T A R T U Ü L I K O O L

MATEMAATIKA-INFORMAATIKATEADUSKOND

Arvutiteaduse instituut

Infotehnoloogia eriala

Taivo Käsper

Dünaamilist andmebaasiskeemi nõudvate rakenduste andmemudelite esitamise võimalused ja iseärasused

Bakalaureusetöö (6EAP)

Juhendajad: Anne Villems, MSc

Targo Tennisberg, AS Nortal

Autor: ……..... "….." mai 2013

Juhendaja: ……..... "….." mai 2013

Juhendaja: ……..... "….." mai 2013

Lubada kaitsmisele

Professor : ..... "….." mai 2013

TARTU

Sisukord

Sissejuhatus

Infotehnoloogia on avanud uued võimalused tööjõu efektiivsemaks kasutamiseks. Üritatakse automatiseerida võimalikult paljud rutiinsed tööd ning raskete ülesannete lahendamisel otsitakse abi arvutitelt. Sellepärast kuuleme üha enam tööjõupuudusest infotehnoloogia vallas. Suurenev sõltuvus tehnoloogiast on tõstnud ka kasutajate ootusi tarkvara suhtes. Järjest rohkem eeldatakse tarkvaralt nutikat käitumist, kiirust ja kvaliteeti, kuid unustatakse, et tarkvara on tegelikult täpselt nii tark kui programmeerija, kes selle kirjutas.

Kasutajate personaalsemaks kogemuseks tuleb infosüsteemidel salvestada andmeid ning juhtida enda käitumist vastavalt. Selleks peab tarkvaraprojekti algfaasis olema defineeritud andmemudel, mille põhjal tarkvara oskab leida õige teabe. Andmemudeli loomiseks peab aga väga täpselt teadma olemeid ja nende vahelisi seosed, sest hilisem mudeli muutmine võib andmebaasi poolt seatud piirangute tõttu olla aja ja ressursimahukas ülesanne. Kõikide rakenduste puhul ei ole võimalik andmeolemeid lõplikult kirjeldada, sest nad muutuvad rakendus elutsükli jooksul korduvalt.

Antud bakalaureusetöö raames uuritakse erinevaid andmete hoiustamise viise, kui andmeolemite tunnuste kohta on teada, et nad muutuvad tarkvara loomise ja edasise kasutamise käigus. Lisaks võrreldakse nende jõudlust erinevate operatsioonide korral ning üritatakse välja tuua millise kasutuse jaoks milline andmemudel kõige parem on.

Töö eesmärk on koostada ülevaade dünaamilistest andmemudelitest, anda teavet nende kasutamise kohta ning jagada infot nende sobilikkusest erinevate rakenduste korral. Lisaks tuli töö käigus luua liides, mis lihtsustaks seminormaliseeritud andmemudeli kasutamist. Mudelite võrdlemiseks tehti jõudlustestid, et teha otsuseid nende võimekusest erinevate andmeoperatsioonide ja kogude korral.

Peatükis 1 kirjeldatakse probleem ja tuuakse kõrvale täpsem seletus ühe hüpoteetilise näite toel. Peatükis 2 tutvustatakse erinevaid dünaamiliste andmemudelite esitamise viise ning peatükis 3 võrreldakse kahe valitud andmemudeli jõudlust erinevate andmeoperatsioonide korral ning tuuakse välja andmete sisestamise ja pärimise jaoks parimad neist vaadeldud mudelitest.

Töö lisades on toodud:

- Terminoloogia
- Adam Milazzo kiri NoSQL andmebaaside puudustest
- Testarvuti riistvara kirjeldus
- *Dynamic-Datastructures* liidese käivitamise käsud

1. Probleem

Tarkvara arendajad puutuvad tihti kokku probleemiga, et on vaja tavalisse relatsioonilisse andmebaasi paigutada infot, mille kuju kohta ei teata piisavalt informatsiooni, et kirjeldada andmemudelit: nende struktuur kas muutub tihti või pole analüütikute poolt kindlaks tehtav.

Selleks, et infosüsteemi arendus oleks odavam peab klient väga täpselt teadma, mida ta soovib, et tarkvara teha oskab. Pärast kliendi probleemi püsitust analüüsitakse väga täpselt ära iga kasutaja tegevus. Selle käigus valminud dokumentides on kirjas iga kasutaja tegevus võimalikult täpse kasutusloona, mille põhjal disainitakse rakenduse arhitektuur ja hakatakse realiseerima äriloogikat. Kuna tegu on mingite kindlate tegevuste jaoks spetsialiseeritud tarkvaraga, siis tuleb selle tellijal iga uuenduse korral pöörduda uuesti arendusmeeskonna poole, kes viib läbi vajalikud muudatused ning esitab nende tegemise eest arve. Vahepeal on vajalik arendusetapi juurutamiseks võtta infosüsteem mingiks ajaks kasutusest maha, mille tõttu võib kliendile lisanduda veel kulutusi saamata jäänud tulu näol vms. Olenevalt infosüsteemist ei pruugi rakenduse maha võtmine olla võimalik, nt. meditsiini infosüsteem peab olema alati kättesaadav.

Vaatleme probleemi täpsemaks selgitamiseks lähemalt meditsiini infosüsteemi, kus hoitakse patsientide analüüside tulemusi. Haigete ja arstide jaoks ei ole lubatav, et süsteem aeg-ajalt tunni jooksul, mil uuendusi paigaldatakse, ei tööta. Samuti ei saa neid tegevus viia läbi öisetel kellaaegadel, sest nt. erakorralise meditsiini, intensiivravi ja kirurgia osakonnad töötavad ka öösel. Süsteemi mitte töökorras olemise tõttu võib arstidel jääda saamata vajalik informatsioon patsiendi kohta, samuti seiskub töö, sest kogu meditsiiniline tegevus peab olema dokumenteeritud, mis tänapäeval toimub vaid elektrooniliselt. See tähendab, et infosüsteem peab olema arendatud nii, et muudatuste läbi viimise käigus oleks rakendus endiselt kasutatav.

Näiteks rakenduse kliiniliste leidude osa, kus on haiguslugu, füüsiline läbivaatus, laboratoorsed uuringud, diagnoosid, peaks olema realiseeritud nii, et aja jooksul, mil nende hulk võib muutuda märkimisväärselt, saaks uuendused viia läbi tarkvara kasutaja ise. Samuti ei ole mõistlik hoida kõigi patsientide kohta ühesugust infot - saaksime suure hulga andmeid, millest enamik väljendaksid sellist informatsiooni, mis konkreetse patsiendi kohta ei ole oluline. Näiteks ei ole mõistlik hoida kõikide patsiendide kohta informatsiooni kas ta on rase või ei, sest meeste puhul oleks selleks alati ei. Tegelikkuses ei oleks kunagi sellist nimekirja kõikidest leidudest kirja panna, sest nende hulk on liiga suur ja ajas suhteliselt kiiresti muutuv.

Nagu näha, sisaldab rakendus andmeid, mille kohta ei saa anda informatsiooni tarkvara arenduse käigus, vaid see kujuneb iseseisvalt selle kasutuse käigus. Siit tekib dünaamiline komponent, mida võib hetkel vaadelda ka kui dünaamilist andmestruktuuri, sest andmete kohta käiv tunnuste hulk on pidevas muutumises.

Dünaamiline andmestruktuur tähendab, et on mingid objektid, millel on erinevate tunnuste suur osakaal. Selliselt erinevate olemite tõttu tekib andmebaasi tabelisse, millesse need objektid salvestatakse, väga suur hulk veerge, millest enamus hoiavad null väärtusi.

1.1. Näidisrakendus probleemi selgituseks

Eeldame, et meil on arendusjärgus meditsiini infosüsteem ning järgmises tsüklis tuleb valmistada tarkvara, mille kaudu saab laborant sisestada laboratoorse analüüsi tulemused infosüsteemi, kust need on arstidele vaadeldavad ja diagnoosimiseks kättesaadavad.

