud* (Joonis 2). Käsitlen muudatusi tabelite struktuuris, kitsendustes ja funktsioonides, kuid mitte vaadetes ja päringutes. Joonis 33 esitab seisundidiagrammi, mis kirjeldab antud muudatuse tulemust. Seejuures peab arvesse võtma ka disainide kirjelduses esitatud nõudeid (vt jaotis 4.1) ning tegema sellest tulenevalt muudatusi ka olemasolevates andmekontrollides. Lähtun eeldusest, et eemaldatava seisundiga pole seotud ühtegi tellimust. Kui see eeldus pole täidetud kuid soov on operatsioon ikkagi läbi viia, siis tuleb lisaks koostada teisendusreeglid, mis aitavad eemaldatavas seisundis olevatele tellimustele määrata uue seisundi.



Joonis 33. Tellimuste elutsükleid kirjeldav seisundidiagramm peale seisundi eemaldamist.

Kuna kõigile disainidele vastava SQL koodi lisamine antud töösse on liiga pikk, siis on need lisatud GitHubi salve, mis on leitav aadressilt https://github.com/susannapeek1/seisundid_SQL. Täpsemalt leiab antud testiks kasutatavad laused iga disaini alamkataloogis olevast kaustast, mille nimi sisaldab tähte „D“. [70]

Seisundiklassifikaator (1) disaini puhul tuleb kustutada eemaldatava seisundi kohta käiv rida tabelist *Tellimuse_seisundi_liik*, kasutades selleks DELETE [42] lauset. Välisvõti tabelis *Tellimus*, mille loomisel on kasutatud ON DELETE NO ACTION määrangut, tagab, et vähemalt ühe tellimusega seotud seisundiklassifikaatori väärtuse kustutamine ebaõnnestub. Seega on siinkohal tegemist täiendava kontrolliga, mis annab märku kui üritatakse eemaldada seisundit, mida ei peaks eemaldama.

Temporaalsed veerud (2) disaini puhul tuleb kustutada olemasolev ajaline veerg (*tuhistamise_aeg*) tabelist *Tellimus*. Lisaks tuleb muuta olemasolevat CHECK kitsendust, mis kontrollib, et iga tellimus on täpselt ühes seisundis, eemaldades kitsenduse loogikaavaldisest veeru nimetuse. Antud muudatuse viin läbi kasutades ALTER [48] lauset, millega kustutan kitsenduse vana versiooni ja loon uue. Eemaldatavale veerule loodud indeks kustutatakse veeru kustutamisel automaatselt.

Tõeväärtustüüpi (3) disaini puhul tuleb kustutada olemasolev tõeväärtustüüpi veerg (*on_tuhistatud*) tabelist *Tellimus*. Enne seda tuleb muuta trigerit, mille tulemusel seisundi muutuse korral muudetakse väärtust ka veerus *seisundimuudatuse_aeg* – tuleb muuta veergude hulka, milles andmete muutmine trigeri käivitab ja WHEN klauslis olevat tingimust. Selleks tuleb kustutada olemasolev triger ja luua uus. Lisaks tuleb enne veeru kustutamist muuta olemasolevat CHECK kitsendust, mis tagab, et iga tellimus on täpselt ühes seisundis, eemaldades kitsenduse loogikaavaldisest veeru nime. Antud muudatuse viin läbi kasutades ALTER [48] lauset, millega kustutan kitsenduse vana versiooni ja loon uue. Eemaldatavale veerule loodud indeks kustutatakse veeru kustutamisel automaatselt.

Vektrokodeerimine (4) disaini puhul tuleb teha muudatus seisundi väärtuste kodeerimiseks ja dekodeerimiseks kasutatavates funktsioonides. Selleks asendan olemasolevad funktsioonid uuega, kus enam ei arvestata võimalusega, et seisund võib olla *Tühistatud*. Muudatuse jaoks kasutan CREATE OR REPLACE FUNCTION lauset. Samuti tuleb muuta CHECK kitsendust, mis kontrollib vektorkoodi lubatud pikkust.

Antud muudatuse viin läbi kasutades ALTER [48] lauseid – kõigepealt tuleb kitsendus kustutada ja siis uuesti luua. Kuid enne selle uuesti loomist tuleb teha olemasolevate andmete seisundi väärtustes muudatus, et kõigile olemitele jääks siiski kehtima sama seisund, mis enne vektorkoodi muutust. Selleks koostan UPDATE [78] laused, mis järgivad Tabel 15 olevaid reegleid.

Tabel 15. Seisundi eemaldamine jaoks uued seisundiväärtused Vektorkodeerimine disainile.

Seisundi nimetus	**Vana vektorkood**	**Uus vektorkood**
Ootel	10000	1000
Töötlemisel	01000	0100
Välja saadetud	00100	0010
Kohale toimetatud	00010	0001

Seisundite esitamine põhiolemitüübi alamtüüpidena (5) disaini puhul tuleb kustutada olemasolev tabel *Tuhistatud_tellimus* kasutades selleks DROP TABLE [56] lauset. Tabeliga seotud triger kustutatakse automaatselt. Trigeri funktsioon tuleb eraldi lausega kustutada. Samuti tuleb täiendada ka teistele alamtüüpidele vastavate tabelitega seotud samasisulisi trigereid. Lisaks tuleb täiendada tabeliga *Tellimus* seotud trigerit, mis tagab, et iga tellimus peab kindlasti olema mingis seisundis. Olemasolevate trigerite muutmisel on vaja asendada trigerite funktsioonid uutega, kus enam ei arvestata kustutatud tabeliga.

Seisundite tuletamine olemite ja seoste kohta registreeritud andmetest (6) disaini puhul tulenevalt ärireeglitest tuleb kustutada selleks tabel *Tellimuse_tühistamine*, milleks kasutatan DROP TABLE [56] lauset. Lisaks tuleb muuta ärireeglite kirjeldust nii, et need ei arvestaks enam eemaldatud seisundiga.

6 Eksperimendi tulemused

Selles peatükis toon välja kõikide eksperimendis läbi viidud mõõtmiste tulemused. Selleks, et tabelid oleksid kergemini jälgitavad, kasutan disainidele viitamisel eksperimendi kirjelduses (vt jaotis 5.2) nimetatud numbreid. **Rasvaselt** tähistatakse mõõtmistulemused, mis on sama lähteülesande korral parim (väikseim).

Tabel 16 esitab andmebaasi mahu megabaitides erinevate andmehulkade ja disainide korral. Andmemahud arvestavad ka indeksite mahtu.

Tabel 16. Andmebaasi maht erinevate disainide korral (MB).

Disain	**Andmehulk 1**	**Andmehulk 2**	**Andmehulk 3**
1	**58,06**	**115,95**	**231,81**
2	58,07	116,0	231,84
3	61,84	123,53	246,89
4	66,08	132,10	264,15
5	75,24	150,34	300,63
6	107,05	213,95	427,77

Andmebaasis luuakse indeksid, et kiirendada päringute täitmist. Sellel põhjusel lõin ka indeksid antud lõputöös esitatud disainidele seal kus nägin selleks vajadust. Samas on indeksite kasutamisel ka negatiivseid külg. Nimelt aeglustavad need andmemuudatuse operatsioone (sest muutes indekseeritavaid andmeid on vaja muuta ka indeksit) ning suurendavad andmemahtu. Nagu tabelid, salvestatakse ka indeksid andmebaasis sisemiselt plokkidesse ja talletatakse püsivalt kettale. [79] Uurimaks, kuidas on mõjutab indeksite kasutuselevõtt töö eksperimendis võrreldavaid andmemahte, viisin läbi lisatesti, mille tulemusel leidsin disaini tabelite andmemahu ilma indeksiteta. Seega Tabel 17 esitab antud testi tulemused megabaitides kõigi disainide ja andmehulkade korral.

Tabel 17. Andmebaasi andmemaht ilma indeksite erinevate disainide korral (MB).

Disain	**Andmehulk 1**	**Andmehulk 2**	**Andmehulk 3**
1	36,56	73,08	146,11
2	**36,55**	**73,07**	**146,04**
3	40,32	80,60	161,16
4	40,32	80,60	161,16
5	53,69	107,34	214,84
6	70,4	140,77	281,53

Ridade arvu kasvamisel suueneb andmemaht (indeksitega või ilma) lineaarselt – nii Tabel 16 kuid Tabel 17 korral on andmemaht andmehulga 3 korral suurem andmemahust andmehulga 1 korral umbes neli korda. Sama palju on ka suurem tellimuste arv vastavates tabelites.

Tabel 18 esitab päringute (S1-S5) ja andmemuudatus operatsioonide (U) kiiruste mõõtmistulemuste geomeetrilise keskmise erinevate disainide ja andmehulk 1 (5000 00 tellimuse) korral. Mõõtmistulemused on sekundites.

Tabel 18. Päringute ja andmemuudatus operatsioonide geomeetrilised keskmised kiirused (sekundites) andmehulga 1 korral.

