TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Informaatikainstituut

IDK40LT

Aleksandr Šved IAPB

ORM-I KASUTAMISE PRAKTILISE OTSTARBEKUSE HINDAMINE TARKVARAPROJEKTIS

bakalaureusetöö

Juhendaja: Martin Rebane
MSc
Lektor

Tallinn 2016

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: Aleksandr Šved

23.05.2016

Annotatsioon

Töö eesmärk on uurida probleeme, mis on seotud ORM-i kasutamisega, ning hinnata ORM-i otstarbekust erinevates tarkvaraprojektides.

Selles töös käsitlen probleeme, mis tekivad või võivad tekkida ORM-i kasutamisel. Lisaks üritan välja tuua põhilised vead, mis on seotud ORM-i kasutamisega, ning pakkuda välja lahendusi või alternatiive. Samuti toon ka näited oma kogemusest lähtuvalt.

Kokkuvõttes võin öelda, et ORM-i kasutamise võimalused on väga laiad, kuid keerukamate probleemide puhul on vaja teada ORM väga hästi, mis lõppude lõpuks viib järgneva valikuni: kas kasutada saja protsendi ulatuses ORM-i ning pühenduda selle õppimisele, või kasutada ORM-i vaid osaliselt.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 34 leheküljel, 3 peatükki, 21 joonist, 2 tabelit.

Abstract

Evaluation of Practical Feasibility of Using ORM in a Software Project

The aim of this work is to research problems, connected with ORM usage and to evaluate practical feasibility of using ORM in different software projects.

In this work I will deal with issues that happen or can happen when working with ORM. I will try to list most common mistakes, related to ORM usage and give them a solution or alternative. Also I will bring examples from my own experience.

As for the result I can say that ORM allows very much, but for complicated cases the good knowledge of ORM is needed, which in the end leaves its user to the choice either to use one hundred percent of ORM and devote time for learning it, or to use ORM partly only.

The thesis is in Estonian and contains 34 pages of text, 3 chapters, 21 figures, 2 tables.

Lühendite ja mõistete sõnastik

API	*Application programming interface*, Rakendusliides
EF	*Entity Framework*, ORM-i raamistik C# keeles
Eager Loading	Virk laadimine
Framework	Raamistik
IDE	*Integrated Development environment*, Integreeritud arendus keskkond
JDBC	*Java Database Connectivity*, Java andmebaasipöördus
Lazy Loading	Laisk laadimine
Odoo	*Suite of business apps*, Ärirakenduste komplekt
OOP	*Object-Oriented Programming*, Objektorienteeritud programeerimine
ORM	*Object-Relational Mapping*, Objekt-relatsiooniline kaardistamine
RDBMS	*Relational database management system*, Andmebaasihaldur, mis põhineb suhtelisel modellil
SQL	*Structrured Query Language*, Struktuurpäringukeel
Thread	Lõim
XQL	*Hibernate Query Language*, Hibernate päringukeel

Sisukord

1 Sissejuhatus....10
1.1 Taust ja probleem....10
1.2 Ülesande püstitus....10
1.3 Metoodika....11
1.4 Ülevaade tööst....11
2 ORM-i analüüs ja metodoloogia....12
2.1 Abstraktsioon....12
2.2 Impedance mismatch....13
2.3 Disain....14
2.4 Andmete pärimine....16
2.4.1 Päring näidise järgi (Query-By-Example)....16
2.4.2 Päring API abil (Query-By-API)....17
2.4.3 Päring keele abil (Query-By-Language)....18
2.5 Objekti osaline pärimine....19
2.6 Päringute tegemisele kuluv aeg....20
2.7 Liiga palju päringuid....22
2.7.1 N+1 probleem....22
2.8 Testimine....23
2.9 Õppimise kõver....24
3 ORM-i jõudluse mõõtmine ja võrdlemine....26
3.1 Objekti osaline pärimine....26
3.2 Odoo ORM API....27
3.3 C# Linq päringud....30
3.4 C# ORM vs SQL....32
4 Kokkuvõte....34
Kasutatud kirjandus....35
Lisa 1 – ORM-i koodi näide....36
Lisa 2 – Pooliku SQL näide....37

Lisa 3 – Täieliku SQL näide..38

Jooniste loetelu

Joonis 1. Objektide ja tabelite seostamise näide. [2]......14
Joonis 2. Näide ORM-i tegutsemisest. Adapteeritud [4] järgi......15
Joonis 3. Näide „SQL-rääkivast“ objektist. Adapteeritud [4] järgi......16
Joonis 4. Päring näidise järgi. Adapteeritud [9] järgi......17
Joonis 5. Päring API abil. Adapteeritud [9] järgi......17
Joonis 6. Päring API abil ilma tabelite nimesid. Adapteeritud [9] järgi......18
Joonis 7. Päring keele abil. Adapteeritud [9] järgi......18
Joonis 8. Päringu aja jaotuse diagramm [8]......21
Joonis 9. Päringule kulutatud aja jaotuse diagramm (ilma programmi käivitamisele mineva ajata) [8]......21
Joonis 10. Näide N+1 päringutest......23
Joonis 11. Näide kuidas N+1 päring peaks tegelikult käivituma......23
Joonis 12. Lihtsustatud ja üldistatud diagramm......28
Joonis 13. Vale ORM-i kasutamine......31
Joonis 14. Tekitatud SQL päring......31
Joonis 15. Halb ORM-i näide......31
Joonis 16. Korduv SQL päring......32
Joonis 17. LINQ ja SQL lahenduste ajalise võrdlemise graafik......32
Joonis 18. ORM-i koodi näide......36
Joonis 19. Pooliku SQL näide......37
Joonis 20. Täieliku SQL näide esimene osa......38
Joonis 21. Täieliku SQL näide teine osa......39

Tabelite loetelu

Tabel 1. Päringute käivitamise ajad EF abil. Adapteeritud [8] järgi................................20
Tabel 2. Odoo pärngute käivitamise ajad...30

1 Sissejuhatus

Tänapäeval on ORM põhiline tehnika, relatsioonandmebaaside struktuuri seostamiseks objektorienteeritud keele objektidega. ORM-i peetakse kõige lihtsamaks ja tihti ka kõige paremaks viisiks selle seostamiseks. Seda on väga lihtne kasutusse võtta ning alguses tundub see kerge ja arusaadav. Kui aga ORM-i veidi kauem kasutada, siis mida raskemaks programm ja päringud lähevad, seda rohkem tekib probleeme, mille lahenduse leidmiseks kulub palju aega, ning lahendus ei pruugi olla kõige efektiivsem. Selle töö eesmärk on leida olulisemad probleemid, mis tekivad ORM-i kasutamisel ning pakkuda välja võimalusi nende lahendamiseks.