Järgnevalt kirjeldame komponenti, mille kaudu laborant saab lisada analüüsi tulemused. Muidugi tuleb arvestada, et selle bakalaureusetöö kirjutamise käigus ei ole tutvutud laborandi reaalse keskkonnaga, seega vaadeldakse hüpoteetilist, mis oleks võimalikult ligilähedane reaalsusele ja näitlikustaks piisavalt hästi probleemi.

1.2.1. Näidisrakenduse funktsionaalsed nõuded

Laborandi tulemuste lisamise töökeskkond koosneb kahest sammust.

1. Patsiendi otsing
    a. saab otsida patsienti isikukoodi järgi
    b. saab märkida patsiendi valituks, et ühendada temaga analüüsi andmeid.
2. Tulemuste lisamise vorm
    a. saab nimekirjast valida tehtud analüüsi nimetuse ja sisestada saadud tulemused.

1.2.2. Näidisrakenduse mittefunktsionaalsed nõuded

Laborandi tehtud analüüside tulemuste lisamise töökeskkond peab vastama järgnevatele mittefunktsionaalsetele nõuetele.

- Rakendus peab päringule vastama vähemalt 10 sekundi jooksul.
- Kaua aega võtvad tööd käivitatakse võimaluse korral öösel (öised tööd) ja peavad lõppema vähemalt 1 h jooksul.
- Kõik öised tööd tohivad töötada kellaaegadel 1:00 - 5:00.

Sellised nõuded on olulised süsteemi kvaliteedi hindamiseks ja hoidmiseks.

2. Dünaamilised andmemudelid

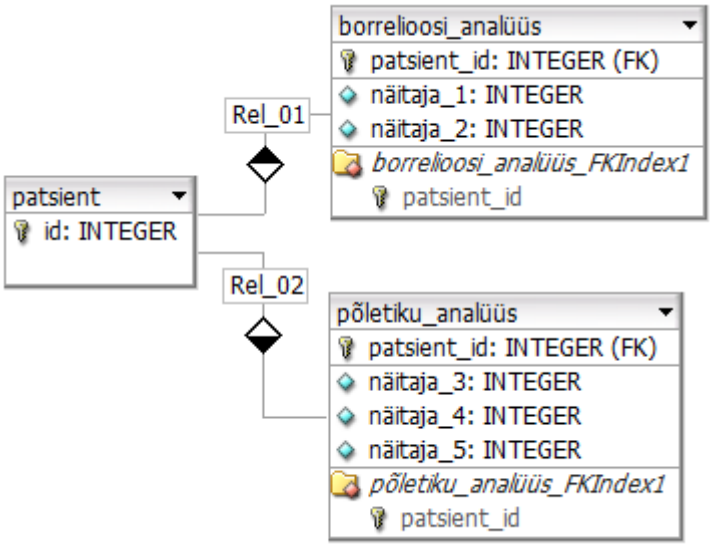
Rakenduse analüüsi tulemuste salvestamise alamsüsteemi realiseerimiseks on võimalik kasutada mitmeid andmemudeleid, kuid selles bakalaureusetöös võrreldakse kahte. Üks neist on olem-tunnus-väärtus andmudel [1], mis on sarnaste probleemide lahendamiseks laialdaselt kasutusel ning teine on seminormaliseeritud andmemudel, mis on laiemat populaarsust kogunud Microsoft SharePointi vahendusel. Olem-tunnus-väärtus ja seminormaliseeritud andmemudel on võrdlusse valitud sellepärast, et olem-tunnus-väärtus on üks populaarsemaid dünaamilisi mudeleid ja seminormaliseeritud andmemudel kaasjuhendaja Targo Tennisbergi soovituse tõttu.

Traditsiooniline andmemudeli arenduskäik hõlmab endas järgnevaid tegevusi:

- Luua rakendusele andmemudel ja selle põhjal andmebaasi skeem.
- Andmemudel on tavaliselt kolmandal normaalkujul ning indekse lisatakse hiljem vastavalt vajadusele.
- Andmebaasi kasutuse käigus ilmsiks tulnud probleemide korral lisatakse veel indekse ja tehakse erinevaid päringute optimeeringuid.
- Kui andmebaasi tabelisse on vaja lisada veerg, siis tuleb selleks iga kord pöörduda arendajate poole ja aeg-ajalt süsteem maha võtta.

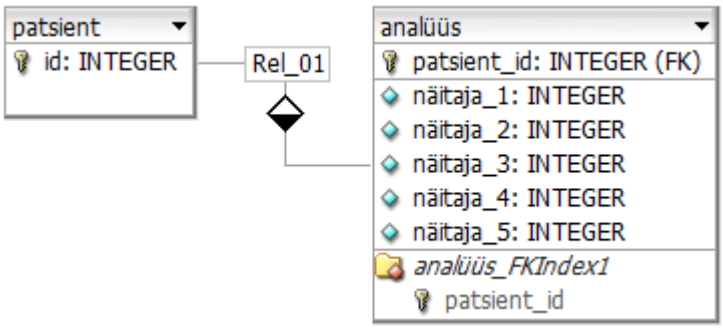
Juhul kui me analüüsi tehes ei saa paika panna andmebaasi mudelit, sest see muutub rakenduse loomise ja edasise kasutamise käigus, tuleb meil kasutada mõnda dünaamilist andmestruktuuri võimaldavat lahendust.

Vaatleme olukorda, kus perearst saadab laborisse analüüsimiseks kaks vereproovi, mis mõlemad on võetud erinevatelt inimestelt. Koos vereproovidega lisab ta saatelehe, kus märgistab teda huvitavad analüüsid - ühel patsiendil on vaja mõõta põletikunäitajaid ja teisel testida puukborrelioosi näitajaid. Iga näitaja koosneb nimest ja vastavast arvust. Selleks, et analüüside tulemused salvestada andmebaasi on vajalik joonisel 1 kujutatud andmemudel, kus iga analüüsi jaoks tuleb teha eraldi tabel.



Joonis 1 - erinevate analüüside tulemused andmebaasi eraldi tabelites

Kuid analüüside arvu kasvades näeme üsna varsti, et erinevaid analüüside tabeleid on liiga palju ja nad kõik tuleks abstraheerida ühte. Selle tulemusel saame joonisel 2 kujutatud andmemudeli, kus laborant täidab ära vaid need näitajate veerud, mida ta analüüsi käigus määras.



Joonis 2 - erinevad analüüsid ühes tabelis

Näidatud andmemudeliga (vt. joonis 2) avastatakse üsna varsti, et näitajate arvu kasvades on suurem osa tabelist tühi ehk sisaldab null väärtusi, sest iga rida kirjeldab ühte analüüsi, kuid tabelis on iga rea jaoks ära kirjeldatud kõikvõimalikud analüüsi näitajad. Samuti tuleb iga kord, kui võetakse uuringutes kasutusele uus analüüs (ühe kuu jooksul mitmeid [1]), pöörduda tarkvaraarendajate poole ja lasta neil lisada uus veerg mingi uue näitaja jaoks. See teeb aga infosüsteemi ülalpidamise ja tegelikkusele vastavana hoidmise väga kalliks. Lisaks tekib ka palju tehnilisi probleeme, millest suurim on andmemudeli muutmine töötavas süsteemis, mille tulemusena lukustatakse muudetav tabel pikaks ajaks (täpne aeg oleneb andmete hulgast). Sellepärast ei ole võimalik töötaval süsteemil teha muudatusi andmetestruktuurile ja lisada samal ajal andmeid muudetavasse tabelisse.

Andmemudeli muutmiseks peab andmebaasimootor kopeerima vana tabeli struktuuri, tegema sellele vajalikud muudatused ning seejärel tõstma kõik andmed vanast tabelist uude. Kui andmed on kopeeritud, saab vana tabeli kustutada ja võtta kasutusele uue. Kogu kopeerimise aja ei tohi vanas tabelis olevatele andmetele saada teha muudatusi ja lisamisi, sest muidu ei jõuaks need uude tabelisse - kannataks andmete terviklikkus. Sellepärast lubab andmebaasi mootor ainult andmete lugemist kopeerimise ajal. Töös olevates andmebaasides on tihti väga suur kogus andmeid, mille kopeerimine võib võtta tunde või isegi päevi, kuid tabeli lukustamine ei ole lubatav, sest inimesed peavad saama samal ajal teha rakendusega tööd.