Disain	**S1**	**S2**	**S3**	**S4**	**S5**	**U**
1	0,60	5,29	0,00044	**0,00068**	0,61	0,00126
2	0,92	**1,85**	**0,00034**	0,10	**0,55**	0,00064
3	0,85	1,86	0,00035	0,12	0,57	0,00091
4	0,99	8,88	0,00035	1,19	1,91	0,00088
5	**0,46**	4,11	0,00074	0,00076	1,38	0,00302
6	3,75	6,05	1,98	0,37	10,56	**0,00059**

Tabel 19 esitab päringute (S1-S5) ja andmemuudatus operatsioonide (U) kiiruste mõõtmistulemuste geomeertilise keskmise erinevate disainide ja andmehulk 2 (1 000 000 tellimuse) korral. Mõõtmistulemused on sekundites.

Tabel 19. Päringute ja andmemuudatus operatsioonide geomeetrilised keskmised kiirused (sekundites) andmehulga 2 korral.

Disain	**S1**	**S2**	**S3**	**S4**	**S5**	**U**
1	1,70	9,78	0,00052	**0,00084**	1,20	0,00142
2	1,75	**3,52**	**0,00035**	0,22	**0,85**	**0,00073**
3	1,61	4,08	0,00037	0,28	1,30	0,00095
4	2,28	26,65	0,00038	1,5	3,68	0,00099
5	**0,86**	7,88	0,00078	0,00094	1,54	0,00351
6	5,44	23,71	4,5	0,90	*	0,00078

** Ei õnnestunud saada serverilt 10 minuti jooksul vastust.*

Tabel 20 esitab päringute (S1-S5) ja andmemuudatus operatsioonide (U) kiiruste mõõtmistulemuste geomeertilise keskmise erinevate disainide ja andmehulk 3 (2 000 000 tellimuse) korral. Mõõtmistulemused on sekundites.

Tabel 20. Päringute ja andmemuudatus operatsioonide geomeetrilised keskmised kiirused (sekundites) andmehulk 3 korral.

Disain	**S1**	**S2**	**S3**	**S4**	**S5**	**U**
1	3,42	21,16	0,00056	**0,00098**	2,37	0,00165
2	3,26	**6,85**	**0,00040**	0,34	**1,13**	**0,00084**
3	3,12	7,31	0,00041	0,43	2,18	0,00105
4	3,61	35,53	0,00042	1,76	7,56	0,00115
5	**1,73**	15,83	0,00083	0,00111	2,83	0,00394
6	10,47	43,21	5,52	1,39	*	0,00086

** Ei õnnestunud saada serverilt 10 minuti jooksul vastust.*

Tabel 21 esitab päringute (S1-S5) ja andmemuudatus operatsioonide (U) keerukuse mõõtmistulemused erinevate disainide puhul.

Tabel 21. Päringute ja andmemuudatus operatsioonide keerukus erinevate disainide korral (koodiridade arv).

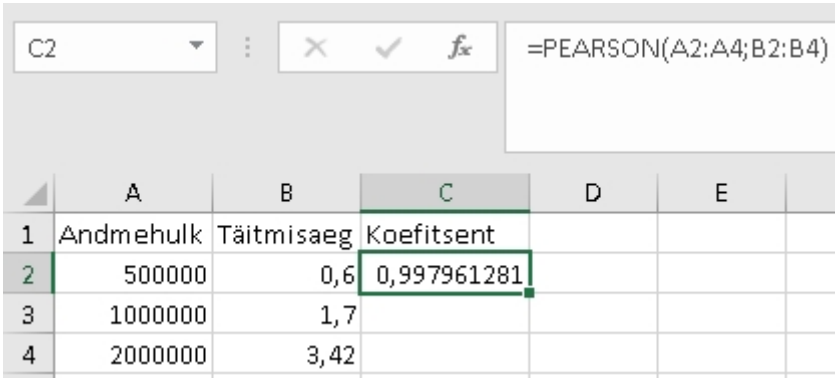
Disain	**S1**	**S2**	**S3**	**S4**	**S5**	**U**
1	4	6	4	**5**	**8**	**2**
2	4	11	10	24	26	3
3	4	11	10	24	26	3
4	4	**5**	**3**	24	24	**2**
5	**3**	17	21	18	21	6
6	8	34	27	44	46	**2**

Tabel 22 esitab seisundite lisamise (I) ja kustutamise (D) operatsiooni ning disainide realiseerimiseks vajalike lausete (C) keerukuse mõõtmistulemused.

Tabel 22. Seisundite lisamise ja kustutamis operatsioonide ning disainide realisatsiooni lausete keerukus erinevate disainide korral (koodiridade arv).

Disain	**I**	**D**	**C**
1	**2**	2	25
2	7	5	**21**
3	20	14	37
4	32	27	43
5	167	98	180
6	8	**1**	27

Tegin töökiiruse mõõtmisi erinevate andmehulkadega nägemaks, kuidas mõjutab andmehulga muutus töökiirust (täitmisaega). Leidsin iga disaini (1-6) ja ülesande (S1-S5;U) paari korral selle töökiiruse korrelatsiooni andmehulga suurusega (Joonis 34) ja esitan tulemused Tabel 23.



Joonis 34. Pearsoni korrelatsiooni koefitsiendi arvutamine Excelis (disaini 1 põhjal lahendatava ülesande S1 korral).

Tabel 23. Pearsoni korrelatsiooni koefitsent iga disaini-ülesande paari korral.

Disain ja ülesanne	**Pearsoni korrelatsiooni koefitsent**
1 S1	0,998
1 S2	0,998
1 S3	0,929
1 S4	0,974
1 S5	1
1 U	1
2 S1	1
2 S2	0,984
2 S3	0,982
2 S4	0,978
2 S5	0,991
2 U	1
3 S1	1
3 S2	0,997
3 S3	1
3 S4	0,978
3 S5	0,991
3 U	0,999
4 S1	0,984
4 S2	0,929
4 S3	0,994
4 S4	0,971
4 S5	1

Disain ja ülesanne	**Pearsoni korrelatsiooni koefitsent**
4 U	0,997
5 S1	1
5 S2	1
5 S3	0,992
5 S4	0,979
5 S5	0,973
5 U	0,974
6 S1	0,966
6 S2	0,987
6 S3	0,909
6 S4	0,977
6 S5	-
6 U	0,913

7 Tulemuste analüüs ja järeldused

Selles peatükis analüüsin eksperimendi põhjal saadud tulemusi ning teen nende põhjal järeldusi, mille põhjal täiendan ka disainide kataloogi.

7.1 Tulemuste analüüs

Selles jaotises analüüsin eksperimendi tulemuste põhjal andmebaaside andmemahte, andmete lugemise (S1-S5) ja andmemuudatus operatsioonide (U) täitmiskiirust ja keerukust. Samuti uurin seisundi lisamise (I), kustutamise (D) ja disainide realiseerimiseks vajalike lausete (C) keerukust. Lisaks vaatlen ka kõigi disaini-ülesande paaride jaoks leitud Pearsoni korrelatsiooni koefitsendi väärtusi.

7.1.1 Andmemahud

Andmemahtude omavahelisest võrdlusest (Tabel 16) selgus, et oma mahult on kõige väiksem *Seisundiklassifikaator* disainile loodud andmebaas. Samas ei ole oma mahult palju suurem ka *Temporaalsed veerud* disaini andmebaas. Kui vaadata samu mõõtmistulemusi ilma indeksiteta andmebaasidele (Tabel 17), siis on see järjekord vastupidine. Seega ei tulene erinevus sellest, et *Temporaalsed veerud* disaini puhul on tabeli *Tellimus* veergude arv palju suurem või see sisaldab palju NULLe, vaid hoopis loodud indeksitest. Nimelt loodi *Temporaalsed veerud* disaini jaoks igale seisundi veerule osaline indeks ning nagu eelnevalt sai öeldud, siis ka need kasvatavad andmemahtu.

Nendele kahele disainile järgnevad omavahel väga ligilähedaste tulemustega *Tõeväärtustüüpi veerud* ja *Vektorkodeerimine* disainide alusel loodud andmebaasid (Tabel 16). Nende kahe omavaheline erinevus tuleneb vektorkoodi sisaldavale veerule loodud indeksi kasvamisest, sest ilma indeksiteta on andmemahud võrdsed (Tabel 17). Võrreldes tulemusi eelnevate disainidega võib öelda, et erinevus *Tõeväärtustüüpi veerud* disainile tuleneb tabeli *Tellimus* kohustuslikest veergudest. Kuigi PostgreSQL kasutab iga tõeväärtuse salvestamiseks vaid ühe baidi, on deklareeritud kohustusliku kitsenduse tõttu selle disaini korral kõikides väljades väärtus [51]. Mittekohustuslikud (NULLe sisaldavad) veerud *Temporaalsed veerud* disaini puhul ei mõjuta andmemahtude suurenemist, sest kuni kaheksa veeruga tabelites ei kulutata PostgreSQL NULLide hoidmiseks üldse ruumi [44]. Erinevus *Vektorkodeerimine* disaini ja kõige väiksemate

mahtudega andmebaaside vahel tuleneb kohustusliku seisundi veeru andmetüübist TEXT [80]. Kasutades funktsiooni *pg_column_size()* leidsin, et iga seisundi väärtuse suuruseks on kuus baiti [81]. See on suurem võrreldes *Seisundiklassifikaator* välisvõtme veeruga, mis on tüüpi SMALLINT ja mille puhul iga väärtuse salvestamiseks kulub kaks baiti [82]. *Temporaalsed veerud* disaini korral on kasutusel TIMESTAMP tüüpi veerud. Kuigi iga sellise väärtuse hoidmiseks kulub kaheksa baiti (mis on suurem kui kuus baiti vektorkoodi puhul) [83], mängib siin jällegi rolli eelnevalt kirjeldatud asjaolu NULLide hoidmise kohta andmebaasis, mistõttu on *Vektorkodeerimine* disaini puhul andmemaht suurem kui *Temporaalsed veerud* korral.