1.1 Taust ja probleem

ORM on viimaste aastate jooksul kogunud populaarsust, ning tänapäeval kasutatakse seda väga paljudes olemasolevates arenduskeskkondades, ning siiamaani võetakse kasutusse ka uutes. ORM-i kasutamisel tekib hulk probleeme, millele vastuseid leida on väga keeruline, eriti eesti keeles. Käesoleva töö autor ei suutnud leida mingisugust andmeallikat, kus oleksid välja toodud olulisemad ORM-i probleemid ning ka neile lahendused välja pakutud. Selle bakalaureusetöö eesmärk on jõudluse probleemide tekkimise olukorras arendajaid aidata ning anda ka mõtteid nendele, kes alles on valiku ees, kas võtta ORM kasutusse või mitte.

1.2 Ülesande püstitus

Käesolev bakalaureusetöö seab endale eesmargiks uurida ORM-i kasutamise otstarbekust kasutusmugavuse ja jõudluse seisukohast, uurida tekkivaid probleeme ning pakkuda neile lahendust.

Ülesande sisuks on määratleda olulisemad probleemid, mis on seotud ORM-i kasutamisega, selgitada, milles seisneb probleem ning leida igale probleemile sobiv lahendus, mida saaks ka reaalselt teostada.

1.3 Metoodika

Püstitatud eesmärkide saavutamiseks otsin ja analüüsin olemasolevaid probleeme, mis on seotud ORM-i kasutamisega. Probleemid võivad olla nii üldised, nagu näiteks struktuur, kui ka spetsiifilised, nagu töötamise kiirus erijuhtudel. Lahenduseks on kas vastus või siis soovitus kuidas mingit asja saab paremini teha.

Probleemide ja nende lahenduste otsimiseks kasutan internetis olemasolevaid allikaid ja oma teadmisi Hibernate, Entity Framework ja LINQ kasutamisest, millega olen töökohustustega seoses korduvalt kokku puutunud kolme aasta jooksul. Lisaks projektidele, kus on kasutusel ORM tehnika tavapärasel kujul, olen töötanud ka Odoo ORM API-ga, kus kõik on ehitatud ülesse ORM-i abil, sealhulgas ka vaated.

Lisaks üldisele ORM-ile vaatlen ka erinevaid programmeerimiskeele spetsiifilisi framework-e, nagu Hibernate, mis on hetkel kõige populaarsem kasutuses olev ORM framework, ning ka Entity Framework ja LINQ, mis on peamised C# ORM framework-id.

Lisaks eelnevale koostan ma näidisprogramme, mille peal testin erinevaid päringuid kasutades ORM framework-e Entity Framework ja LINQ ning võrdlen tulemusi puhta SQL päringute tulemustega.

1.4 Ülevaade tööst

Esimeses osas analüüsin ORM-i ja toon välja selle miinused ja plussid. Seejärel vaatlen näiteid oma töökogemusest lähtuvalt ja loon ka eraldiseisvaid näiteid.

2 ORM-i analüüs ja metodoloogia

2.1 Abstraktsioon

ORM pakub abstraktsiooni, mis peaks lihtsustama päringute tegemise, kuid reaalselt peab ORM-i kasutaja lisaks SQL keelele ära õppima ka teise keele (näiteks HQL), mille abil päringuid teha. Abstraktsioon on lekkiv, ehk kui avada ükskõik milline ORM dokumentatsioon, siis selgub, et see sisaldab hulgaliselt viiteid SQL põhimõtetele. Kuigi ORM annab abstraktsiooni, siis SQL-i teadmata on suhteliselt võimatu ORM-i kasutada.

Samuti on mõnedel juhtudel väga keeruline kirjutada ORM-i abil päringut. Abstraktsioon võib varjata nii palju, et ei ole arusaadav, kuidas ta käitub ja mida teeb. Sellistel juhtudel kasutatakse toores SQL-i, ehk kirjutatakse otse SQL lauseid ORM abstraktsiooni kasutamata, mis omakorda tekitab segadust, kuna osa päringuid on kirjutatud ühtemoodi, ning teine osa teistmoodi.

Tihtipeale tekib vajadus optimeerida päringuid, kuna andmete hulk aina kasvab. Sellisel juhul on kõige lihtsam võimalus aru saada kus on kõige kriitilisem koht - vaadata päringute SQL lauseid. See tähendab, et tuleks ORM laused tagasi SQL keelde tõlkida, mis teeb abstraktsiooni mõttetuks [1].

Samuti peaks arendaja, kes kasutab ORM-i, täpselt aru saama mida ja kuidas ta teeb ning milline on tema kirjutatud päringute mõju. Ainult sellisel juhul on võimalik neid õigesti kirjutada. Tekib küsimus, milleks on üldse ORM-i vaja, kui arendaja peab ikkagi hästi tundma andmebaase ja SQL keelt [1]?

Enamus ORM framework-e toetab mitmeid andmebaase, sealhulgas ka uusi. Abstraktsioon aitab kontsentreeruda päringule, mitte andmebaasi keelele, mille kaudu ORM suhtleb andmebaasiga. See tähendab, et baaside muutmine ei tohiks vigu tekitada.

2.2 Impedance mismatch

Impedance Mismatch, või „ebakõlane impendants“ on konseptuaalsete ja tehniliste raskuste kogum, mis tekivad olukordades, kui programm, mis on kirjutatud objektorienteeritud programmeerimiskeeles kasutab relatsioonilist andmebaasi, ehk teisisõnu kui üritatakse viia vastavusse programmi objekte ja andmebaasi tabeleid. Probleem tekib selle tõttu, et objektide ja relatsioonide vahel on väga nõrk suhe [6].

Relatsioonilises mudelis on tabelid, mis hoiavad informatsiooni ridade kaupa. Tabelite vahel võivad olla seosed (üks-ühele, üks-mitmele ja mitu-mitmele). Tabelid on omavahel seotud võtmete abil. Primaarvõti (Primary Key) on võti, mis identifitseerib kirje ning välisvõti (Foreign Key) viitab sellele. Kogu äriloogika on väljaspool tabeleid ja käivitub protseduuride ja trigerite abil. Selleks, et kirjutada äriloogikat on vajalikud tabelite struktuur ja seosed.

Teiselt poolt, OOP koosneb objektidest. Objektid hoiavad samuti informatsiooni, kuid objektide sees on ka äriloogika, mida kutsutakse välja meetodite abil. Koodi arendamiseks ei ole vaja teada, mis on objekti sees ja kuidas see täpselt töötab, kuna liidese kaudu saab kutsuda klassi meetodeid teadmata, mis seal sees on. Teine suur erinevus on selles, et seotud objektid ei pea olema eraldi klassides, vaid võivad olla klassi sees, või klassi pärida, ning võtmeid seostamiseks ei vaja.

Kui üritada viia vastavusse OO objektid ja tabelid, siis kõige parem lahendus oleks viia objekti parameetrid ja tabeli veerud vastavusse üks-ühele, kuid hästi kirjutatud OO objekti mudel juba veidi keerulises süsteemis võib vajada klassi, mis vastab mitmele tabelile, kus igaühest on võetud mingi osa või vastupidi, mitu klassi võivad vastata samale tabelile vastavalt äriloogika vajadustele, mille puhul tihtipeale tuleb tabelis teha uus veerg, mis sisaldab klassi nimetust. Kui üritada neid ideaalselt vastavusse viia, siis tuleb väga palju muuta mõlemat mudelit, mis mõjub halvasti üldisele pildile ning teeb edasise arendamise raskemaks.