Bakalaureuse astme õpingute käigus on kirjutaja puutunud kokku sellise probleemiga, kui ettevõttele ZeroTurnaround arendati statistika rakendust, mis oli staatilise andmemudeliga. Arenduse käigus oli vaja umbes 54,5 miljoni reaga tabelisse lisada juurde kaks numbri välja. Antud tabelit kasutati peamiselt andmete sisestamiseks, mis pärinesid JRebeli kasutusest tulenevast anonüümsest kasutusstatistikast, mille alusel Zeroturnaround sai suunata arendusjõud õigetesse kohtadesse ja näha müügiosakonna edukust erinevates piirkondades. Sellest tulenevalt oli õige ja ajakohane info nende jaoks äärmiselt tähtis, kuid väikesed andmekaod aktsepteeritavad.

Sellisesse tabelisse kahe veeru lisamine osutus keerulisemaks kui oleks osanud arvata, sest nende lisamine võttis aega umbes 2 tundi, mille jooksul oli tabel lukustatud sisestuste ja muudatuste jaoks. Enne päris keskkonna muutmist tuli andmete põhjal ennustada, mis kell on antud tabelil kõige väiksem koormus ning kui palju andmeid peaks lisanduma lukustuse ajal. Seejärel tuli saadud info kooskõlastada spetsialistidega ning alles siis võis tööle asuda.

Õnneks olid ZeroTurnaroundi arendajad teinud nii head tööd JRebeli arendamisel, et kui andmete edastamisel tekkis probleem, hoiti neid kasutajate arvutites lokaalselt ning prooviti natukese aja pärast uuesti.

Seni ajani on probleemi lahenduseks pakkutud välja kolmandal normaalkujul lahendusi, kuid kõigi nendega esinevad dünaamiliste tunnuste esitamisel probleemid (joonis 1, joonis 2), mille tõttu neid ei saa paljude juhtudel kasutada. Edaspidi võrreldakse kahte täiesti erinevat andmemudelit dünaamiliste andmestruktuuride esitamiseks ning sobitatakse need meditsiiniinfosüsteemi näitega.

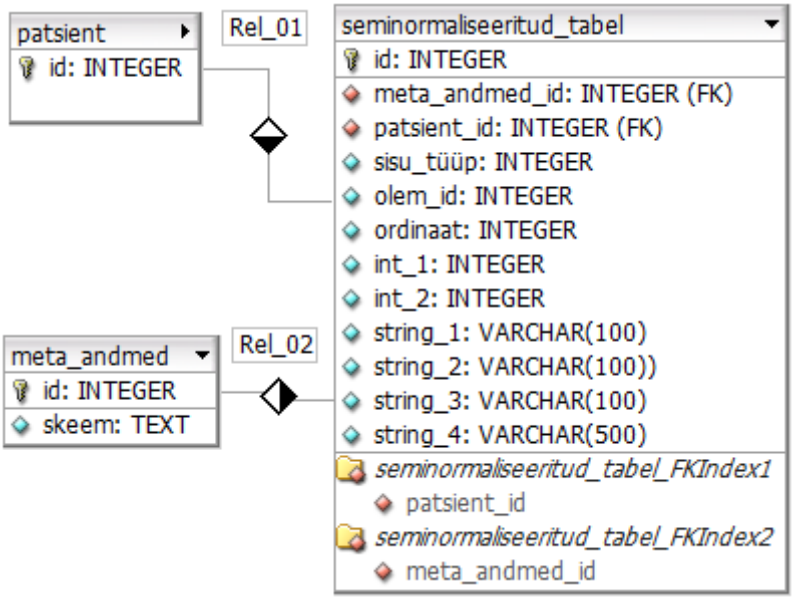
2.1 Seminormaliseeritud andmemudel

Seminormaliseeritud andmemudeli põhieesmärk on võimaldada programmeerijal salvestada olemasolevatesse relatsioonilise andmebaasi tabeli veergudesse ükskõik kui palju tunnuse väärtusi - iga rida võib teoreetiliselt olla erinev, kuid praktikas on andmed jaotatud sisutüüpideks (*content type*). Mõeldes relatsioonilise mudeli kolmanda normaalkuju peale võib kujutada ette, et sisu tüüp on sama mis tabeli nimi - saame ühte seminormaliseeritud kujul andmestruktuuri paigutada mitu tabelit ehk erineva struktuuriga andmeid. Sellises andmemudelis luuakse tarkvaraarenduse algfaasis iga andmetüübi kohta tabelisse mingi arv veerge ning edaspidi kasutatakse olemasolevaid võimalusi. Juhul kui eeldefineeritud tabeli veergude hulgas ei ole enam vabasid, õige tüübiga veerge, paigutatakse andmed uuele reale. Sellest tulenevalt esitatakse andmebaasi tabelis ühe olemieksemplari (item_id) andmeid mitme reaga. Andmete asukoha määramiseks kasutatakse lisaks dünaamilistele veergudele ka süsteemseid nagu rea identifikaator (id), sisu tüüp (content_type), olemieksemplari identifikaator (item_id) ja ordinaat (ordidant_id).

Rakendus, mis kasutab andmebaasi andmeid ei tea kunagi nende asukohta seminormaliseeritud tabelis. Kui on vaja pärida andmebaasilt mingeid andmeid, siis kõigepealt pöördub ta selle poole küsimusega, mis koordinaatidel hoitakse mingit tüüpi andmete mingit tunnust. Seejärel vaatab andmebaas metaandmeid ja vastab tunnuse koordinaatidega - võime kujutada ette, et tegu on x ja y koordinaatidega, kus x tähistab veergu ja y rea numbrit konkreetse olemi piires. Seejärel saab rakendus pöörduda otse teda huvitava tunnuse väärtust otsima.

2.1.1. Seminormaliseeritud andmemudeli modelleerimine

Andmemudeli võib modelleerida vastavalt rakenduse spetsiifikale, kuid meditsiiniinfosüsteemi näitel näeb see välja nii nagu on kujutatud joonisel 3. Süsteemseteks veergudeks on tabelis seminormaliseeritud_tabel id, patsient_id, sisu_tüüp, olem_id, ordinaat ning ülejäänud veergude arv ja nende tüübid on tuvastatud analüüsi käigus.



Joonis 3 - seminormaliseeritud andmemudel

Mittesüsteemsete väljade arv (joonisel 3 on nendeks kõik alates veerust int_1) on oluline, sest kui neid on liiga vähe, siis tekib andmebaasi objektide salvestamisel liiga palju ridu, mis teeb andmetagastus päringu ja salvestamise aeglaseks, sest tegelikult tuleb töötada mitme reaga. Kui neid on liiga palju siis võib andmebaasi tabelisse tekkida palju null väärtusi, mis võtavad kõvakettal sama palju ruumi kui maksimaalselt täidetud veerud ning teevad andmebaasi suureks ja aeglaseks. On äärmiselt oluline, et andmebaasi disainimise käigus luuakse optimaalne arv erinevat tüüpi mittesüsteemseid veerge. Loodavate tunnuste tüüp sõltub rakenduse valdkonnast. Kui andmebaasis on vaja hoida palju dünaamilisi teksti tüüpi tunnuste väärtusi, peaks seminormaliseeritud mudeliga tabel sisaldama rohkem tesktilisi veerge; kui andmebaasis on vaja hoida rohkem dünaamilisi numbrilisi tunnuseid, siis numbrilisi veerge.

2.2 Olem-tunnus-väärtus andmemudel

Olem-tunnus-väärtus andmemudelis on iga olemi tunnusest ja selle väärtusest tehtud andmebaasi tabelisse uus kirje. Ühes andmebaasi tabelis on kolm veergu: esimene olemi identifikaatori, teine tunnuse nimetuse ja kolmas tunnuse väärtuse hoidmiseks (alati tekstilise tunnusena). Andmebaasi read, mis on ühesuguse olemi identifikaatoriga, moodustavad ühe olemi, kus iga atribuut on selle tunnuseks. Andmete näitlik paigutus sellises andmebaasi tabelis on kujutatud tabelis 1.