Põhiolemitüübi alamtüüpidena ja *Seisundite tuletamine* disainid on oma andmemahult teistest juba palju suuremad. *Seisundite tuletamine* puhul on maht peaaegu kaks korda suurem kui kõige väiksemate andmemahtudega disainidel. Erinevus teiste disainidega tuleneb sellest, et lisaks põhitabelis *Tellimus* olevatele ridadele on mõlema disaini puhul tellimuste kohta andmeid ka teistes tabelites. Kui *Seisundite tuletamine* disaini puhul on ligikaudu 90% protsendi (kõik peale seisundis *Ootel*) tellimuste andmetest lisatud ka teistesse tabelitesse, siis *Põhiolemitüübi alamtüüpidena* disaini korral leidub iga tellimuse korral rida mõnes teises tabelis. Samas leidub iga tellimuse korral vastav rida ainult ühes alamtabelis, mitte kõikides. See on ka põhjuseks antud disainide omavahelisele erinevusele. Siiski ei tasu ära unustada, et *Seisundite tuletamine* disaini tabelites on palju lisainformatsiooni (nt terve seisundimuudatuse ajalugu) tellimuste kohta.

7.1.2 Päringute ja andmemuudatus operatsioonide kiirused

Esimese päringu (S1), millega leitakse ühes kindlas seisundis olevad tellimused, tulemuste põhjal selgus, et kõige kiiremini tagastatakse kõigi andmehulkade puhul vastus disaini *Põhiolemitüübi alamtüüpidena* korral. Põhjus on, et kõik otsitavad tellimuse numbrid on koos ühes tabelis ning andmebaasisüsteemil puudub vajadus üleliigseid ridu tulemusest eemaldada. Kuigi disainide *Seisundiklassifikaator*, *Tõeväärtustüüpi veerud*, *Temporaalsed veerud* ja *Vektorkodeerimine* korral päritakse andmed samuti ainult ühest tabelist, tuleb nende puhul süsteemil läbi viia lisaoperatsioon, leidmaks kõigi tellimuste hulgast just soovitud seisundis tellimused. Samas pole kõigi nende disainide puhul täitmiskiirused teineteisest märkimisväärselt erinevad kui võrrelda neid disaini *Seisundite tuletamine* tulemustega, mille puhul võtab täitmine palju rohkem aega. Erinevuse põhjus

disainile *Seisundite tuletamine* tuleneb vajadusest viia läbi andmete kontroll üle mitme tabeli.

Teise päringu (S2), millega leitakse kõik tellimused, mis pole ühes kindlas seisundis, tulemustest selgus, et kõige kiiremini täidetakse antud ülesanne *Temporaalsed veerud* ja *Tõeväärtustüüpi veerud* disainide puhul. Üksteisele väga lähedane tulemus tuleb sellest, et laused, mida täidetatakse andmebaasides on oma ülesehituselt samasugused ning ka nende disainide omavahelisest sarnasusest. Lõin mõlema disaini seisundiveergudele otsingute kiirendamiseks osalised indeksid. Päringu täitmiskiirus ülejäänud disainidele on kõikide andmehulkade juures juba esimestest palju suurem. Põhjuseks on nii vajadus pärida andmeid mitmes tabelist, mistõttu tuleb läbi viia ühendi leidmise operatsioone kui ka funktsiooni kasutamisest. Antud päringu korral tuleb disaini *Vektorkodeerimine* puhul kasutada funktsiooni lausa kahekordselt. Seda kasutatakse nii selleks, et leida otsitavale seisundile vastavat vektorkoodi kui ka kõigile leitud tellimustele seisundi nimetuste leidmiseks.

Kolmanda päringu (S3), millega leitakse konkreetse olemi hetkeseisund, tulemustest selgus, et kõige kiiremini täidetakse antud ülesanne disainide *Temporaalsed veerud*, *Tõeväärtustüüpi veerud* ja *Vektorkodeerimine* puhul. Kõigi nende disainide puhul peab andmebaasisüsteem lugema ainult ühte tabelit ning kasutab rea leidmiseks tabeli primaarvõtmele automaatselt loodud indeksit. *Seisundiklassifikaator* disaini puhul jäi täitmiskiirus juba esimestest maha, sest hetkeseisundi nime leidmiseks tuleb läbi viia ühendamisoperatsiooni klassifikaatori ja põhitabeli vahel, mis päringu täitmist aeglustab. Samal põhjusel on teistest natukene aeglasem ka *Põhiolemitüübi alamtüüpidena* disain, kus peab läbi vaatama kõik alamtabelid leidmaks otsitava tellimuse kohta käiva rea. Päringu täitmiskiirus disainile *Seisundite tuletamine* on kõikide andmehulkade juures juba palju väiksem, sest peab kontrollima erinevates tabelites eksisteerivaid andmeid ärireeglitest tulenevate tingimuste alusel.

Neljanda päringu (S4), mis otsib seisundeid, milles pole ühtegi olemit, tulemuste põhjal selgus, et oluliselt kiiremini täidetakse antud ülesanne *Seisundiklassifikaator* ja *Põhiolemitüübi alamtüüpidena* disainide korral. Antud disainide omavaheline erinevus tuleb sellest, et *Põhiolemitüübi alamtüüpidena* disaini puhul tuleb leida ühend üle viie tabeli samas kui *Seisundiklassifikaator* disaini korral loetakse ühte tabelit ning korreleeruva alampäringu abil teist [47]. Samas näitavad mõõtmistulemused, et tabelite

ühendi leidmine ei ole antud juhul üldse nii kulukas operatsioon kui võiks arvata. Nimetatud disainidele järgnevad jällegi väga sarnaste tulemustega disainid *Tõeväärtustüüpi veerud* ja *Temporaalsed veerud*. Selgelt kõige aeglasem täitmine on *Seisundite tuletamine* ning *Vektorkodeerimine* disainide korral.

Viienda, koondandmete päringu (S5), mis leiab igale seisundile vastava tellimuste koguarvu, tulemustest selgus, et nagu ka enamike eelmiste päringute puhul tagastatakse vastus kõige kiiremini disaini *Temporaalsed veerud* puhul. Samas ei jää sellest oluliselt maha disainid *Tõeväärtustüüpi veerud, Seisundiklassifikaator* ja *Põhiolemitüübi alamtüüpidena*. Antud disainide puhul kasutatavate lausete erinevus tuleneb ühendi (UNION) leidmisest ja välisühendamisest (LEFT JOIN). Siiski, kui võiks arvata, et LEFT JOIN operatsiooni täitmine võtab andmebaasisüsteemil aega, siis tegelikult see antud töö tulemuste põhjal nii ei ole. *Temporaalsed veerud* ja *Tõeväärtustüüpi veerud* disaini puhul kasutatakse samuti ühendi leidmist, kuid tänu ühisele tabeli avaldisele ja FILTER klauslile loetakse ühe tabeli läbimisega kokku erinevates seisundites olevate tellimuste arv ning teise sammuna pannakse saadud väikese andmehulga pealt ühendi leidmist kasutades kokku lõpptulemus. *Põhiolemitüübi alamtüüpidena* disaini korral suudab andmebaasisüsteem ühendi leidmist sisaldava lause kiiresti täita, kuna alamtabelitest loetakse andmeid paralleelselt [84]. *Vektorkodeerimine* korral oli täitmine juba aeglasem, sest selle puhul näeb lause ette ühe tabeli korduvat lugemist ning andmebaasisüsteem selleks paralleeltööd ei tee. Kõige aeglasemaks osutus lause *Seisundite tuletamine* puhul, sest kahe andmehulga korral kolmest ei andnud servel kümne minuti jooksul vastust. Ka selle lause puhul tuleb leida erinevate alampäringute ühend, kuid võrreldes eelmistega on nendes alampäringutes palju keerukamad otsingutingimused, mis võivad ka hõlmata andmete lugemist mitmest tabelist.