Kuna tabelites kasutatakse primaarvõtmeid, milleks tihti on ID, siis see peab kajastuma ka objektides. Kui objekt ja tabel on vastavuses üks-ühele, siis objekt võib kasutada andmebaasi ID genereerimiseks, mis ei tohiks probleeme tekitada, kuid [7] järgi, kui ühele objektile vastab mitu tabelit, siis objekti salvestamisel tuleb uuendada kõiki

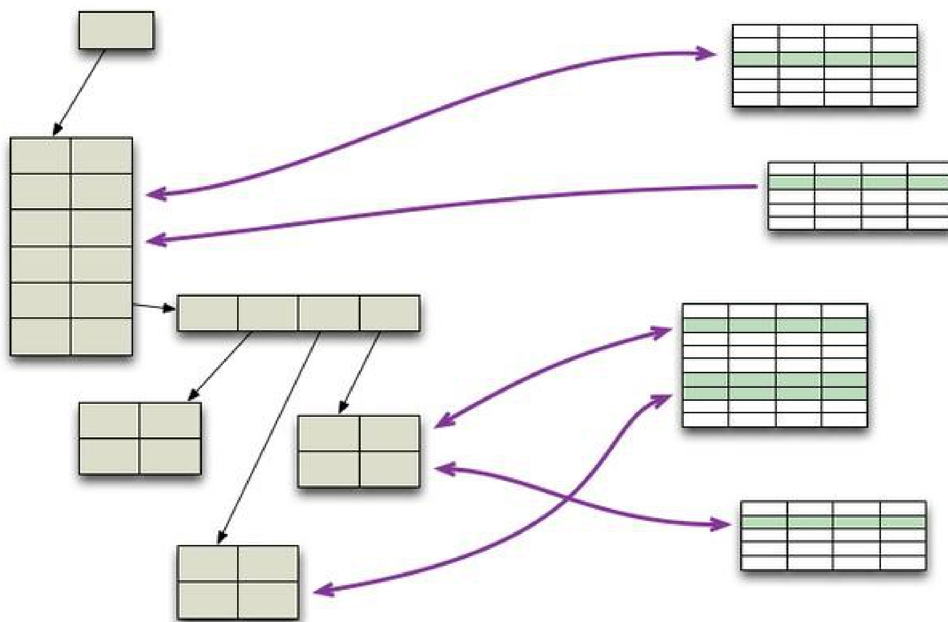
tabeleid ID järgi, mis peavad olema grupeeritud ühe ID järgi. Selleks tuleb teha veel üks tabel, kus ühele ID-le, mis on kasutusel klassis vastab mitu teist ID-d, mis on tabelite ID-d.

Alternatiiviks on kasutada mitme tabeli asemel ühte tabelit, mis sisaldab mõlema tabeli informatsiooni. Sellise lähenemise abil rea otsimisel ei ole vaja ühendada tabeleid omavahel.

Kui andmebaasil on kasutusel väljade valideerimine ja muu kontroll, siis see peab olema ka üleviidud objektidele.

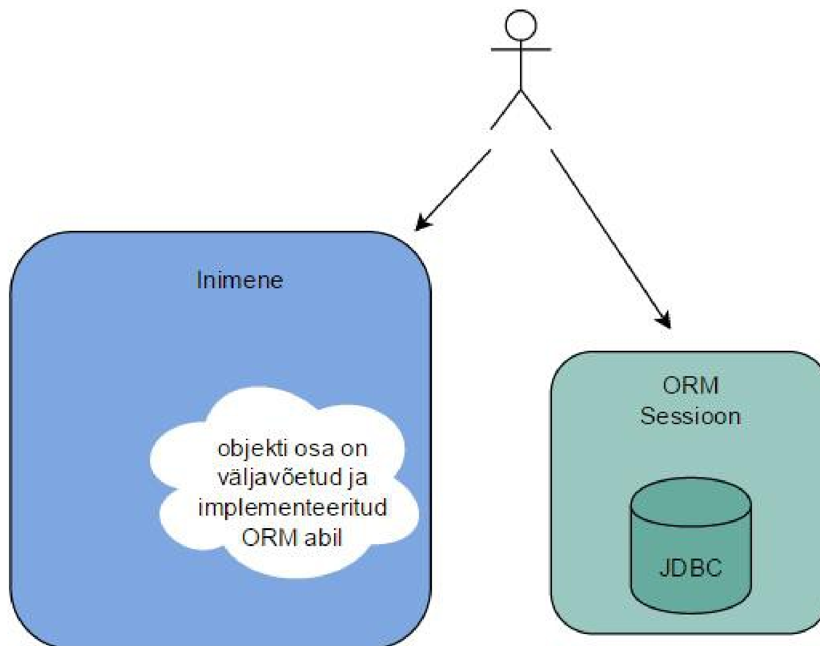
2.3 Disain

Tugev sõltuvus ORM tarkvarast on välja toodud kui halva andmebaaside disaini üks põhjuseid. Kuna ORM üritab seostada relatsioonilist andmebaasi ja andmeid, mida kasutatakse koodis, siis tekivad probleemid ja raskused, ning koostatakse seostamine, mis teeb asja veel keerulisemaks. Samuti mainib Martin [2], et veel rohkem keerukust lisab see, et andmeväljad võivad muutuda nii andmebaasis, kui ka programmis, ning seda võivad tahta teha mitu inimest korraga.



Joonis 1. Objektide ja tabelite seostamise näide. [2]

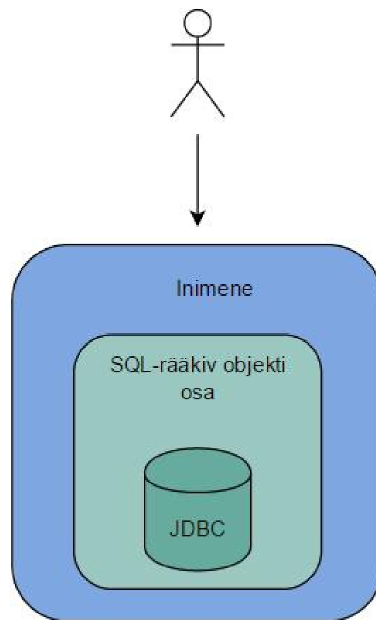
Yegori [4] väitel tükeldab ORM, selle asemel, et kapseldada andmebaasi päringuid ühte objekti, otseses mõttes sidusa süsteemi kaheks osaks. Üks osa objektist hoiab andmeid ja teine, implementeeritud ORM mootori sees, teab kuidas tuleb andmeid töödelda ja andmebaasi sisestada. Järgmine pilt näitab, kuidas ORM tegutseb.