Olemi id	**Atribuut**	**Väärtus**
1	Pikkus	191
1	Kaal	75
2	Pikkus	190
2	Kaal	85
2	Netopalk	700

Tabel 1 - näidisandmete paigutus olem-tunnus-väärtus mudeliga tabelis

Nagu näha, on võimalik erinevatele olemitele lisada suvalisi atribuute ja väärtusi (vt. tabel 1). Selleks, et hiljem oleks teada, mis olemiga on tegu, on vaja teha eraldi tabel (vt. tabel 2), kus hakatakse hoidma erinevate olemite kohta informatsiooni.

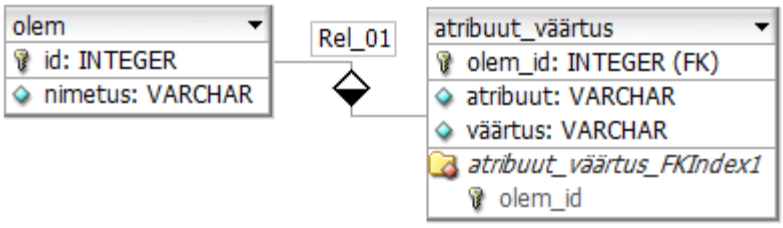
Id	**olem**
1	Inimene 1
2	Inimene 2

Tabel 2 - näidisandmete paigutus olem-tunnus-väärtus mudeliga olemit kirjeldavas tabelis

Sellises andmemudelis võib hoida ükskõik milliseid olemeid. Lisaks saab vältida null väärtusi, sest juhul kui olemil on mingi atribuut väärtuseta, siis võime andmebaasi selle salvestamata jätta. See hoiab andmete mahtu kokku, kuid samas peame igale reale salvestama alati tunnuse nimetuse või tegema atribuudi välisvõtmeks mõnele teisele tabelile, kus kirjeldame ära nende nimetused.

2.2.1 Olem-tunnus-väärtus andmemudeli modelleerimine

Olem-atribuut-väärtus andmemudeli saab modelleerida joonisel 4 kujutatud viisil.

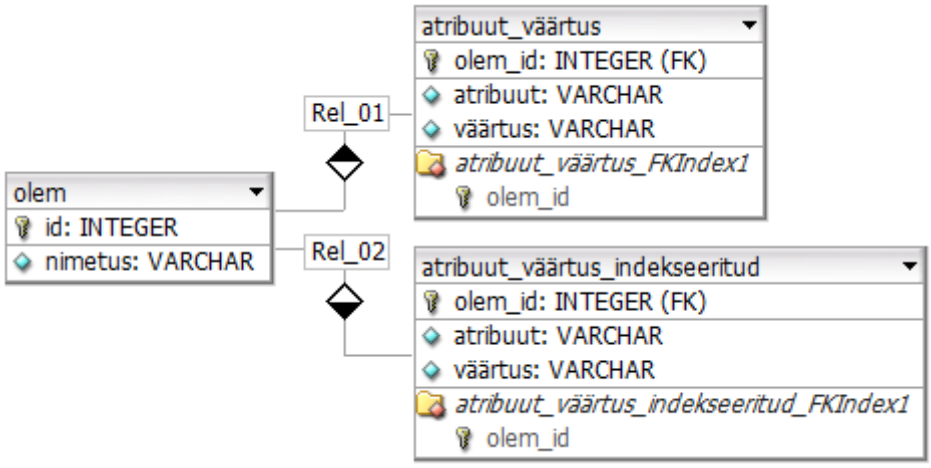


Joonis 4 - Olem-tunnus-väärtus andmemudel

Andmete terviklikkuse säilitamiseks on vajalik lisada atribuut_väärtus tabelile unikaalsuse piirang - atribuut_väärtus tabelis peavad olem_id ja atribuut olema unikaalsed, sest ühelgi olemil ei tohi olla mitu sama nimega atribuuti.

Näeme, et kui atribuudi või selle väärtuse järgi on kunagi vaja otsingut sooritada, siis peame indekseerima nii atribuut kui ka väärtus veerud, sest selliselt disainitud tabelisse tekib kordades rohkem ridu kui relatsioonilisse kolmandas normaalkujus olevasse tabelisse. Lisaks saab tabel hoida erineva struktuuriga andmeid, mis relatsioonilises kolmandas normaalkujus oleksid paigutatud eraldi tabelitesse. Sellepärast on sellises olem-tunnus-väärtus mudeliga tabelis suuremahulised indeksid.

Indeksite suuruse probleemi ära hoidmiseks saab teha sellise andmemudeli, kus paigutame kõik indeksit vajavad väärtused eraldi tabelisse. Iga kord kui tahame mingile päringule lisada tingimuslause teame, et tegelikult tunnus nimega pikkus peaks olema teistest tunnustest eraldi tabelisse paigutatud. Sellise muudetud andmemudeli probleemiks on olemi andmete mitmest tabelist kokku kogumine. Indeksi mahu optimeeringuga olem-tunnus-väärtus andmemudeli saab visuaalselt väljendada joonisel 5 näidatud mudeliga, kus tabelite atribuut_väärtus ja atribuut_väärtus_indekseeritud ainuke erinevus on selles, et viimasel on atribuut ja väärtus veergudel indeks.



Joonis 5 - Olem-tunnus-väärtus andmemudel indeksi mahu optimeeringuga

2.3 NoSQL

Üheks võimaluseks dünaamiliste andmemudelite realiseerimisel on näiteks NoSQL andmebaasisüsteemid [2]. Käesolevas töös käsitletakse neid ainult põgusalt, sest töö eesmärk on uurida dünaamiliste andmemudelite esitamise viise mõnes traditsioonilises relatsioonilises andmebaasis.

Üha kasvavate andmemahtudega tekkis vajadus andmebaasi järele, mis ei vajaks ühte keskset serverit. Keskne server piirab andmebaasi jõudluse ja usaldusväärsuse enda võimekuse ja usaldusväärsusega. Lisaks on serverarvutid märkimisväärselt kallimad kui tavalised arvutid. Keskse serveri asendamine mitme tavalise tarbearvutiga on NoSQL andmebaaside peamine eelis, sest ettevõtted saavad hoida raha kokku kallilt riistvaralt ning endiselt saavutada suurema jõudluse, usaldusväärsuse ja skaleeruvuse.

NoSQL andmebaasides saab hoida andmeid, mis ei pea olema kindlalt struktureeritud. Seega ei pea muretsema suurte andmekogude struktuuri muutmisele kuluva aja pärast, sest NoSQL andmebaasides ei võta tabelite struktuuri muutmine aega - tegelikult puudub seal üldse selline tegevus nagu struktuuri muutmine, sest andmed ei ole struktureeritud.

Järgnevad väited pärinevad Adam Milazzo kirjast, kus ta arutleb, miks nad võtsid FlairPoint dokumendihaldussüsteemis kasutusele seminormaliseeritud andmemudeli, mitte NoSQL andmebaasi. A. Milazzo on loonud FlairPoint'i dokumendihalduse süsteemi arhitektuuri ning töötab hetkel ettevõttes Nortal süsteemide arhitektina. Autor saatis A.

Milazzole kirja, et miks ei võetud kasutusele mõnda NoSQL andmebaasi. Originaalset kirja saab täispikkuses lugeda lisast 2.

NoSQL andmebaasi peamisteks miinusteks on andmete struktuuri puudumine. Relatsioonilistes andmebaasides töödeldakse andmed vastavalt andmemudeli muutusele, kuid NoSQL andmebaasides tekib andmemudeli muutuse tõttu andmetest mitu versiooni – sellega saavutatakse andmemudeli muutmise kiirus. See tähendab, et rakendus peab saama hakkama kõikide erinevate andmeversioonidega ning uus töötaja peab lisaks hetkel olemasolevale andmemudelile õppima ära ka kõik vanad andmemudelid. NoSQL andmebaasid andmete struktuuri ei hoia, kuid selleks, et rakendus saaks andmebaasiga suhelda, peab selle kusagil ikkagi defineerima.