Andmemuudatuse operatsioonina (U) viidi läbi test, kus muudeti ühe tellimuse seisundit. Eksperimendi tulemustest selgus, et kõige kiiremini täidetakse antud ülesanne *Temporaalsed veerud* ja *Seisundite tuletamine* disainide korral. *Temporaalsed veerud* disaini korral tuleb kustutada väärtus eelnevale seisundile vastavas väljas (muuta see NULLiks) ja väärtustada uuele seisundile vastav väli. Ühtegi trigerit see ei käivita. *Seisundite tuletamine* puhul võib põhjuseks olla, et muudatus tehakse tabelis, kus pole kõikide tellimuste kohta andmeid ja seega tuleb süsteemil muudetava rea leidmiseks teha vähem tööd. Samuti tehakse selle disaini korral muudatus ühes väljas ning muudatus ei käivita ka trigerit. Ülejäänud disainide puhul käivitab muudatus kasutaja loodud trigeri

(*seisundiklassifikaator, toevaartustuupi_veerud, vektorkodeerimine, pohiolemituubi_alamtuupidena*), kutsub välja kasutaja loodud funktsiooni (*vektorkodeerimine),* nõuab lisamise ja kustutamise operatsiooni läbiviimist ühe tehingu e transaktsioonina (*pohiolemituubi_alamtuupidena*) või nõuab süsteemilt välisvõtme deklaratsioonist tulenevalt viidete terviklikkuse kontrollimist (*pohiolemituubi_alamtuupidena, seisundiklassifikaator*). Kõik see võtab aega. Oodatult on *pohiolemituubi_alamtuupidena* kõige aeglasem, sest muuta tuleb andmeid kahes tabelis ja lisaks käivitub triger.

7.1.3 Lausete keerukus

Esimese päringu (S1) koodiridade arv on kõige väiksem disaini *Põhiolemitüübi alamtüüpidena* korral, sest leidmaks ühes kindlas seisundis olevaid olemeid tuleb väljastada kõik read sellele seisundile vastavast tabelist. Samas suuri erinevusi disainide vahel ei ole, sest peale *Seisundite tuletamine* disaini, on ülejäänud disainide koodiridade arvuks 4. *Seisundite tuletamine* disainile vastav arv on teistest ligi kaks korda suurem, sest andmete kontroll tuleb läbi viia mitme tabeli üleselt.

Teise päringu (S2) puhul erinevad tulemused teineteisest juba suurelt. Kõige väiksem tulemus kuulub disainile *Vektorkodeerimine* ning sellele järgneb koheselt *Seisundiklassifkaator* disain. Märkimisväärne koodiridade arvu erinevus teiste disainidega seisneb selles, et antud disainide puhul ei ole lisatud päringutesse loogikat seisundi nimetuste tuletamiseks. *Vektorkodeerimine* puhul kasutatakse selleks funktsiooni ning *Seisundiklassifikaator* puhul päritakse see klassifikaatori tabelist.

Nagu ka teise päringu puhul on kolmanda päringu (S3) keerukuse tulemused teineteisest üpris erinevad. Jällegi kuulub kõige väiksem tulemus disainile *Vektorkodeerimine* ning sellele järgneb disain *Seisundiklassifikaator*. Nende kahe omavaheline erinevus tuleneb sellest, et disaini *Seisundiklassifikaator* puhul tuleb ühendada põhitabel klassifikaatori tabeliga leidmaks sellele olemile vastava seisundi nimetust. Suure erinevuse põhjuseks ülejäänutega on endiselt see, et nimetatud kaks päringut ei sisalda seisundite tuletamise või nimetuse arvutamise loogikat.

Nii neljanda päringu (S4) kui viienda päringu (S5) tulemuste põhjal selgus, et kõige vähem keerukad päringud kuuluvad mõlema korral disainile *Seisundiklassifikaator*. Koodiridade arvu erinevus teiste disainidega on seal juba mitmekordne. Põhjuseks on see,

et kui *Seisundiklassifikaator* disaini puhul tuleb ühendada omavahel ühekordselt ainult kaks tabelit, siis ülejäänute puhul viiakse mitmekordselt läbi ühendi leidmist (üks kord iga erineva seisundi kohta).

Andmemuudatus operatsiooni (U) päringute keerukuses suuri erinevus ei ilmne. Lihtsuse mõttes esimest kohta, kaherealise lausega, jagavad disainid *Vektorkodeerimine*, *Seisundiklassifikaator* ja *Seisundite tuletamine*. Erinevus teistest seisneb selles, et andmeid tuleb muuta ainult ühes veerus, kuid *Temporaalsed veerud* ja *Tõeväärtustüüpi veerud* puhul tuleb teha muudatus kahes veerus. *Põhiolemitüübi alamtüüpidena* disaini korral tuleb muudatus teha lausa kahes tabelis.

Seisundite lisamise operatsiooni (I) puhul kuulub kõige väiksema koodiridade arvuga lahendus disainile *Seisundiklassifikaator*. Erinevuse teistest disainidest põhjustab asjaolu, et antud disaini korral puudub vajadus peale uue väärtuse lisamist klassifikaatori tabelisse teha muudatus kas tabeli struktuuris, mõnes kontrollkitsenduses või funktsioonis. Seevastu väljendub see vajadus eriti suurelt disaini *Põhiolemitüübi alamtüüpidena* korral, kus lisaks uue seisundi tabeli ning trigeri loomisele tuleb teha muudatus ka kuues olemasolevas trigeris.

Seisundi eemaldamise operatsiooni (D) korral, selgus tulemustest, et koodiridade arvult kõige lühem lahendus kuulub disainile *Seisundite tuletamine*. Sellele järgneb disain *Seisundiklassifikaator*. Jällegi on põhjuseks, et muudatused, mida tuleb antud disainide puhul läbi viia kas andmetes endas, tabeli struktuuris, mõnes kontrollkitsenduses või funktsioonis, on minimaalsed. Seevastu *Põhiolemitüübi alamtüüpidena* disaini puhul tuleb läbi viia suurel hulgal muudatusi, sest kuigi tabeli kustutamiseks vajalike koodiridade arv on väiksem kui uue tabeli loomiseks vajalike koodiridade arv, tuleb siiski teha muudatusi ka kõigis olemasolevates trigeris.

Disainide realiseerimiseks vajalike lausete (C) omavahelisest võrdlusest selgus, et kõige väiksem koodiridade arv kuulub disainile *Temporaalsed veerud*. Samas ei ole koodiridade arv oluliselt suurem ka sellelele pingereas järgnevate disainide korral: *Seisundiklassifikaator*, *Vektorkodeerimine*, *Seisundite tuletamine*, *Tõeväärtustüüpi veerud*. Erinevus kõige esimese ja nende nelja järgneva vahel tuleneb peamiselt sellest, et *Temporaalsed veerud* korral ei looda eraldi trigerit, mille tulemusel väärtustatakse automaatselt *seisundimuudatuse_aeg*. Teistest mitmeid kordi suurem koodiridade arv

kuulub jällegi *Põhiolemitüübi alamtüüpidena* disainile, sest lisaks mitmete tabelite loomisele tuleb luua igale tabelile triger, mis viib läbi andmekontrolle vastavalt töös esitatud nõuetele (vt jaotis 4.1).

7.1.4 Pearsoni korrelatsiooni koefitsent

Eksperimendi lõpus leidsin Pearsoni korrelatsiooni koefitsendi igale disaini-andmekäitluse ülesande (edaspidi disaini-ülesande) paarile, võrreldes omavahel erinevatele andmehulkadele vastavaid mõõtmistulemusi. Selle eesmärgiks on hinnata iga disaini-ülesande paari korral andmehulga suurenemise ja kas töökiiruse vähenemise vahel on lineaarne sõltuvus või mitte [67]. Definitsioon Pearsoni korrelatsiooni koefitsendile ütleb, et kaks muutujat on omavahel kasvavas positiivses seoses, kui selle väärtus jääb 0-1 vahele. Arvu 1 puhul korreleeruvad uuritavad tunnused teineteisega täielikult. Kui korrelatsioonikordaja on alla 0, siis tähendab see kahanevat seost kahe erineva väärtuse vahel. Arvu 0 korral puudub lineaarne korrelatsioon täielikult [68]. Sotsiaalteaduste puhul peetakse küllaltki tugevaks seoseks juba koefitsenti, mille väärtus on üle 0,5. Reaalteaduste puhul on tugeva seose piir kaugemal. Samas öeldakse siiski, et mida lähemal on arv ühele, seda tugevam seos tulemuste vahel eksisteerib [85]. Kui uurida lisaks veel mõnda materjali antud teemal, siis võib leida, et kui koefitsent on üle 0,7, siis on tegemist tugeva lineaarse seosega [86]. Antud töö eksperimendi tulemustest selgus, et kõigi disaini-ülesande paaride puhul jäi koefitsendi väärtus alati vahemiku 0,909-1. Seega võib tuginedes Pearsoni korrelatsiooni koefitsenti definitsoonile ning erinevates materjalidest leitud infole öelda, et igale töös vaadeldavale disaini-ülesande paarile eksisteerib tugev lineaarne sõltuvus täitmisaja suurenemise (e töökiiruse vähenemise) ja andmehulga suurenemise vahel.