Joonis 2. Näide ORM-i tegutsemisest. Adapteeritud [4] järgi

ORM-i kasutaja peab tegutsema kahe komponendiga: ORM-iga ja tükeldatud objektiga, mida kasutaja saab tagasi, kuigi ideaalis see võiks olla üks OOP objekt. Just selle disainivea tõttu tekib ka enamus probleeme, millest mõnda on ka vaadeldud selles töös, näiteks: ORM-i aeglasus, andmebaasi uuendamine ja testimine on keerukus, SQL ei ole peidetud.

Alternatiivina pakub Yegor [4] välja „SQL-rääkivaid" objekte, kus objekt, mis on täielikult kapseldatud, suhtleb andmebaasiga SQL abil.



Joonis 3. Näide „SQL-rääkivast" objektist. Adapteeritud [4] järgi

Yegori [4] idee on selles, et meil on kaks erinevat klassi iga tabeli kohta, millest üks on ainsuses ja teine mitmuses, näiteks „Inimene" ja „Inimesed". „Inimesed" esindab andmebaasi tabelit ja „Inimene" tabeli rida. Klassis, kus töödeldakse inimesi, ehk näiteks inimese otsing ja inimese lisamine, ei ole midagi selle kohta, millise baasiga suhtlus käib, ning ei ole teadagi, et suhtlus on baasiga, sest see võib olla ka tekstifaili muutmine. Kogu implementatsioon on peidetud. Andmebaasi päringute implementatsioonid võivad olla mingisuguse lihtsa andmebaasi suhtlus, näiteks JDBC Java programmerimiskeeles. Peamine, et kogu implementatsioon on peidetud sinna. Selline lähenemine tundub olema rohkem objektorienteeritud. Objekt on hästi kapseldatud ja suhtleb SQL kaudu.

2.4 Andmete pärimine

Järgmises kolmes alamosas ma toon välja Ted Neward-i mõtteid populaarsest blogist [9]. Selleks, et pärida andmebaasist andmeid on mitu erinevat varianti:

2.4.1 Päring näidise järgi (Query-By-Example)

Sellise lähenemise puhul tuleb teha objekt ja täita väljad, mille järgi tuleks otsida. Näiteks, kui otsida inimeste tabelist inimest, kellel on perekonnanimi „Kask" siis tuleb teha midagi sellesarnast:

```
Inimene inimene = new Inimene(); //kõik väljad on null
inimene.Perenimi = "Kask";
ObjectCollection oc = QueryExecutor.Execute(inimene);
```

Joonis 4. Päring näidise järgi. Adapteeritud [9] järgi

Sellised päringud on lihtsad ja arusaadavad, kuid kui minna veidigi keerulisema näite juurde, siis selle realiseerimine on raske. Ühe näitena toob Ted [9] välja olukorra, kus on vaja pärida inimesi, kelle perekonnanimi on kas „Kask“ või „Vaher“. Kahte eraldi päringut kasutades on see triviaalne, kuid kui üritada seda teostada ühe päringuga näidise järgi, siis see muutub päris keeruliseks. Teiseks näiteks on olukord, kui on vaja otsida inimesi, kelle perekonnanimi ei ole „Kask“. Need kaks näidet on tehtavad, kuid nende realiseerimine on liiga keerukas, kusjuures tuleb võtta arvesse, et päringud ise on väga lihtsad. Teine takistus on see, et kõik väljad peavad lubama NULL väärtusi.

2.4.2 Päring API abil (Query-By-API)

Esimesest variandist on välja kujunenud päring API abil. Sellel juhul luuakse päring päringu objektidest.

```
Query query = new Query();
query.From("INIMENE").Where(
    new EqualsCriteria("INIMENE.PERENIMI", "Kask"));
ObjectCollection oc = QueryExecutor.Execute(query);
```

Joonis 5. Päring API abil. Adapteeritud [9] järgi

Sellel juhul ei tehta päringut näidise pealt, kuid võetakse kõik andmed, ning neid juba filtreeritakse erinevate kriteeriumite abil. Erinevad kriteeriumid on ühendatud loogiliste „Jah“ ja „Või“ seostega. Samuti saab kasutada nii grupeerimist, kui ka sortimist. Sellised päringud peavad olema päris täpsed ja tulevad välja suhteliselt suurena.

Ted [9] märgib, et selline lähenemine on palju arusaadavam tavalistest SQL päringutest. See teeb arendamise ja päringu kirjutamise lihtsamaks. Kui aga kirjutada mingisuguseid spetsiifilisi päringuid (näiteks OUTER JOIN), siis selgub, et päringud API abil lähevad kas liiga keeruliseks, või on neid üldse võimatu kirjutada. Lisaks peavad kõik tabelite ja veerude nimetused olema koodis kirjutatud, mis tekitab vea võimaluse kuna inimene võib kogemata valesti sõna sisestada. Seda võib vältida testimist kasutades, kuid neid teste tuleb käivitada andmebaasi vastu, mis ei ole päris lihtne ülesanne.

Eelmine variant ei too nii hästi välja loogikat, mille järgi päringut tehakse, vaid pigem näitab objektide omavahelisi seoseid. Selleks, et teha asja arusaadavamaks, kirjutavad arendajad midagi sellist:

```
Query query = new Query();
Field lastNameFieldFromPerson = Inimene.GetDeclaredField("perenimi");
query.From(Inimene).Where(
    new EqualsCriteria(lastNameFieldFromPerson, "Kask"));
ObjectCollection oc = QueryExecutor.Execute(query);
```

Joonis 6. Päring API abil ilma tabelite nimesid. Adapteeritud [9] järgi

Selline lähenemine Ted-i [9] sõnul aitab vältida ka ohtu, et mingi nimetus võib olla valesti sisse trükitud, kuid ei lahenda probleemi keerukate päringute kirjutamisel (näiteks mitmete tabelite pealt admete pärimine filtreerides andmeid mõlemas tabelis).

2.4.3 Päring keele abil (Query-By-Language)

Järgmise lähenemise puhul on üritatud rohkem objekte ja meetodeid kasutada, mis tõstab koodi haldavust ja arusaadavust.

```
SELECT Inimene i1, Inimene i2
FROM Inimene
WHERE i1.getSpouse() == null
  AND i2.getSpouse() == null
  AND i1.isThisAnAcceptableSpouce(i2)
  AND i2.isThisAnAcceptableSpouce(i1);
```

Joonis 7. Päring keele abil. Adapteeritud [9] järgi

Meetod, mida kasutatakse on objekti „Inimene“ meetod, ning ei ole arusaadav, kas selline meetod oleks võimalik päringute keeles ja kas üldse oleks vaja, et see oleks võimalik. Selline lähenemine võimaldab kirjutada ka keerulisi päringuid, kuid kuna iga päringu jaoks on vaja kõiki päringus olevaid kirjeid transformeerida objektideks, et käivitada meetod, siis see tekitab probleeme kiirusega. Kiiruse tõstmiseks pakub Ted [9] välja kaks varianti: kodeerida vajalikud andmed eraldi tabelisse ja lisada need päringusse, mille tulemuseks on leheküljepikkused päringud või tekitada andmebaasi protseduure, mida käivitatakse, kuid see eemaldab koodi objektist, ning jätab kogu loogika andmebaasi.