NoSQL andmebaaside suureks miinuseks relatsiooniliste ees on nende uudsus. Selle tõttu puudub neil palju tähtsaid lisasid nagu näiteks vaated, protseduurid, funktsioonid jne. Lisaks on palju raskem teha keerulisi päringuid, sest võib eksisteerida mitu versiooni erineva struktuuriga andmeid.

3. Andmemudelite jõudlus erinevate operatsioonide korral

Andmemudelite jõudluse võrdlemiseks on selle bakalaureusetöö raames tehtud Java programmeerimise liides (*Application Programming Interface*) nimega *Dynamic-Datastructures*, mis lihtsustab oluliselt arendust, sest peidab programmeerija eest suurel hulgal seminormaliseeritud andmemudeli keerukusest. Liides on kõigile vabalt kättesaadav repositooriumist nimega *Bitbucket* [3].

Lisaks saab liidesega testida nii seminormaliseeritud andmemudeli kui ka olem-tunnus-väärtus andmemudeli jõudlust erinevate operatsioonide ja koormatuse korral. Andmebaasi koormuse tõstmiseks töötab liides mitme paralleelse lõimega, millega sooritatakse igaühega mingi teatud hulk operatsioone. Lõimede ja operatsioonide arvu saab määrata liidese konfiguratsiooni failist nimega „conf.properties“, kuid protsessori poolt toetatud paralleelsete lõimede arvu ei ole mõttekas ületada, sest neid ei täidetaks korraga. Liidese kasutamiseks vajalikud käsud asuvad lisas 3.

Selles bakalaureusetöös toodud testide tulemused on sooritatud võimalikult sarnastel tingimustel. Kasutati sülearvutit Acer, mille riistvara tehnilised andmed asuvad lisas 4,

operatsioonisüsteemi Windows 7 64 bit, andmebaasi serverit MySQL versioon 5.6.10 64 bit. Testarvuti poolt oli toetatud maksimaalselt 4 paralleelset lõime.

Kõigepealt tuleb luua konkreetne olem (*entity*) ning seejärel siduda (kaardistada) selle tunnused andmebaasi tabeli konkreetsetele väljadele. Näiteks kui olemiks on isik, kellel on tunnus nimi, siis tuleb kaardistada nimi andmebaasi mõnele tekstilist tüüpi veerule. Kui rakenduses soovitakse teostada nime järgi otsinguid, siis tuleks see atribuut kaardistada mõnele indekseeritud andmebaasi tekstilisele väljale.

Pärast sidumise tegemist saab programmeerija teha andmebaasi päringuid teadmata, et tegelikult kasutatakse seminormaliseeritud andmestruktuuri. Näiteks saab programmeerija teha päringu isik olemite kohta ja lisada tingimuslause „WHERE nimi = 'Urmas'“, mille tulemusena vaatab liides kõigepealt, kuhu tunnus „nimi“ seotud on ning seejärel kasutab nime asemel õiget tabeli veergu.

Kasutades *Dynamic-Datastructures* liidest, saab programmeerija keskenduda äriloogika implementeerimisele ja ei pea segama ennast andmemudeli keerukusega.

3.1 Seminormaliseeritud andmemudeli jõudlus

Dynamic-Datastructures liidesega testiti mitmete lõimedega paralleelset andmete sisestamist ja pärimist, et võrrelda seminormaliseeritud andmemudelit olem-tunnus-väärtus andmemudeliga. Sisestamisel genereeriti suvalised andmed. Tabel, milles andmeid hoiti, sisaldas ainult primaarvõtme indeksit, seega ei olnud andmete sisestamisel vajalik uuendada suurel hulgal indekseid, vaid ainult ühte. Andmete pärimise puhul lisati tabelile juurde vajalik indeks.

3.1.1 Seminormaliseeritud andmemudeli andmesisestuse jõudlus

Jõudluse testimiseks kasutati tabelil 3 näidatud andmeolemit, millel on 13 tunnust, mis jaotusid joonisel 3 näidatud tabelis seminormaliseeritud_tabel kolme rea vahel:

Klient
Id Eesnimi Perekonnanimi Kirjeldus Isikukood Linn Riik Maakond Telefoninumber Email Sünnikuupäev Pikkus Jalanumbri_suurus

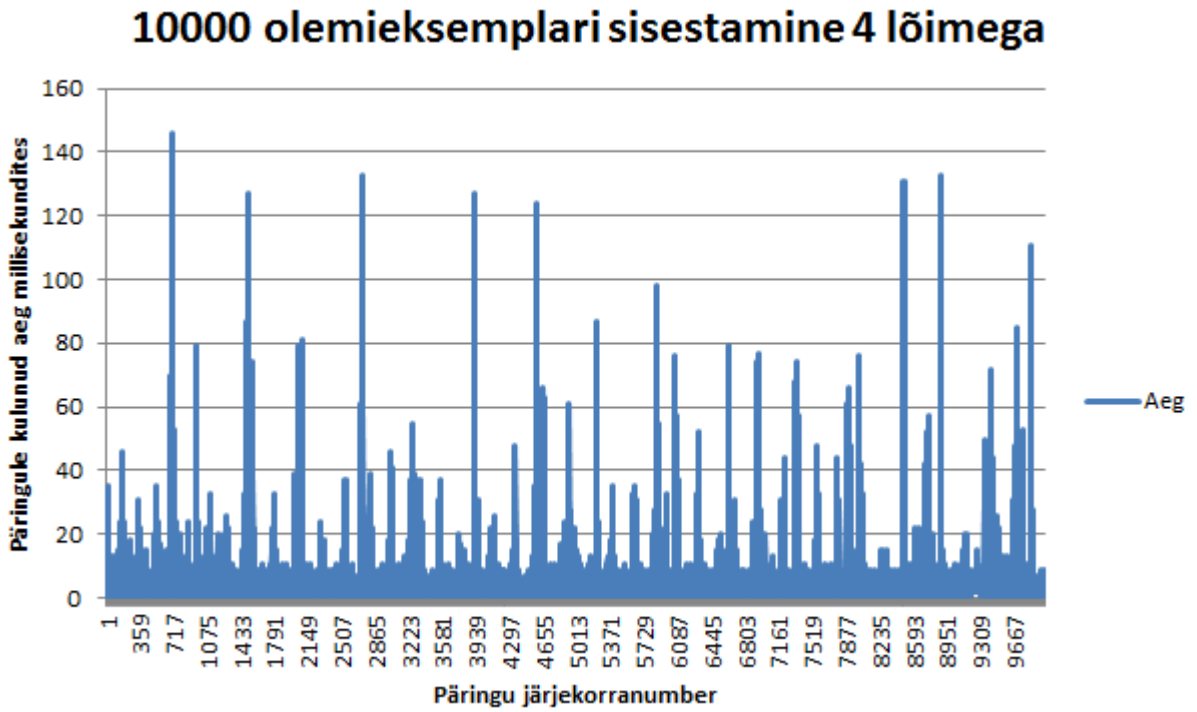
Tabel 3 - klient testolemi tunnuste loetelu

Tehtud testide tulemused on tabelis 4, kus mõõdetud aeg on terve ühe olemieksemplari lisamisele kulunud aeg.

Lõimede arv		**1000 andmeobjekti (3000 rida)**	**10000 andmeobjekti (30000 rida)**	**100000 andmeobjekti (300000 rida)**
1	Minimaalne aeg	4 ms	3 ms	1 ms
	Keskmine aeg	5 ms	5 ms	5 ms
	Maksimaalne aeg	333 ms	500 ms	616 ms
4	Minimaalne aeg	4 ms	3 ms	3 ms
	Keskmine aeg	5 ms	5 ms	5 ms
	Maksimaalne aeg	527 ms	1129 ms	1254 ms

Tabel 4 - ühe olemieksemplari salvestamisele kulunud aeg

Tabelist 4 on näha, et keskmine on väga lähedal miinimumile, kuid kaugel maksimumist, mis tähendab, et kaua aega võtnud päringuid on vähe. On näha, et andmete salvestamisele kulunud aeg ei sõltu andmete hulgast ega lõimede arvust. Protsessori hõivatuse näitaja ei ületanud andmete salvestamisel kordagi 30% piiri. Vaatleme täpsemalt näiteks nelja lõimega 10000 andmeobjekti sisestamiseks kulunud aegu.