7.2 Järeldused

Eksperimendi tulemuste analüüsi põhjal saab öelda, et andmehulga kasvades kasvas lineaarselt ka kõikidele päringutele kuluv aeg. Siiski ei mõjutanud see läbi viidud eksperimendite tulemuste järjekorda.

Seisundiklassifikaator disaini selgete eelistena ilmnesid suur täitmiskiirus ja väike keerukus koondandmete leidmise päringus ja ühegi tellimuseta seisundite leidmise päringus. Lisaks saab ära mainida teistest oluliselt lihtsama uue seisundi lisamise või olemasoleva eemaldamise andmebaasist. Samuti kuulusid antud disainile ka teistest

mõnevõrra väiksemad andmemahud. Samas selgus, et päringute puhul, kus tuleb leida kõik tellimused, mis ei ole ühes kindlas seisundis ja tagastada nende hetkeseisund, olid antud disainile vastavad tulemused enamikest oluliselt aeglasemad.

Temporaalsed veerud disaini tugevusena selgus, et vastus tagastatakse kiirelt iga katsetatud päringu korral – selle disaini täitmiskiirused olid parimad kolme väga erineva päringu puhul. Lisaks võib öelda, et tegmist on disainiga, mille korral on andmemahud väiksemad kui enamus teiste disainide korral. Kui eemaldada veel sellele disainile loodud indeksid, siis on tulemus võrdne kõige väiksema andmemahuga *Seisundiklassifikaator* disainiga. Väikeste andmemahtude põhjus on PostgreSQL sisemises andmete salvestamise viisis, mille korral ei kuluta süsteem kuni kaheksa veeruga tabelis NULLide salvestamiseks ruumi. Kui tabelis on üle kaheksa veeru (üheks põhjuseks võib olla suurem seisundite hulk), siis see eelis kaob. Samuti leidsin, et antud disaini eeliseks on seisundimuudatuse operatsiooni läbiviimise kiirus ja disaini realiseerimiseks vajalike lausete vähene keerukus. Negatiivse poole pealt selgus, et koondandmete päring on oluliselt keerukam kui *Seisundiklassifikaator* puhul.

Disaini *Tõeväärtustüüpi* disaini puhul selgus, et nagu ka eelmise disaini korral olid töökiirused head nii koondandmete leidmisel kui ka konkreetse olemi seisundi otsimisel. Andmemuudatuse operatsiooni täitmiskiiruse ja andmemahtude mõõtmistulemused olid keskpärased. Negatiivsete asjaoludena ilmnesid uue seisundi lisamiseks või olemasoleva eemaldamiseks vajalike lausete ning koondandmete leidmiseks vajalike lausete keerukus. Lisaks oli antud disaini realiseerimiseks vajalike koodiridade arv suhteliselt suur.

Vektrokodeerimine disaini puhul selgus, et selle suureks nõrkuseks on enamike päringute väike täitmiskiirus võrreldes teiste disainidega. Kiire oli see vaid juhul kui oli vaja leida mingi kindla olemi seisundit. Soovitan vähemalt operatiivandmetega andmebaasides sellist tüüpi disaini vältida. Siiski andis funktsiooni kasutamine päringutes väikse eelise koodi keerukuse osas. Seda eriti päringu puhul kus tuleb leida konkreetse olemi seisund – osade disainidega olid erinevused mitmekordsed. Andmemahtude mõõtmistulemused olid pigem keskpärased. Põhjuseks on vektorkoodi sisaldavale veerule loodud indeks, mille kasv sõltus otseselt andmemahu suurenemisest. Antud disaini realiseerimiseks ning ka uue seisundi lisamiseks või olemasoleva eemaldamiseks vajalike lausete koodiridade arvud olid oma tulemustelt pigem suured. Seevastu SQL kood, mida kasutasin tellimuse seisundimuudatuse läbi viimiseks, oli väga lihtne.

Päringu täitmiskiiruste analüüsist selgus, et kui on vajadus leida kõiki mingis kindlas seisundis olemeid, siis tuleks eelistada disaini *Põhiolemitüübi alamtüüpidena*. Selle puhul tuleb taolise päringu täitmiseks lugeda andmeid ainult ühest tabelist ilma, et andmebaasisüsteem peaks kulutama aega üleliigsete ridade tulemusest eemaldamisele. Samas oma andmemahult on antud disain teistest juba mõnevõrra suurem, sest olemi kohta andmeid esitavaid ridu leidub mitmes tabelis. Sellel samal põhjusel tuleb koondandmete päringu puhul läbi viia mitmeid ühendi leidmise operatsioone, mis muudab päringu keerukamaks. Samuti osutus teistest disainidest oluliselt aeglasemaks tellimuse seisundimuudatuse operatsioon. Lisaks on sellel disainil võrreldes teiste disainidega kordades suurem disaini realiseerimiseks, uue seisundi lisamiseks või olemasoleva eemaldamiseks vajalike koodiridade arv. Võrreldes teiste disainidega on kordades keerukam ka tellimuse seisundimuudatuseks läbiviidav operatsioon.

Disaini *Seisundite tuletamine* selgeks nõrkuseks on igat tüüpi päringute täitmiskiirus ja keerukus. Nagu ka disaini *Vektorkodeerimine* puhul on mõõtmistulemused teistest märkimisväärselt suuremad. Põhjuseks on, et vastavalt ärireeglitele tuleb kasutada mitmeid tabeleid. Lisaks selgus mõõtmistulemustest, et antud disaini puhul on andmemaht enamikest teistest pea kaks korda suurem, sest nagu disaini *Põhiolemitüübi alamtüüpidena* puhul, leidub ka siin olemitele vastavaid andmeid mitmes tabelis. Samas ei tohi ära unustada, et võrreldes teiste disainidega hoitakse selle puhul tabelites olemite kohta palju rohkem infot. Näiteks on tabelitest leitav igale tellimusele kogu seisundimuudatuste ajalugu. Positiivsete asjaoludena selgus, et antud disaini puhul on teistest vähem keerulisem eemaldada olemasolevat seisundit ja muuta tellimuse seisundit. Samas on tehtavad operatsioonid täielikus seoses ärireeglitega, mistõttu mõne teise seisundi lisamisel või eemaldamisel tuleks läbi viia hoopis teistsugune tegevuste jada.

Kokkuvõtvalt võib öelda, et antud eksperimendi tulemuste põhjal eristusid selgelt iga disaini puhul selle eelised ja puudused. Enim eeliseid leidsin disainidel *Seisundiklassifikaator* ja *Temporaalsed veerud*, millele järgnesid keskpäraste tulemustega *Tõeväärtustüüpi veerud* ja *Põhiolemitüübi alamtüüpidena*. Samas leidsin, et ühe tabeliga disainidele vastavad eksperimendi tulemused olid enamasti kõikide võrdluste eesotsas. Nende puhul ei esinenud omavahel suuri erinevusi ei päringute täitmiskiiruste ega ka keerukuse osas. Kuigi enim negatiivseid asjaolusid tuli esile disainidele *Seisundite tuletamine* ja *Vektorkodeerimine*, ei tähenda see veel, et antud disainid tuleks koheselt valikust kõrvaldada, sest ka nende kasutamises leidub positiivset.

Näiteks kui süsteemianalüüs selgitab, et süsteemi peamisteks ülesanneteks on konkreetse olemi seisundi leidmine ja selle nimetuse kuvamine või kui selles toimuvad sagedased seisundimuudatused, siis võiks eelistada teistele just neid.

Kokkuvõttes võib öelda, et kõigil kataloogis toodud disainidel on piisavalt positiivseid külgi, et neid võiks ka mustriteks kutsuda.

8 Kokkuvõte

Infosüsteemi põhiolemite seisundi kohta andmete haldamine on enamike andmebaasirakenduste jaoks tavapärane funktsionaalsus. Seega on vaja teada kuidas oleks kõige parem selliseid andmeid andmebaasides esitada. 2019. aasta kevade seisuga on SQL-andmebaaside loomiseks mõeldud andmebaasisüsteemid endiselt populaarsuselt esirinnas. [3] Kuna SQL andmebaasikeelel põhinevad andmebaasisüsteemid on olnud turul juba aastakümneid, siis on välja mõeldud ka väga mitmeid lahendusi SQL-andmebaasides olemi seisundite haldamiseks [14]. Samas ei ole ma leidnud allikat, kus neid kõiki oleks üheskoos ja struktureeritud viisil kirjeldatud ning seejuures omavahel läbi praktiliste katsetuste võrreldud.