2.5 Objekti osaline pärimine

Väga paljudel tekib olukord, kui on vaja optimeerida programmi, et ta käivituks kiiremini. Andmebaasi puhul see on päringute optimeerimine ning üheks optimeerimisviisiks on pärida ainult neid andmeid mida vajatakse. Üks levinud juhtum on kui näidatakse nimekirja inimestest, ning ühte inimest valides juba näidetakse kõiki tema andmeid. Kui inimeste tabelis on igasuguseid andmeid inimese kohta, nagu nimi, perekonnanimi, sünnipäev, isikukood, kodune aadress, telefon ja teised, siis kui meil on vaja ainult nime, perekonnanime ja sünnikuupäeva, siis ei ole otstarbekas kõiki andmeid pärida, kuna see lisab päringu aega.

Kuna ORM-i üheks ülesandeks on näidata arendajale objekti ja võimaldada teha tööd ainult objekte kasutades, siis tekib probleem, kuna ei ole võimalik teada saada, milliseid välju hakatakse objektis kasutama, eriti kuna objekti tihtipeale saadetakse ühest meetodist teisse. Selleks, et ikkagi lubada seda teha, toob Ted [9] välja kaks varianti, mis on mõlemad puudustega. Esimene variant on lubada objekti väljadeks null väärtused, isegi siis kui see väli ei tohiks tühi olla. Teine on kasutada „laiska laadimist“ mis laeb andmeid ainult siis, kui neid kasutatakse, aga mis võib kaasa tuua N+1 probleemi.

Selleks et vältida N+1 probleemi, on võimalik kasutada seadistusi. Tavaliselt kasutatakse globaalseid seadistusi, mille puhul seadistatakse igat klassi, mis aga ei võimalda ühte klassi erinevates päringutes erinevalt kasutada. Lisaks, võimaldab laisk laadimine tavaliselt laadida kas kõik väljad või mitte ühtegi, mis ikka ei lahenda varianti, kus oleks vaja pärida ainult mõnda välja.

Väljade asemel võivad olla ka teised objektid, millel on omad väljad. See olukord tekib siis, kui päritakse objekti tabelist, millel on seosed teiste tabelitega (üks-ühele, üks-mitmele, mitu-ühele või mitu-mitmele). Sellisel juhul on väga raske otsustada, millal laadida kõik andmed ja millal mitte ja on suhteliselt lihtne seda valesti teha ning laadida üleliigseid andmeid, või jätta mõned andmed laadimata [9].

2.6 Päringute tegemisele kuluv aeg

Kuna ORM ei kasuta valmisolevaid päringuid ja tekitab neid päringuid alles käivitamisel, siis selleks, et käivitada ORM-i päringut esimest korda on vaja päringut kompileerida. Peale esimest käivitamist salvestatakse päring vahemällu, ehk järgmised päringu käivitamised võtavad juba vähem aega. Mida keerulisem on päring, seda rohkem aega kulub selle kompileerimisele. Näiteks on [11] räägittud sellest, et päring kompileerus 12 kuni 15 sekundiga, kusjuures ilma päringu käivitamisele kuluva aja arvesse võtmata. On muidugi olemas võimalused kompileerida päringuid enne nende käivitamist ja salvestada need vahemällu, aga see ei ole alati sobilik lahendus ja kohati on seda ka raske teostada. Üks võimalus on üritada hoiduda keerulistest päringutest ja kõik ära teha vaid lihtsate päringute abil. Selle tulemuseks on hulgaliselt koodi kirjutamist, mis võtab palju rohkem arendamisaega, kui samaväärse päringu kirjutamine.

Lisaks peab esimese päringu käivitamisel iga rakenduse domeeni kohta kompileeruma ka kogu andmebaasi mudel, mis võib võtta enam kui pool minutit keerulise mudeli puhul. David Roth [10] toob välja kolm erinevat lahendust sellele probleemile, kuid nad ikkagi ei lahenda probleemi lõplikult ning kaks kolmest variandist vajavad vahemällu salvestamist, mis ei ole alati võimalik mälupuuduse tõttu.

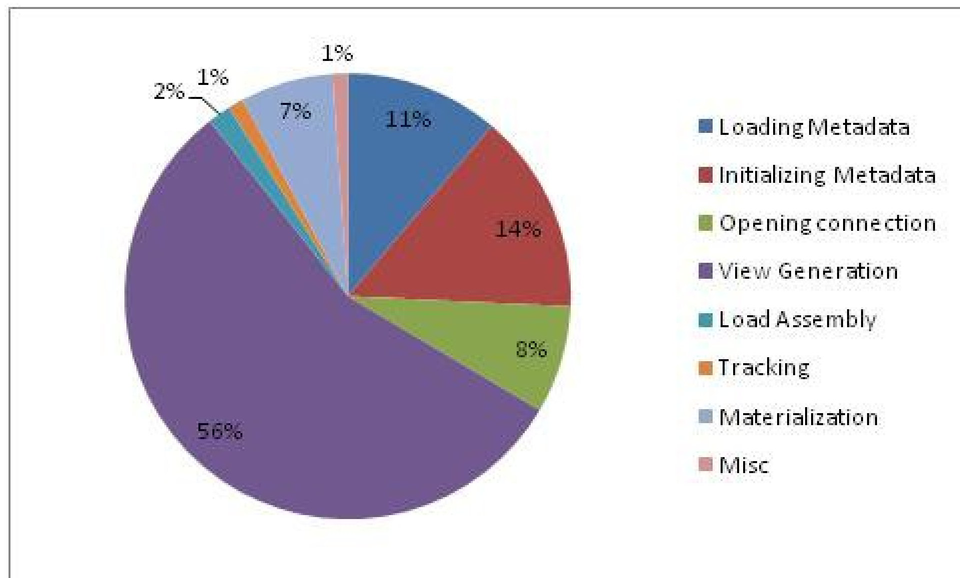
Sellele eelneb veel metaandmete laadimine [13], mis võib samuti võtta palju aega, ning mida ei ole võimalik vahemällu laadida enne päringu käivitamist. Üks lahendus on kirjeldatud [12] vastuses, mis rakenduse käivitamisel paneb käima lihtsa päringu, mis paneb metaandmete laadimise tööle, kuid see ei ole võimalik iga rakenduse puhul, eriti kui rakenduse käivitamisel on kohe vaja andmebaasi päringuid teha.