Joonis 6 - 10000 olemieksemplari sisestamise ajad 4 lõimega

Jooniselt 6 on näha, et tulemused on normaaljaotusega, ehk koondunud keskväärtuse 5 ms lähedale valimi standardhälbega 9.55. Seega on võimalik 99% tõenäosusega öelda, et suvalise andmesisestuse päringu kiirus koormuse all jääb vahemikku 4.76 kuni 5.25 millisekundit.

3.1.2 Seminormaliseeritud andmemudeli andmetagastuse jõudlus

Andmeid tagastava päringu kiiruse testimiseks lisati veerule, mis sisaldab inimese pikkust indeks. Salvestati suvaliselt genereeritud andmetega olemieksemplare ning mõõdeti kui kiiresti saadakse andmebaasist kõik ühe andmeobjekti väärtused, kus filtreeritava pikkus tunnuse väärtuseks on mingi vabalt valitud arv nii, et iga päring tagastaks ühe andmeobjekti. Mitme lõime korral valib iga lõim suvalise olemieksemplari ja leiab sellele kõik tunnuste väärtused. Tulemused on tabelis 5.

Lõimede arv andmete selekteerimisel	**1000 andmeobjekti (3000 rida)**	**10000 andmeobjekti (30000 rida)**	**100000 andmeobjekti (300000 rida)**
1	4 ms	3 ms	4 ms
4	keskmine 12 ms maksimaalne 15 ms minimaalne 4 ms	keskmine 13 ms maksimaalne 16 ms minimaalne 7 ms	keskmine 11 ms maksimaalne 15 ms minimaalne 8 ms

Tabel 5 - andmetagastus päringu kiirused

Tabelis 5 olevatest andmetest on näha, et andmete pärimise kiirus ei olene andmete hulgast ja oleneb vähesel määral koormusest, mida andmebaasile mitme paralleelse lõimega proovitakse tekitada. Eelnev väide kehtib ainult siis, kui tabeli veerg, milles filtreeritav tunnus asub, on indekseeritud ja tingimusel, et iga päring tagastab samasuguse arvu tulemeid. Juhul kui koos andmeobjektide arvu kasvamisega kasvaks ka tagastatavate ridade hulk, muutuksid kiirused vastavalt, sest andmebaasi tabelist tuleks leida suurem hulk kirjeid, millest saab kokku andmeobjekti.

3.2 Olem-tunnus-väärtus andmemudeli jõudlus

Suvaliselt genereeritud tunnuse väärtustega klient jaotatakse igaüks eraldi tabeli reale ning selle salvestamisel tekivad andmebaasi tabelisse nimega atribuut_väärtus tabelis 6 näidatud viisil info.

Olem_id	Atribuut	Väärtus
1	Eesnimi	Mari
1	Perekonnanimi	Saar
1	Tutvustus	Olen tore neiu
1	Isikukood	49301261234
1	Linn	Tartu
1	Riik	Eesti
1	Maakond	Tartumaa
1	Telefoninumber	5555555
1	Email	email@email.ee
1	Sünnikuupäev	26.01.1993
1	Pikkus	185
1	Jalanumbri_suurus	36

Tabel 6 - näidis klient olemi andmed olem-tunnus-väärtus andmemudeliga tabelis

Sarnaselt sellele olemieksemplarile genereeriti suvaliste andmetega suurem kogus olemieksemplare ja salvestati andmebaasi, igaühe salvestamisel mõõdeti kogu ühe eksemplari salvestusele kulunud aeg. Sellise olemiga tuleb salvestamiseks lisada andmebaasi tabelisse 12 rida.

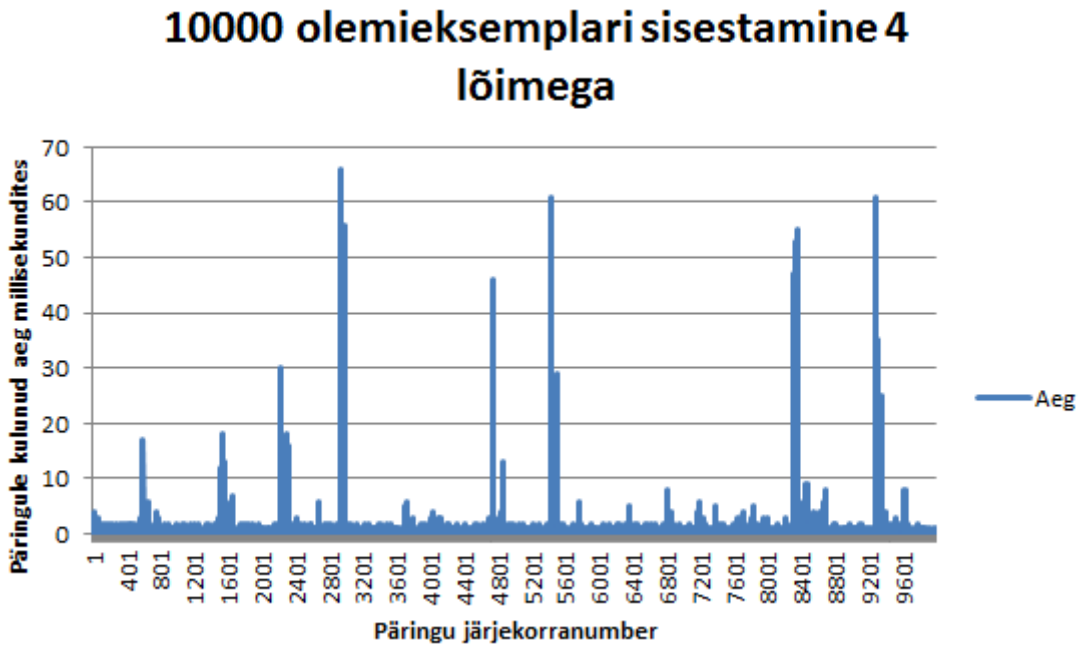
3.2.1 Olem-tunnus-väärtus andmemudeli andmesisestuse jõudlus

Sisestatakse suvaliselt genereeritud tunnuse väärtustega olemeid atribuut_väärtus tabelisse. Saadud tulemustest saame tabeli 7.

Lõimede arv		**1000 andmeobjekti (13000 rida)**	**10000 andmeobjekti (130000 rida)**	**100000 andmeobjekti (1300000 rida)**
1	Minimaalne aeg	1 ms	1 ms	1 ms
	Keskmine aeg	1 ms	2 ms	1 ms
	Maksimaalne aeg	121 ms	259 ms	371 ms
4	Minimaalne aeg	1 ms	1 ms	1 ms
	Keskmine aeg	1 ms	4 ms	2 ms
	Maksimaalne aeg	242 ms	518 ms	1120 ms

Tabel 7 - andmesisestus päringu ajad

Sarnaselt seminormaliseeritud andmemudeliga on ka sellel andmemudelil keskmine aeg maksimaalsest palju erinev. Näiteks 10000 andmeobjekti 4 lõimega sisestuse ajad tabelist 7: ei ole oluline, milliseid tulemusi vaadeldakse, sest andmed on sarnaste suurusjärkudega. Protsessori hõivatuse näitaja ei ületanud andmete salvestamisel kordagi 30% piiri. Uurime täpsemalt näiteks nelja lõimega 10 000 andmeobjekti sisestamiseks kulunud aegu.



Joonis 7- 10000 olemieksemplari sisestamine 4 lõimega

Joonise 7 tulemused on normaaljaotusega ehk enamus on koondunud keskväärtuse 1 lähedale valimi standardhälbega 2.96. Seega saab 99% tõenäosusega öelda, et suvalise andmesisestus päringu kiirus koormuse all jääb vahemikku 0.93 – 1.08 millisekundit.