Antud magistritöö eesmärgiks oli väljaselgitada võimalikud disainid infosüsteemi põhiobjektide seisundite registreerimiseks SQL-andmebaasides, panna need kirja ühtset struktuuri kasutades ning uurida nende headust erinevate päringute ja andmemuudatuste kontekstis. Selle tulemusel valmis disainide kataloog, mille saavad andmebaasi kavandajad võtta aluseks oma valikute põhjendamiseks. Alustasin tööd vastavateemalise informatsiooni otsimisest erinevatest allikatest, millele tuginedes koostasin peatüki varasematest uuringutest (vt peatükk 3) ning panin kokku disainide kataloogi (vt peatükk 4). Kuna mõnikord vastab ühele üldlahendusele palju variatsioone (näiteks on erinevus ainult seisundi veeru andmetüübis), siis valisin kataloogi koostamisel põhilahenduseks selle disaini, mille kasutamises leidsin enim positiivset. Oma olemuselt väga sarnased lahendused lisasisin antud disaini variatsioonide alla. Sel viisil esitasin kokku kuus erinevat disaini. Töö teises osas kavandasin ja viisin läbi eksperimendi näite *Tellimus* (vt jaotis 4.1) alusel. Praktiliste katsetuste läbiviimiseks valisin andmebaasisüsteemi PostgreSQL (11) tänu selle populaarsusele, võimalusterohkusele ning heale kinnipidamisele SQL standardist [3]. Eksperimendi käigus võrdlesin kõiki kataloogis esitatud põhidisaine nii päringute ja andmemuudatus operatsioonide täitmiskiiruse ja keerukuse, seisundite hulga muutmise keerukuse kui ka andmemahtude alusel. Enne seda projekteerisin kõigile disainidele andmebaasid ning täitsin need samasuguste omadustega ning hetkeseisundiga tellimuste kohta käivate testandmetega. Viisin kõik testid läbi kolme erineva andmehulga juures selgitamaks iga töös vaadeldava disaini korral välja, kas töökiiruse vähenemise ja andmehulga suurenemise vahel on lineaarne sõltuvus või mitte. Analüüsisin eksperimendi tulemuste põhjal kõiki disaine.

Kogu töös kasutatav SQL-kood on lisatud formaaditud kujul GitHub'i, mis on internetis kättesaadav aadressil https://github.com/susannapeek1/seisundid_SQL.

Eksperimendi tulemuste põhjal selgus, oodatult, et kõigi töös esitatavate disainide kasutamisele leidub eeliseid ja puuduseid. Seega saab öelda, et disaini valik on otseselt sõltuv konkreetsest olukorrast ja täidetavast ülesandest. Samas tuleb mõista, et ükski disain ei saa sobida kõigeks. Seetõttu on oluline juba alguses paika panna, millised on loodava süsteemi suhtes esitatud nõuete prioriteedid. Siiski selgus töös tehtud analüüsi põhjal, et mõnevõrra paremad tulemused kuulusid disainidele *Seisundiklassifikaator* ja *Temporaalsed veerud*. Seega kui eelistate väiksemaid andmemahtusid ning vähemkeerukat SQL koodi, siis võiks vaadata antud disainide poole. Samas, kui andmebaasirakenduse üheks olulisemaks funktsionaalsuseks on kõigi konkreetses seisundis olevate olemite leidmine, siis on selle päringu täitmine teistest oluliselt kiirem disaini *Seisundite esitamine põhiolemitüübi alamtüüpidena* korral. Kui huvitab päringu lihtsus ja hea töökiirus konkreetse olemi seisundi leidmisel, siis sobiks selleks kõige paremini disain *Vektorkodeerimine*. Disaini *Tõeväärtustüüpi veerud* puhul on päringute töökiirus üsna hea kõigi katsetatud päringute korral. Disaini *Seisundite tuletamine olemite ja seoste kohta registreeritud andmetest* testimisel täheldasin, et selle selgeteks puudusteks on suuremad andmemahud ja aeglasemad päringud. Samas ei tohi unustada, et antud disaini alusel loodud andmebaas sisaldab võrreldes teiste disainidega palju rohkem infot olemi seisundite kohta (näiteks terve seisundimuudatuse ajalugu). Nagu näha, siis erinevaid soovitusi saab anda palju. Seega on oluline, et andmebaasi kavandajad paneksid juba varakult paika neile olulised disainide valikukriteeriumid. Käesolev magistritöö võiks aidata disainide nendele kriteeriumitele vastavust hinnata.

Leian, et saavutasin püstitatud eesmärgid. Ma leidsin ning kirjeldasin hulga disaine ning uurisin nende headust mõningates mõõdetavates aspektides. Töö tulemina koostasin kataloogi, millele on võimalik tugineda uute tarkvarasüsteemide loomisel ning olemasolevate parandamisel ja edasiarendamisel. Samas tuleb seejuures meeles pidada, et uute tehnoloogiate tulekutega muutuvad ka kasutatavad praktikad. Seetõttu oleks töö üheks edasiuurimise võimaluseks laiendada seda ka NoSQL andmebaasisüsteemidele, mis viimasel kümnekonnal aastal on üha rohkem tähelepanu saanud. Samuti tasub eraldi uurimist, kuidas mõjutaks töökiirust erinevate disainide puhul see kui kasutada PostgreSQL poolt pakutavat tabelite andmebaasi sisemisel tasemel sektsioonideks jagamist. Kolmandaks võimalikuks edasiarenduseks on kasutada antud töös esitatud

disaine, võrdlemaks, kui keeruline on ehitada nende peale rakendust mingis kindlas programmeerimiskeeles. Ka sellise uuringu põhjal saaks teha täiendusi disainide kataloogis. Neljandaks võimaluseks on parima disaini valimine konkreetse süsteemi jaoks kasutades selleks näiteks Saaty analüütiliste hierarhiate meetodit. Nõuded sellele süsteemile määravad hindamiskriteeriumite suhtelise olulisuse. Antud töös toodud mõõtmistulemused saaksid olla aluseks disainide võrdlemisele nende kriteeriumite suhtes.

Kasutatud kirjandus

[1] E. Eessaar, „Teema 7. Andmebaaside projekteerimine: strateegiline analüüs ja detailanalüüs,“ 2014. [Võrgumaterjal]. Available: https://maurus.ttu.ee/ained/IDU0220_2014/doc/3/Teema_IDU0220_7_2014.pdf. [Kasutatud 30 04 2019].

[2] „UML Protocol State Machine Diagrams,“ UML-Diagrams, 2018. [Võrgumaterjal]. Available: https://www.uml-diagrams.org/protocol-state-machine-diagrams.html#protocol-state-machine. [Kasutatud 06 05 2019].

[3] „DB-Engines Ranking,“ 2019. [Võrgumaterjal]. Available: https://db-engines.com/en/ranking. [Kasutatud 25 03 2019].

[4] „Artifact-centric business process model,“ Wikipedia, 2019. [Võrgumaterjal]. Available: https://en.wikipedia.org/wiki/Artifact-centric_business_process_model. [Kasutatud 30 04 2019].

[5] M. Estañol, A. Queralt, M.-R. Sancho ja E. Teniente, „Artifact-Centric Business Process Models in UML,“ 2013. [Võrgumaterjal]. Available: https://www.researchgate.net/publication/278662445_Artifact-Centric_Business_Process_Models_in_UML. [Kasutatud 06 05 2019].

[6] A. M. S. P. J. R. S. Hevner, „Design science in information systems research,“ *MIS Quarterly*, kd. 28, nr 1, 2004.

[7] T. Mikli, Sissejuhatus infosüsteemidesse, Tallinn : Tallinna Tehnikaülikooli Kirjastus, 1998.

[8] „[EKSS] "Eesti keele seletav sõnaraamat",“ Eesti Keele Instituut, 2009. [Võrgumaterjal]. Available: http://www.eki.ee/dict/ekss/ekss.html. [Kasutatud 30 04 2019].

[9] E. Eessaar, Andmebaaside projekteerimine, Tallinn: Tallinna Tehnikaülikooli Kirjastus, 2008.

[10] „Zachman Framework,“ Wikipedia, 2019. [Võrgumaterjal]. Available: https://en.wikipedia.org/wiki/Zachman_Framework. [Kasutatud 30 04 2019].

[11] L. Silverston ja P. Agnew, „The Data Model Resource Book, Volume 3: Universal Patterns for Data Modeling,“ Chichester, John Wiley & Sons Ltd, 2009.

[12] J. Sanz, „Entity-Centric Operations Modeling for Business Process Management: A Multidisciplinary Review of the State-of-the-Art,“ %1 *Proceedings of 2011 IEEE 6th International Symposium on Service Oriented System (SOSE)*, 2011.

[13] „Ühtne modelleerimiskeel,“ Wikipedia, 2019. [Võrgumaterjal]. Available: https://et.wikipedia.org/wiki/%C3%9Chtne_modelleerimiskeel. [Kasutatud 30 04 2019].

[14] „SQL,“ Wikipedia, 2019. [Võrgumaterjal]. Available: https://en.wikipedia.org/wiki/SQL. [Kasutatud 30 04 2019].

[15] S. M. Mikk, „Graafide esitamine SQL-andmebaasides,“ Tallinna Tehnikaülikool, Tallinn, 2017.