Brian Dawson-i blogis [8] on välja toodud lihtsa päringu käivitamise ajad suhteliselt lihtsa andmemudeli korral kasutades Entity Framework-i:

Tabel 1. Päringute käivitamise ajad EF abil. Adapteeritud [8] järgi

Esimene käivitamine (ms)	Järgmised 9 (ms)								
4241	13	13	14	13	13	13	13	24	14

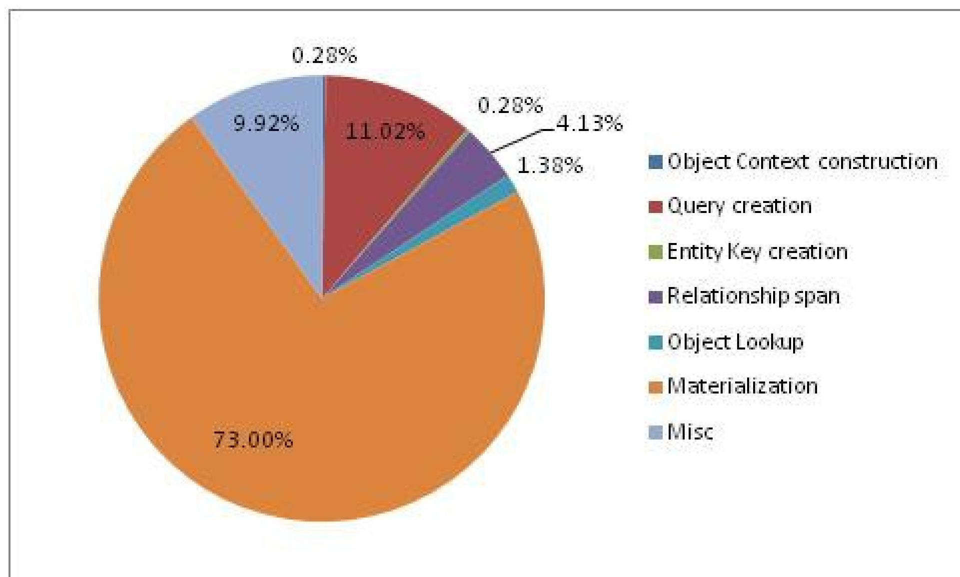
Siin on väga hästi näha, kuidas esimene päring võtab kordades rohkem aega, kui järgmised. Samas blogis tõi Brian Dawson [8] välja diagrammi, kus on näha, millele see aeg läheb:



Joonis 8. Päringu aja jaotuse diagramm [8]

Siit on näha, et veidi üle poole ajast läheb mudeli ja vaadete genereerimisele. Samuti läheb suur osa metaandmete genereerimisele.

Kui eemaldada aeg, mis kulub käivitamisele, siis ülejäänud aeg jaguneb järgmiselt:



Joonis 9. Päringule kulutatud aja jaotuse diagramm (ilma programmi käivitamisele mineva ajata) [8]

Sellelt diagrammilt on näha, et enamus ajast läheb päringu andmete objektideks tegemisele. Suurusjärgult teisel kohal, kuid mitmekordselt vähem aega kulub päringu loomisele, mis on keeruliste päringute puhul veel suurem.

Ülaltoodud diagrammidelt on näha, et suurem osa päringutele kuluvast ajast läheb sellele, millele tavalise SQL päringu kirjutamisel aega ei lähe. Väljatoodud põhjuste tõttu ORM-i päringute jõudlus on madalam, kui sama päringute jõudlus SQL abil, ehk kui rakenduses on kriitilised kohad, kus on käivitamise aeg väga oluline, siis seal oleks mõistlikum kasutada puhast SQL-i ORM-i asemel.

2.7 Liiga palju päringuid

Väga tihti teeb ORM mitmeid päringuid andmebaasi selle asemel, et teha ühe päringu, mis koormab andmebaasi üle ja teeb ORM-i kasutamise ebaefektiivseks.

Üheks lahenduseks on salvestada päringute tulemused vahemälusse, mis lubaks järgmiste päringute käivitamisel andmeid võtta mitte andmebaasist, vaid vahemälust, mis kiirendab tööd ja vähendab päringute arvu. Kuid sellisel lähenemisel on kaks probleemi. Esiteks, andmete muudatus ei kajastu vahemälus, ehk kui andmed muutuvad tihti, siis neid tuleb ka tihti andmebaasist lugeda Teiseks, tuleb esimesel korral ikkagi lugeda andmebaasist kõike.

2.7.1 N+1 probleem

Üheks levinumaks probleemiks on „N+1 probleem“. Võtame näiteks kaks tabelit: ühes on inimesed ja teises särgid, mis viitavad inimesele, kes seda omab, ehk ühele inimesele vastab mitu särki. Kui on vaja leida igale inimesele kuuluvaid särke, siis selle asemel, et teha ühe JOIN päringu, ORM teeb päringu kogu inimeste tabeli lugemiseks, itereerib selle üle ja teeb iga inimese kohta päringu särkide tabelisse. Ehk kokku teeb see n+1 päringut, kus n on inimeste arv. Kui inimeste arv on piisavalt suur, siis sellised päringud koormavad andmebaasi, ning kui andmebaas on mingil teisel masinal, millele päringu tegemine võtab näiteks 1-2ms, siis 1000 päringu tegemine võtab aega 1-2s, mis on juba märgatav ka kasutajale.

Sellele probleemile on mitmeid lahendusi, kuid need sõltuvad sellest, mille tõttu täpselt probleem tekib.

Kui on vaja itereerida üle kõiki inimesi, siis selleks, et N+1 probleem ei tekiks tuleb alguses laadida peale inimeste ka särkide andmeid ja ainult siis itereerida. Seda nimetatakse „Eager Loading“. Sellisel juhul tehakse alati kaks päringut, ühe inimeste ja teine särkide saamiseks, sõltumata sellest mis objekte on lõpptulemuses vaja, ja siis töödeldakse neid, näiteks

```
SELECT * FROM 'Inimesed'
SELECT * FROM 'Särgid' WHERE 'Inimesed'.'id' = 1;
SELECT * FROM 'Särgid' WHERE 'Inimesed'.'id' = 2;
SELECT * FROM 'Särgid' WHERE 'Inimesed'.'id' = 3;
SELECT * FROM 'Särgid' WHERE 'Inimesed'.'id' = 4;
```

Joonis 10. Näide N+1 päringutest

asemel teeb

```
SELECT * FROM 'Inimesed'
SELECT * FROM 'Särgid' WHERE 'Inimesed'.'id' IN(1,2,3,4);
```

Joonis 11. Näide kuidas N+1 päring peaks tegelikult käivituma

Kuid seda lahendust ei saa kasutada kõigi päringute tegemisel, kuna paljudes kohtades ei ole vaja sõltuvate tabelite sisu, kuna see suurendab andmete suurust ning teeb päringuid aeglasemaks.

2.8 Testimine

Kuna andmebaasiga suhtlemine ja programmi loogika on seotud, siis ORM objekte on väga raske komponenttestida ilma andmebaasita. Kuna komponenttestid sobivad rohkem isoleeritud objektide jaoks, mis on üks OOP kapseldamise mõistetest ning ORM-i puhul on objekt seotud andmebaasiga, siis vahel ORM objektide testimine võib olla väga keeruline.

Üheks väljapakutud variandiks [5] on koostada eraldi andmebaas testandmetega, mida kasutada testimiseks ning kontrollida päringu tulemusi. Peale testpäringute käivitamist viiakse andmebaas esialgsesse seisundisse selleks, et järgmine kord saaks kasutada samu teste. Sellisel lähenemisel on mitu probleemi. Esiteks, tabelite muutmisel tuleb

neid muuta ka testandmebaasis, ning kuna see automaatselt ei muutu, siis tihtipeale jääb testandmebaasi midagi panemata. Teiseks, andmebaasis on arendaja poolt sisestatud andmed ning paljud veakohad võivad jääda testimata selle pärast, et erinevat viisi sisendit ei ole testitud.