3.2.2 Olem-tunnus-väärtus andmemudeli andmetagastuse jõudlus

Andmete päringu jõudluse testimiseks lisati olem-tunnus-väärtus mudeliga tabelile indeks, mis sisaldab nii atribuut kui väärtus veerge. Sellise andmemudeliga kasvab tabeli pikkus, millest väärtusi otsitakse, olemi tunnuste arv korda pikemaks kui klassikalises relatsioonilises andmemudelis. Tehtud päringute ajad kanti tabelisse 8.

Lõimede arv	**1000 andmeobjekti (12000 rida)**	**10000 andmeobjekti (120000 rida)**	**100000 andmeobjekti (1200000 rida)**
1	21 ms	171 ms	241 ms
4	keskmine 77.75 ms maksimaalne 83 ms minimaalne 61 ms	keskmine 670.625 ms maksimaalne 672 ms minimaalne 665 ms	keskmine 1155 ms maksimaalne 1172 ms minimaalne 1143 ms

Tabel 8 - andmetagastus päringu kiirused tervetele olemitele

Tabelist 8 näeme, et päringu kiirus oleneb nii andmete hulgast kui ka lõimede arvust, millega andmebaasile koormust antakse. Lisaks näeme, et mitme lõimega korraga tehtud päringute ajad on ühesuuruste andmehulkade korral suhteliselt sarnased. Järelikult ei ole koormuse all töötades päringut, mis võtaks rohkem aega kui teised.

3.3 Jõudlustestide tulemuste analüüs

Testidest erinevate andmebaasi operatsioonidega ja andmehulkade ning koormusega, saab välja tuua milline andmemudel on parim operatsiooni lõikes. Kindlasti peab arvestama, et tulemused olenevad suuresti sellest, kuidas andmemudel realiseeritud on ning missuguseid päringuid tehakse.

Rakendustele, mille eesmärk on enamasti sisestada teavet on olem-tunnus-väärtus kiireim andmemudel. Tabelist 7 on näha, et keskmine olemieksemplari salvestamisele kulunud aeg jäi 1 ms lähedale. Selle aja jooksul lisati andmebaasi 12 uut rida. Tabelis 7 toodud seminormaliseeritud mudeliga aegadest kulus ühe olemieksemplari lisamiseks keskmiselt 5 ms. Seega võib väita, et kitsa (väheste veergudega) tabeliga olem-tunnus-väärtus andmemudeliga tabelisse toimub lisamine kiiremini kui seminormaliseerituga, kuhu lisati kolm uut rida iga olemieksemplari kohta.

Rakendustele, mille põhieesmärgiks on info pärimine, on seminormaliseeritud andmemudel. Tabelist 5 näeme, et ühe olemieksemplari saamiseks kulus keskmiselt 3,7 ms ning koormuse all 12 ms. Olem-tunnus-väärtus andmemudeliga (vaata tabel 8) kõikusid tulemused suuresti teabe ja lõimete hulgast sõltuvalt, kuid olid märkimisväärselt suuremad kui seminormaliseeritud andmemudeliga. Seega saame väita, et seminormaliseeritud mudel on päringute jaoks kiirem.

Kokkuvõte

Antud bakalaureusetöö eesmärk oli uurida dünaamilisi andmemudeleid, nende esitamise viise ja jõudlust erinevate operatsioonide korral. Enamik tänapäeva rakendusi kasutavad andmebaase ning vahest ei ole võimalik teabe struktuuri defineerida, siis tuleb kasutada mõnda dünaamilist mudelit.

Võrdlusse võeti seminormaliseeritud ja olem-tunnus-väärtus andmemudel. Arutati nende olemuse üle, seletati lahti info paigutus nendes ning testiti jõudlust erineva suurusega kogude ja koormuste puhul. Töö viimases osas toodi võrdlusesse andmemudelite operatsioonide ajad ning tuvastati andmesisestuse ja -tagastuse päringute jaoks parimad mudelid.

Töö käigus selgus, et olem-tunnus-väärtus mudel on parim salvestuste jaoks ja seminormaliseeritud mudel pärimise jaoks. Kõige enam valmistas raskusi seminormaliseeritud mudeli realiseerimine ning programmeerimise liidese tegemine. Ühtlasi oli seminormaliseeritud ka kõige huvitavam uurida, sest selle kohta puudus igasugune avalik materjal ning ainuke infoallikas oli juhendaja Targo Tennisberg.

Dynamic datastructures, their implementation and performance

Bachelor's Thesis (6 ECTS)

Taivo Käsper

Summary

The goal of this thesis was to analyze the possibilities of implementing a dynamic datastructure and it's performance when inserting or selecting data. Because most of the applications developed in these days use database and it is not always possible to define the structure of the data, it is necessary to use a dynamic datamodel.

In the thesis seminormalised datamodel was compared to entity-attribute-value datamodel. Their nature was analysed, the placement of data in the tables examplified and the speed of inserts and selects compared. The last part of the thesis brought out which of the models is best for which data operations.

During the thesis it was concluded that for applications which mostly insert data, the best dynamic datamodel is entity-attribute-value model and for applications which mostly select data, the best one is seminormalized datamodel. Implementing the interface for seminormalized datamodel turned out to be the most difficult, but also the most rewarding, because of the opportunity to familiarize myself with a subject that lacks public information so far.

Kasutatud kirjandus

1. Wikipedia - Olem-tunnus-väärtus andmemudel. [Võrgumaterjal] 26. 04 2013. a. [Tsiteeritud: 11. 05 2013. a.] http://en.wikipedia.org/wiki/Entity%E2%80%93attribute%E2%80%93value_model.

2. Wikipedia - NoSQL. [Võrgumaterjal] 10. 05 2013. a. [Tsiteeritud: 11. 05 2013. a.] https://en.wikipedia.org/wiki/NoSQL.

3. **Käsper, Taivo.** Dynamic-Datastructures. [Võrgumaterjal] 11. 05 2013. a. https://bitbucket.org/taivo_kasper/dynamic-datastructures.

4. **Villems, Anne.** Courses.cs.ut. [Võrgumaterjal] 05. 03 2012. a. [Tsiteeritud: 11. 05 2013. a.] https://courses.cs.ut.ee/MTAT.03.264/2013_spring/uploads/Main/t%C3istekst_6.

5. Wikipedia - null väärtus. [Võrgumaterjal] 30. 03 2013. a. [Tsiteeritud: 11. 05 2013. a.] http://en.wikipedia.org/wiki/Null_(SQL).

6. **Rouse, Margaret.** Searchdatamanagement. [Võrgumaterjal] 09 2011. a. [Tsiteeritud: 11. 05 2013. a.] http://searchdatamanagement.techtarget.com/definition/NoSQL-Not-Only-SQL.

7. **Strauch, Christof.** [Võrgumaterjal] [Tsiteeritud: 11. 05 2013. a.] http://www.christof-strauch.de/nosqldbs.pdf.

8. Wikipedia - andmebaasi indeks. [Võrgumaterjal] 25. 04 2013. a. [Tsiteeritud: 11. 05 2013. a.] http://en.wikipedia.org/wiki/Database_index.

Lisad

Lisa 1. Terminoloogia

Tunnus

Üksik omadus mingi objekti kohta. Näiteks olgu meil objektiks auto, mille tunnusteks on kaal, kütusekulu, värv jne.

Indeks

Indekseerida saab andmebaasi tabeli mingeid veerge. Seega on indeks tabeli mingi osa koopia, mis võimaldab kiiresti leida mingite reeglite alusel reastatud andmeid [2]. Näiteks saab indekseeritud rea peal andmebaasimootor kasutada kahendotsingut, sest andmed on reastatud.

Andmemudel

Andmebaasi andmemudel on tehniline kirjeldus andmete struktuuri ja kasutamise kohta.

Kolmas normaalkuju

Definitsioon on pärit aine Andmebaaside teooria loengumaterjalidest [4]. Relatsioon R on kolmandas normaalkujus (3 NF), kui ei leidu sellist kolmikut X, Y, A, kus X on mingi relatsiooni R võti, A on sekundaarne atribuut ja kus kehtiksid väited:

1) X => Y

2) Y => A

3) Y ≠> X

Null väärtus

Null on spetsiaalne märge, mida kasutatakse SQL keeltes tähistamas seda, et tunnusel ei eksisteeri väärtust [5].