[16] L. Silverston, „The Data Model Resource Book, Volume 1: A Library of Universal Data Models for All Enterprises,“ New York, John Wiley & Sons Inc, 2001.

[17] L. Silverston, „The Data Model Resource Book, Volume 2: A Library of Universal Data Models by Industry Types,“ New York, John Wiley & Sons Inc, 2001.

[18] D. C. Hay, „Enterprise Model Patterns: Describing the World (UML Version),“ Technics Publications, 2011.

[19] B. Karwin, SQL Antipatterns: Avoiding the Pitfalls of Database Programming, Pragmatic Bookshelf, 2010.

[20] E. Eessaar, „Mustrid,“ TTÜ: Andmebaasid I / Andmebaasid II , 2018.

[21] A.-S. Vellemaa, „Mõned disainimustrid klassifikaatorite esitamiseks SQL andmebaasides,“ Tallinna Tehnikaülikool, Tallinn, 2015.

[22] S. Hoberman, Data Modeler's Workbench: Tools and Techniques for Analysis and Design, New York: John Wiley & Sons, Inc., 2002.

[23] S. Peek, „Denormaliseerimise praktika uurimine ühe SQL-andmebaasi näitel,“ Tallinna Tehnikaülikool, Tallinn, 2016.

[24] N. Rõžova, „Disainimustrid üldistusseoste realiseerimiseks,“ Tallinna Tehnikaülikool, Tallinn, 2011.

[25] M. Ossipova, „Mitme väite ühe andmeväärtusena esitamise eelised ja puudused SQL-andmebaasides,“ Tallinna Tehnikaülikool, Tallinn, 2016.

[26] „Finite State Machine Patterns,“ [Võrgumaterjal]. Available: http://community.wvu.edu/~hhammar/rts/adv%20rts/statecharts%20patterns%20papers%20and%20%20examples/yacoub-ammar%20fsm%20ch%2010.pdf. [Kasutatud 5 3 2019].

[27] „How to store statuses in DB,“ Stack Overflow, 2011. [Võrgumaterjal]. Available: https://stackoverflow.com/questions/8281106/how-to-store-statuses-in-db. [Kasutatud 01 05 2019].

[28] „MySQL database schema design and relationships,“ Stack Overflow, 2017. [Võrgumaterjal]. Available: https://stackoverflow.com/questions/45690943/mysql-database-schema-design-and-relationships. [Kasutatud 01 05 2019].

[29] „How to handle status columns in designing tables,“ Stack Exchange, 2013. [Võrgumaterjal]. Available: https://softwareengineering.stackexchange.com/questions/214940/how-to-handle-status-columns-in-designing-tables. [Kasutatud 17 03 2019].

[30] „How to design database for storing object states in mysql?,“ StackExchange, 2018. [Võrgumaterjal]. Available: https://dba.stackexchange.com/questions/206447/how-to-design-databse-for-storing-object-states-in-mysql. [Kasutatud 01 05 2019].

[31] „Software Design Pattern,“ Wikipedia, 2019. [Võrgumaterjal]. Available: https://en.wikipedia.org/wiki/Software_design_pattern. [Kasutatud 5 3 2019].

[32] „Design Patterns,“ Wikipedia, 2019. [Võrgumaterjal]. Available: https://en.wikipedia.org/wiki/Design_Patterns. [Kasutatud 5 3 2019].

[33] „Rule Of Three,“ 2011. [Võrgumaterjal]. Available: http://wiki.c2.com/?RuleOfThree. [Kasutatud 5 3 2019].

[34] K. Krönström, „Hierarhiliste andmete esitamine SQL-andmebaasides kolme disainilahenduse näitel,“ TTÜ Informaatikainstituut, Tallinn, 2015.

[35] M. Fowler, Patterns of Enterprise Application Architecture, Addison-Wesley Professional, 2002.

[36] „Algoritmiline mõtlemine - mis see on?,“ 21 02 2017. [Võrgumaterjal]. Available: http://kristiproge.blogspot.com/2017/02/algoritmiline-motlemine.html. [Kasutatud 04 04 2019].

[37] „Customize the order status page,“ Shopify Help Center, [Võrgumaterjal]. Available: https://help.shopify.com/en/manual/orders/status-tracking/customize-status-tracking. [Kasutatud 5 3 2019].

[38] „Managing Orders,“ WooCommerce, [Võrgumaterjal]. Available: https://docs.woocommerce.com/document/managing-orders/. [Kasutatud 3 5 2019].

[39] „Order Statuses,“ BigCommerce Help Center, [Võrgumaterjal]. Available: https://support.bigcommerce.com/s/article/Order-Statuses. [Kasutatud 5 3 2019].

[40] „CREATE DOMAIN,“ The PostgreSQL Global Development Group, [Võrgumaterjal]. Available: https://www.postgresql.org/docs/9.5/sql-createdomain.html. [Kasutatud 16 04 2019].

[41] „INSERT,“ The PostgreSQL Global Development Group, 2019. [Võrgumaterjal]. Available: https://www.postgresql.org/docs/9.5/sql-insert.html. [Kasutatud 16 04 2019].

[42] „DELETE,“ The PostgreSQL Global Development Group, 2019. [Võrgumaterjal]. Available: https://www.postgresql.org/docs/current/sql-delete.html. [Kasutatud 19 04 2019].

[43] „SQL query to get status as of a given date,“ Stack Overflow, 2013. [Võrgumaterjal]. Available: https://stackoverflow.com/questions/13706354/sql-query-to-get-status-as-of-a-given-date. [Kasutatud 17 03 2019].

[44] „How much disk-space is needed to store a NULL value using postgresql DB?,“ Stack Overflow, 2011. [Võrgumaterjal]. Available: https://stackoverflow.com/questions/4229805/how-much-disk-space-is-needed-to-store-a-null-value-using-postgresql-db. [Kasutatud 2019 04 26].

[45] „PostgreSQL Tables Can Only Have 1600 Columns, Ever.,“ 03 01 2017. [Võrgumaterjal]. Available: https://nerderati.com/2017/01/03/postgresql-tables-can-have-at-most-1600-columns/. [Kasutatud 04 04 2019].

[46] „PostgreSQL: Create an index to quickly distinguish NULL from non-NULL values,“ Stack Overflow, 2016. [Võrgumaterjal]. Available: https://stackoverflow.com/questions/31966218/postgresql-create-an-index-to-quickly-distinguish-null-from-non-null-values. [Kasutatud 12 04 2019].

[47] „Conditional Expressions,“ The PostgreSQL Global Development Group, [Võrgumaterjal]. Available: https://www.postgresql.org/docs/11/functions-conditional.html#FUNCTIONS-GREATEST-LEAST. [Kasutatud 04 04 2019].

[48] „ALTER TABLE,“ The PostgreSQL Global Development Group, 2019. [Võrgumaterjal]. Available: https://www.postgresql.org/docs/9.1/sql-altertable.html. [Kasutatud 19 04 2019].

[49] A. B. P. M. L. Michael Kifer, „Database Systems: An Application-Oriented Approach,“ [Võrgumaterjal]. Available: http://testbank360.eu/sample/solution-manual-database-systems-2nd-edition-kifer.pdf. [Kasutatud 17 03 2019].

[50] P. Zaitsev, „Picking datatype for status fields,“ Percona, 2008. [Võrgumaterjal]. Available: https://www.percona.com/blog/2008/08/09/picking-datatype-for-status-feilds/. [Kasutatud 17 03 2019].

[51] „PostgreSQL Boolean Data Type with Practical Examples,“ PostgreSQL Tutorial Website, 2019. [Võrgumaterjal]. Available: http://www.postgresqltutorial.com/postgresql-boolean/. [Kasutatud 26 04 2019].

[52] „State pattern,“ Wikipedia, 17 03 2019. [Võrgumaterjal]. Available: https://en.wikipedia.org/wiki/State_pattern. [Kasutatud 18 03 2019].

[53] „Constraints,“ The PostgreSQL Global Development Group, 2019. [Võrgumaterjal]. Available: https://www.postgresql.org/docs/9.4/ddl-constraints.html. [Kasutatud 19 04 2019].

[54] „SQL Assertions / Declarative multi-row constraints,“ 18 05 2016. [Võrgumaterjal]. Available: https://community.oracle.com/ideas/13028. [Kasutatud 04 04 2019].

[55] „CREATE TABLE,“ The PostgreSQL Global Development Group, 2019. [Võrgumaterjal]. Available: https://www.postgresql.org/docs/9.1/sql-createtable.html. [Kasutatud 19 04 2019].

[56] „DROP TABLE,“ The PostgreSQL Global Development Group, 2019. [Võrgumaterjal]. Available: https://www.postgresql.org/docs/8.2/sql-droptable.html. [Kasutatud 19 04 2019].

[57] „Dynamically update status column after X amount of time,“ Stack Overflow, 2014. [Võrgumaterjal]. Available: https://stackoverflow.com/questions/25166271/dynamically-update-status-column-after-x-amount-of-time. [Kasutatud 18 03 2019].