Teiseks variandiks on võtta testandmebaasiks reaalse andmebaasi koopia, kus saaks teha kõike just nii, nagu seda tehakse ka programmis. Probleemiks on see, et kui andmebaasi andmed muutuvad, siis tuleb muuta ka testandmebaasi, mis võib põhjustada testide katki minemist, kuna on vaja kontrollida vastust, kuid andmeid pole teada.

2.9 Õppimise kõver

Üheks suureks probleemiks, millega väga tihti ei arvestata, on õppimise kõver ehk õppimise raskus. Teiste sõnadega: kui palju aega kulub sellele, et osata kasutada õiget funktsionaalsust õiges kohas. Selleks, et alustada ORM-i kasutamist ei ole väga palju teadmisi vaja, kuid kui läheb veidi keerulisemaks, siis võib tekkida olukordi, kus ORM-i kasutamine ei tasu ära selle pärast, et ei suudeta õiget lahendust välja käia, ning selleks, et osata seda teha kulub aega rohkem, kui lahendada olukorda teisel viisil, mis ei oleks jõudluse poolest halvem. Tuleb osata kasutada selliseid lähenemisi nagu lazy-loading, fetching strateegiad, esimese ja teise taseme vahemälu kasutamine, päringute vahemälu ja nii edasi. Lisaks, iga ORM omab erinevat API-d, mis tähendab seda, et iga uue ORM-i kasutamisega tuleb uuesti õppida, ning kuna ORM on lekkiv abstraktsioon, siis ka SQL õppimisest ei pääse [4].

Sellele probleemile otsest lahendust ei ole. Enne ORM-i kasutusele võtmist tuleb kaaluda, kas see tasub ära või mitte, eriti kui teadmisi ja oskusi piisavalt ei ole, kuna mingil hetkel võib tekkida olukord, kus on vaja keerulisemaid päringuid mis vajavad lisateadmisi ning aega, mis omakorda toovad ebaefektiivsust arendamisel.

Ted Neward kirjutas oma populaarses blogis [9], et ORM on nagu Vietnam arvutiteaduses. Selle all mõtles ta seda, et ORM tundub algul olema väga hea valik. Arendamine liigub kiirelt, lihtsalt ja arusaadavalt, kuid mida suuremaks läheb skoop, seda raskemaks ja ajakulukamaks läheb arusaamine ja arendamine. Mingil hetkel hakkab iga muudatus võtma suure hulga ajast, mis on mitmekordselt suurem võrreldes

sellega, mis ta algul oli. Kõige halvem on see, et tagasiteed ei ole. Ainuke võimalus on kõik otsast peale alustada, kuid see tähendab, et kogu ORM kasutamine oli puhas aja raiskamine.

3 ORM-i jõudluse mõõtmine ja võrdlemine

3.1 Objekti osaline pärimine

Selleks, et testida kas andmete osaline pärimine säästab aega ja kui säästab, siis kui palju, tegin ma lihtsaid päringuid suurte tabelite peal.

Kasutasin Microsoft SQL Server Management Studio päringute tegemiseks ning ajamõõtmiseks kasutasin SQL Server Profiler. Testimiseks võtsin kaks tabelit, mida reaalselt kasutatakse programmis. Igat päringut testisin mitu korda ja võtsin viimase tulemuse, et vältida erinevust sõltuvalt sellest, kas salvestatakse andmeid vähemällu, või mitte.

Esimeseks testimiseks kasutasin tabelit, millel oli 26 erinevat veergu ja pärisin 100.000 rida.

Kõigi andmete pärimine võttis aega 11.568 ms, kui ainult kolme mitteindekseeritud välja pärimine võttis 5.111 ms, mis on ligi kaks korda kiirem.

Teiseks testimiseks ma kasutasin tabelit, millel oli 72 erinevat veergu ja pärisin 100.000 rida.

Kõigi andmete pärimine võttis aega 2.353 ms, kui ainult kolme mitteindekseeritud välja pärimine võttis 371 ms, mis on enam kui kuuekordne vahe.

Kolmandaks testimiseks ma kasutasin sama tabelit, mis ka teisel testimisel, kuid pärisin 1.000.000 rida.

Kõigi andmete pärimine võttis aega 20.764 ms, kui ainult kolme mitteindekseeritud välja pärimine võttis 4.772 ms, mis on enam kui neljakordne vahe.

Need testimised näitasid, et osaline pärimine on kasulik ning võib vähendada päringu aega mitmekordselt isegi ühe tabeli puhul, rääkimata juba sellest, et võtta tabeliga

seotuid tabeleid ja pärida ka nende välju. Selline suur vahe on suhteliselt kriitiline, eriti kui rääkida kasutatavusest.

3.2 Odoo ORM API

Kuna Odoo ORM API kasutab väga tihedalt ORM-i, siis iga vaate jaoks tekitatakse uus tabel, kus ridades on väljad, nupud ja kirjeldused. Suurem hulk tabelitest on ajutised mida mõne aja pärast kustutatakse ära. Teisiti öeldes on ajutised tabelid vahemälu asenduseks. Üks muredest on see, et iga väikese muudatuse jaoks on vaja teha muudatus ka baasis. Lisaks sellele, kui on vaja mitmest tabelist JOIN teha, siis see läheb:

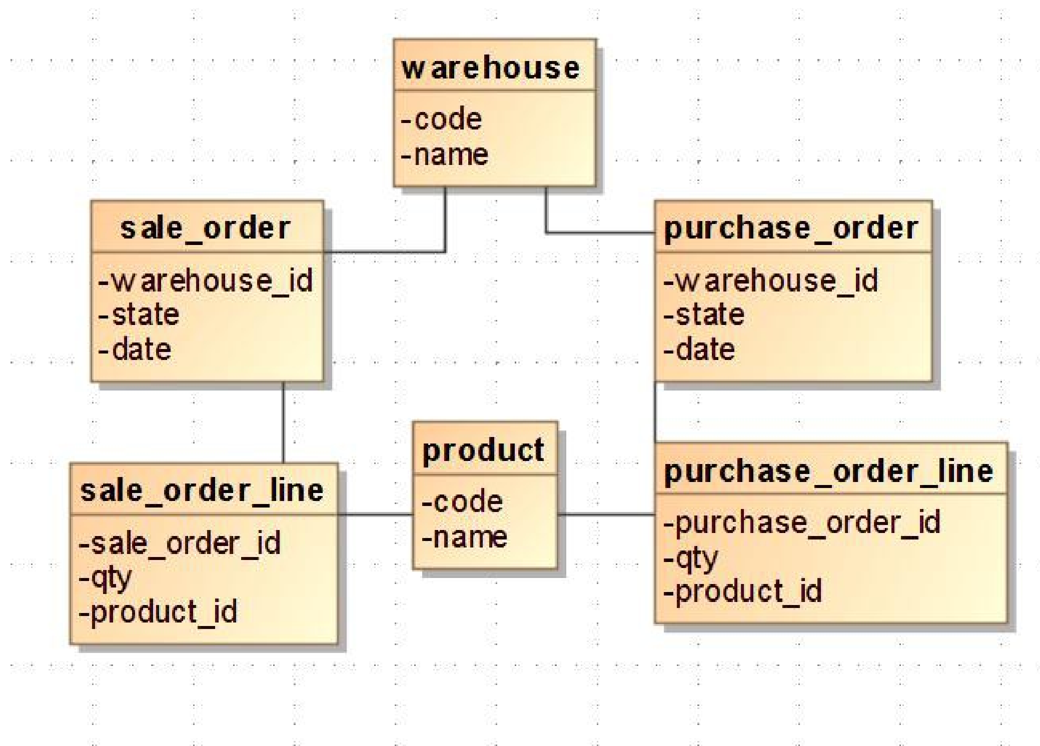
1. liiga keeruliseks
2. vajab liiga palju mälumahtu