Relatsiooniline andmemudel

Relatsiooniline andmemudel on selline andmete esituse viis, kus andmed on jaotatud loogilistesse blokkidesse ehk tabelitesse. Seejärel on erinevate tabelite vahel loodud seosed

(relatsioonid), kus üks andmeobjekt viitab teises tabelis olevale andmeobjektile. Seega kujutab iga rida mingit kompleti seotud andmeid.

NoSQL

Järgnev lõik pärineb allikast [6]. NoSQL andmebaasid, vahel ka nimega “mitte ainult SQL”, on selline andmemudel, mis on kasulik suure koguse hajusate andmete hoidmiseks. NoSQL proovib lahendada neid probleeme, millega relatsioonilised andmebaasid hakkama ei saa - skaleeruvus ja suuremahuliste andmete hoiustamise jõudlus. NoSQL on kasulik kui on vaja pärida ja analüüsida suurt kogust struktureerimata andmeid või andmeid, mis asuvad pilves.

Lisa 2. Adam Milazzo kiri NoSQL andmebaaside puudustest

Adam Milazzo on (tarkvara süsteemide arhitekt ettevõttes Nortal). Küsisime A. Milazzo käest, et miks tema disainitud FlairPoint dokumendihaldussüsteem ei kasuta mõnda dokumendi andmebaasi, kus andmete struktuuri ei ole vaja defineerida.

There are several reasons why FlairPoint stores structured data instead of unstructured data. The first is simply that FlairPoint is meant to replace SharePoint, so it should support the same features. But on a more theoretical level, I think structured databases are superior to unstructured ones.

Parallel with static vs. dynamic typing

In programming languages like C++ and C#, we have well-defined classes. This is akin to structured data: all the fields are well-known, every object of a given type has the same fields, and the fields cannot be changed at runtime. Programming languages like Javascript and Perl support "classes" due simply to the fact that every object is a hash table and you can store any key/value pairs you like in the hash table. This is akin to unstructured data: you can create an object in Javascript/Perl/MongoDB by adding whatever fields you want. The fields can be different for each object and can change at runtime.

Most large software projects are written in statically typed languages for good reason. Static typing allows compile-time checks to catch many bugs. Performance is better. Development tools are better because they have more information about the program. (Compare IDEs for static and dynamic languages.) Javascript and Perl are most often used for small programs or prototypes. They're rarely used for medium-to-large projects and when they are, it's only by rigorous adherence to convention that they can avoid becoming a maintenance nightmare. But programming by convention is considered a bad idea. It's much better if your tools can automatically verify that you've done things correctly.

You have to specify the schema „somewhere“

"Unstructured" databases still have structure. It's just not defined or enforced at the database level. Instead, the work of defining and enforcing the structure must be done at the application level. When a user wants to enter data into the database, the application must still present a set of fields for the user to fill in, and must know the correct types of the fields.

In the records management project (which uses FlairPoint), the schema is not hardcoded and can be changed at runtime. This means the schema must be stored somewhere. If it's not stored in the database, then it must be stored somewhere else, so what you're really doing is storing the data in one database and the schema in another database (even if it's just a text or XML file). Why split the the schema and data into two databases?

The benefit is that you can change the schema without updating any data, and that can be helpful, but it also requires great care because the objects in the database would be a mix of many different versions of the schema. The application would have to handle all of the different versions and would have to be very careful to not make assumptions that weren't true for some older version of the schema. If the software is simply loading and displaying data, then it's no problem, but if it's doing any computations or decisions based on the data, then it's much more difficult. In my experience, programmers make hidden assumptions all the time and it would be very difficult for them to keep not just the current schema in their heads, but all older schemas as well, when writing the software. New programmers would have to learn and know not just the current schema but all previous schemas.

Furthermore, if you separate the schema from the database, the database can't guarantee that objects have the right fields or types. Once again you have to rely on the programmer to not make mistakes.

Ability of one to emulate the other

Structured databases can emulate unstructured databases and vice versa. You can easily define a key/value table in a structured database and implement an unstructured data storage system. You can even design it so that if somebody bypasses your API and writes directly to the table using SQL, they can easily change the data and can't corrupt it.

You can't easily implement a structured database in an unstructured database. You'd have to build schemas, types, constraints, etc. on top of it, which would be a lot of work, and it would be precarious because if somebody didn't go through your API, they could easily corrupt the data and even corrupt the schema.

For this reason, I think structured databases are more powerful.

Performance and maturity

The fully dynamic nature of an unstructured database makes it harder for it to support complex queries with the same level of performance. (Intuitively, you can't expect software to run as fast if all objects are hash tables with arbitrary types than if objects have static types and fields are laid out in well-known locations in memory.) Also, unstructured database software is much less mature than products like SQL Server and Oracle.

Features

Although not directly related to the structured/unstructured data debate, relational databases tend to have a lot more features: ACLs, views, triggers, stored procedures, etc. simply due to their maturity.

In conclusion

If you like to program classes by sticking function pointers and data into hash tables, then you'll like unstructured databases. :-) But if you prefer static typing, you should ask yourself why. A lot of the same reasons apply to the structured/unstructured database debate.

Where FlairPoint fits in

In addition to using a database, FlairPoint /is/ a database. (And the fact that it uses SQL Server internally is an implementation detail.) The FlairPoint database is actually somewhere between rigidly structured and unstructured databases, providing, I think, a good combination of the best features of both.

Like structured databases, each object has a well-defined schema. But like unstructured databases, objects of different schemas can be mixed together in the same container.

FlairPoint actually does support key/value pair storage if you want that. Each object has a hash table in which you can store any data that you want. The intent is that you will use static schemas for most work, but can /extend/ objects with additional dynamically typed fields. But you could also store all your data in the hash table. This way, you're not forced into one paradigm or another, but can choose the one that fits the problem best.

Unlike straight SQL databases, FlairPoint allows you to change the schema more easily, and it does a lot of work to correctly move, convert, and validate everything for you. This allows a lot of flexibility to change the schema at runtime while maintaining guarantees

about the data stored so that the application doesn’t need code to handle every historical schema version.

Like document-oriented databases, one object can contain other objects. That is, objects are arranged in a hierarchy, not just a flat table, so you can have a company object that stores employee objects, which store review objects, etc. Actually, it's a list of trees, so you can also make a flat table too if you want.

Lisa4. Testarvuti riistvara kirjeldus

Protsessor: Intel Core i5540M processor @ 2.53 GHz

RAM mälu: 8GB DDR3 1066 MHz

Kõvaketas: 640GB Wester Digital 5400rpm

Lisa 3. *Dynamic-Datastructures* liidese käivitamise käsud

Liidese ehitamiseks käsurealt järgnev *Maven* käsk: „*mvn clean package*“.

Liidese käivitamiseks seminormaliseeritud andmemudeliga kirjutada käsureale käsk: „*java -jar dynamic-datastructures-1.0.jar*“.

Liidese käivitamiseks olem-tunnus-väärtus andmemudeliga kirjutada käsureale käsk: „*java -jar dynamic-datastructures-1.0.jar -DentityAttributeValue=true*“.

Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks

Mina, Taivo Käsper (sünnikuupäev: 17.01.1991)

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) enda loodud teose „Dünaamilist andmebaasiskeemi nõudvate rakenduste andmemudelite esitamise võimalused ja iseärasused“, mille juhendaja on Anne Villems ja Targo Tennisberg,

1.1. reprodutseerimiseks, säilitamise ja üldsusele kättesaadavaks tegemise eesmärgil, sealhulgas digitaalarhiivi DSpace-is lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;

1.2. üldsusele kättesaadavaks tegemiseks Tartu Ülikooli veebikeskkonna kaudu, sealhulgas digitaalarhiivi DSpace´i kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.

2. olen teadlik, et punktis 1 nimetatud õigused jäävad alles ka autorile.

3. kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest tulenevaid õigusi.

Tartus, **11.05.2013**