[58] „SQL set column with derived value,“ Stack Overflow, 2019. [Võrgumaterjal]. Available: https://stackoverflow.com/questions/54013880/sql-set-column-with-derived-value. [Kasutatud 18 03 2019].

[59] „Specify Computed Columns in a Table,“ Microsoft, 2017. [Võrgumaterjal]. Available: https://docs.microsoft.com/en-us/sql/relational-databases/tables/specify-computed-columns-in-a-table?view=sql-server-2017. [Kasutatud 29 03 2019].

[60] D. Kašnikova, „Vaadete mõju päringute täitmisplaanide koostamisele kahe andmebaasisüsteemi näitel,“ Tallinna Tehnikaülikool, Tallinn, 2015.

[61] „How to Get Table, Database, Indexes, Tablespace, and Value Size in PostgreSQL,“ PostgreSQL Tutorial , 2019. [Võrgumaterjal]. Available: http://www.postgresqltutorial.com/postgresql-database-indexes-table-size/. [Kasutatud 25 03 2019].

[62] „Source lines of code,“ Wikipedia, 12 03 2019. [Võrgumaterjal]. Available: https://en.wikipedia.org/wiki/Source_lines_of_code. [Kasutatud 05 03 2019].

[63] „LocMetrics - C#, C++, Java, and SQL,“ 10 2007. [Võrgumaterjal]. Available: http://www.locmetrics.com/. [Kasutatud 25 03 2019].

[64] „EXPLAIN,“ The PostgreSQL Global Development Group, 2019. [Võrgumaterjal]. Available: https://www.postgresql.org/docs/9.1/sql-explain.html. [Kasutatud 16 04 2019].

[65] „Excel GEOMEAN Function,“ ExcelJet, 2019. [Võrgumaterjal]. Available: https://exceljet.net/excel-functions/excel-geomean-function. [Kasutatud 26 04 2019].

[66] C. Gallant, „Arithmetic Mean and Geometric Mean,“ 16 04 2019. [Võrgumaterjal]. Available: https://www.investopedia.com/ask/answers/06/geometricmean.asp. [Kasutatud 16 04 2019].

[67] E. Eessaar, „Mõned mõtted ja soovitused lõputööde teemavaliku ja teemakäsitluse kohta,“ 2019. [Võrgumaterjal]. Available: http://staff.ttu.ee/~eessaar/loputood_soovitused.html. [Kasutatud 26 04 2019].

[68] „Correlation Coefficient: Simple Definition, Formula, Easy Steps,“ Statistics How To, 2019. [Võrgumaterjal]. Available: https://www.statisticshowto.datasciencecentral.com/probability-and-statistics/correlation-coefficient-formula/#Pearson. [Kasutatud 16 04 2019].

[69] „Excel statistical functions: PEARSON,“ Microsoft, 2019. [Võrgumaterjal]. Available: https://support.microsoft.com/en-us/help/828129/excel-statistical-functions-pearson. [Kasutatud 01 05 2019].

[70] „seisundid_SQL,“ GitHub, 2019. [Võrgumaterjal]. Available: https://github.com/susannapeek1/seisundid_SQL. [Kasutatud 02 05 2019].

[71] „How to generate random number in a range,“ PostgreSQL Tutorial, 2019. [Võrgumaterjal]. Available: http://www.postgresqltutorial.com/postgresql-random-range/. [Kasutatud 17 04 2019].

[72] „Generate test data in PostgreSQL table,“ Stack Overflow, 2016. [Võrgumaterjal]. Available: https://stackoverflow.com/questions/36463134/generate-test-data-in-postgresql-table. [Kasutatud 17 04 2019].

[73] „LIMIT and OFFSET,“ The PostgreSQL Global Development Group, 2019. [Võrgumaterjal]. Available: https://www.postgresql.org/docs/8.1/queries-limit.html. [Kasutatud 26 04 2019].

[74] „VACUUM,“ The PostgreSQL Global Development Group, 2019. [Võrgumaterjal]. Available: https://www.postgresql.org/docs/9.5/sql-vacuum.html. [Kasutatud 26 04 2019].

[75] M. Männil, „Mõnede SQL-andmebaasisüsteemide võimekusest SQLi keelelise liiasuse silumisel,“ Tallinna Tehnikaülikool, Tallinn, 2014.

[76] „Don't use NOT IN,“ PostgreSQL Wiki, 2019. [Võrgumaterjal]. Available: https://wiki.postgresql.org/wiki/Don%27t_Do_This#Don.27t_use_NOT_IN. [Kasutatud 06 05 2019].

[77] H. Benita, „Be careful with CTE in PostgreSQL,“ 2019. [Võrgumaterjal]. Available: https://medium.com/@hakibenita/be-careful-with-cte-in-postgresql-fca5e24d2119. [Kasutatud 06 05 2019].

[78] „UPDATE,“ The PostgreSQL Global Development Group, 2019. [Võrgumaterjal]. Available: https://www.postgresql.org/docs/9.1/sql-update.html. [Kasutatud 04 05 2019].

[79] „How does database indexing work? [closed],“ Stack Overflow, 2009. [Võrgumaterjal]. Available: https://stackoverflow.com/questions/1108/how-does-database-indexing-work. [Kasutatud 26 04 2019].

[80] „Character Types,“ The PostgreSQL Global Development Group, 2019. [Võrgumaterjal]. Available: https://www.postgresql.org/docs/9.1/datatype-character.html. [Kasutatud 26 04 2019].

[81] „How to estimate the size of one column in a Postgres table?,“ Stack Overflow, 2015. [Võrgumaterjal]. Available:

https://stackoverflow.com/questions/18316893/how-to-estimate-the-size-of-one-column-in-a-postgres-table. [Kasutatud 2019 04 26].

[82] „Numeric Types,“ The PostgreSQL Global Development Group, 2019. [Võrgumaterjal]. Available: https://www.postgresql.org/docs/9.1/datatype-numeric.html. [Kasutatud 2019 04 26].

[83] „Date/Time types,“ The PostgreSQL Global Development Group, 2019. [Võrgumaterjal]. Available: https://www.postgresql.org/docs/9.1/datatype-datetime.html. [Kasutatud 26 04 2019].

[84] „Parallel Plans,“ The PostgreSQL Global Development Group, 2019. [Võrgumaterjal]. Available: https://www.postgresql.org/docs/current/parallel-plans.html. [Kasutatud 06 05 2019].

[85] K. Rootalu, „Korrelatsioonikordajad,“ 2014. [Võrgumaterjal]. Available: http://samm.ut.ee/korrelatsioonikordajad. [Kasutatud 28 04 2019].

[86] D. Mindrila ja P. Balentyne, „Scatterplots and Correlation,“ [Võrgumaterjal]. Available: https://www.westga.edu/academics/research/vrc/assets/docs/scatterplots_and_correlation_notes.pdf. [Kasutatud 28 04 2019].

Lisa 1 – allikate otsimiseks kasutatud otsingustringid

Nii varasemate uuringute peatükki allikate otsimiseks kui ka disainide kataloogi koostamiseks kasutasin järgnevaid fraase, mida omavahel ka kombineerisin. Kõigi esitatud fraaside jaoks viisin otsingu läbi mitu korda, asendades sõnad *status*, *state*, *condition* üksteisega.

1. *Hold object states*
2. *Store entity statuses*
3. *Statuses in SQL databases*
4. *SQL database tables design for object status*
5. *Hold metadata in SQL databases*
6. *UML modelling for entity attributes*
7. *Database models for object conditions*
8. *Main object life-cycle*
9. *Database for storing statuses*
10. *Design for holding object status values*
11. *Presenting the status for the main objects in information system*
12. *How to store statuses in databases*
13. *What is the easiest way to hold statuses for objects in database*
14. *Storing statuses for PostgreSQL tables*
15. *Database design mistakes for status fields*
16. *What column type should be used for status field in SQL*
17. *Best value for storing object statuses in database*
18. *How to hold classificator values in databases*
19. Olemi seisund andmebaasis
20. Klassifikaatori tüüpi väärtuste hoidmine andmebaasis
21. Andmebaasis seisundite salvestamine

Lisa 2 – SQL koodi formaatimise näide leidmaks füüsilist koodiridade arvu

```
SELECT a.seisund,
       a.tellimuste_arv
FROM   (SELECT Count(*) AS tellimuste_arv,
               'Ootel'  AS seisund
        FROM   ootel_tellimus
        UNION
        SELECT Count(*)        AS tellimuste_arv,
               'Töötlemisel' AS seisund
        FROM   tootlemisel_tellimus
        UNION
        SELECT Count(*)      AS tellimuste_arv,
               'Tühistatud' AS seisund
        FROM   tuhistatud_tellimus
        UNION
        SELECT Count(*)          AS tellimuste_arv,
               'Välja saadetud' AS seisund
        FROM   valja_saadetud_tellimus
        UNION
        SELECT Count(*)            AS tellimuste_arv,
               'Kohale toimetatud' AS seisund
        FROM   kohale_toimetatud_tellimus) a;
```