Järgmisena toon välja näite ülesandest, millega olen isiklikult töökohustustega seoses kokku puutunud.

Alguses veidi taustainformatsiooni: riigi kohta on mitusada poodi, kus igas poes müüakse tuhandeid tooteid. Poed ostavad aeg-ajalt uusi tooteid, mis tulevad poodi sisse, ning igapäevaselt müüvad neid edasi klientidele. Iga ost (purchase_order) võib sisaldada mitu erinevat toote-koguse paari (purchase_order_line). Sama kehtib ka müükide kohta.

Kood on kirjutatud python keeles ning andmebaasiks on postgresql. SQL päringute tegemiseks kasutatakse Psycopg driver-it ning ORM-i lahendamiseks kasutatakse Odoo ORM API.

Eesmärgiks on võimaldada koostada väljavõtte õnnestunud ostude ja müükide koguste koguarvust iga päeva kohta mis on grupeeritud poe ja toote järgi. Samuti peab olema võimalik filtreerida alguse ja lõppu perioodi järgi ning poodide järgi.



Joonis 12. Lihtsustatud ja üldistatud diagramm

Kuna poode ja tooteid on palju ning suuremat osa poe toodetest ei osteta/müüda igapäevaselt, siis nende järgi andmete väljavõtmine ei ole mõistlik. Ehk andmed tuleb välja võtta sale_order ja purchase_order tabelite põhjal.

Selle ülesande lahendamiseks proovisin algul kasutada ORM-i, kuid paari tunni möödudes sain aru, et mul ei ole selleks piisavalt palju oskusi. Peale kogenud töökaaslasega suhtlemist jõudsime otsusele, et teeme puhta SQL päringu, kuna keerulisemad ORM-i päringud võivad olla palju aeglasemad või saata tuhandeid päringuid paari asemel, kui veidigi valesti koostatud.

Tegime otsuse kõrgema efektiivsuse kasuks ehk üritasime väiksema ajakuluga lahendada probleemi selle asemel, et üritada ORM-ist läbi murda.

Vaatamata sellele, et ülesanne sai lahendatud puhta SQL abil, otsustasin, et proovin sama probleemi lahendada ka ORM-i abil ja võrdlen lahenduste tulemusi ja aega, mis läks arendamiseks. Kui üritasin ORM-i abil lahendada, siis kulutades kokkuvõttes kolm korda rohkem aega see siiski mul ei õnnestunud. Selle ajaga jõudsin välja võtta ostud ja müügid ja arvutada välja nende arv teatud kindla ajaperioodi jooksul, ehk täielikust lahendusest oli puudu grupeerimine poe, toote ja päeva järgi ning müükide ja ostude

kokku viimine, mis SQL teeb ise. Kusjuures ORM-i abil võetud ostud ja müügid on päritud eraldi, mis peaks tõstma kiirust, aga tegema arendamise keerulisemaks.

Nüüd võrdleme kahte erinevat viisi probleemi lahendamiseks ning proovime võrrelda erinevate tunnuste järgi.

Koodi loetavus: ORM-i abil kirjutatud päringud on palju kompaktsemad ja selle tõttu kood on paremini loetavam, kuna kogu pilt on silme ees. Mõlemad päringud on üherealised, kuid samas võib koodi loetavus ka halvemaks minna, kui jõuda lõpptulemuseni. Puhas SQL päring mahub napilt ühele leheküljele. Samas, kui eemaldada grupeerimise osa ja väljad, mida ei kasutata pooliku ORM-i lahenduse jaoks, siis SQL päring on kompaktsem ja võtab poole vähem ruumi. Objektkeele jaoks ei ole loetavus kõige parem, kuid SQL lause ei ole ka liiga keeruline, et koodi loetavust väga halvaks teha.

Koodi haldavus: siinkohal on kindlasti eelis ORM-i abil kirjutatud päringul, kuna esiteks on kergem aru saada, mida tehakse ja seetõttu ka kergem muudatusi teha; teiseks, näiteks veeru muutmise puhul ei ole ORM-i päringus vaja midagi muuta, kuna muudatust saab teha mudeli peal, kuid SQL päringut tuleb käsitsi muuta, et ta sobiks. Kolmandaks, näiteks tabeli nimetuse muutmisel tavapäevased IDE-d aitavad koodi refaktoreerida, ehk refaktoreerimise töörista abil saab väga kergelt muuta koodi ORM-i päringus, kuid SQL päringute refaktoreerimine ei ole nii kerge, kuna see on programmi jaoks lihtsalt tekst, ehk tabeli või veeru nime muutmisel tuleb kõik SQL päringud läbi käia ja muuta seal nimetuse käsitsi.

Arenduse kiirus: selle konkreetse olukorra puhul tuli välja, et puhta SQL abil oli probleemi lahendamine kordades kiirem, vähemalt minu ja minu töökaaslaste puhul. Samas, kui seda osa hiljem hallata ja muuta mitmekordselt, siis muudatuste tegemine puhta SQL puhul on ajakulukam. See tähendab, et pikas perspektiivis, kui seda koodiosa tihti muudetakse ja täiendatakse, võib ORM-i abil probleemi lahendamine osutada mõistlikum ajakulu mõttes.

Päringute kiirus: Kuna ORM-i päringud ei lahenda probleemi täielikult, siis võrrelda aegu on keeruline. Selleks, et võrdlemine oleks õigem ma veidi korrigeerisin SQL päringu, et ta teeks sama asja, mida ORM-i päring teeb ja mõõtsin kõiki kolme. Aja

mõõtmiseks kasutasin koodis olevat taimerit, mis pandi tööle enne päringu käivitamist ja peatati selle lõppemisel. Mõõtmise tegin testandmebaasi põhjal, kus on suhteliselt vähe andmeid (paarsada müüki ja ostu). Tulemused on järgmised:

Tabel 2. Odoo pärngute käivitamise ajad

	ORM	Poolik SQL	Täielik SQL
Esimene katse	177 ms	22 ms	31 ms
Teine katse	135 ms	13 ms	36 ms
Kolmas katse	186 ms	22 ms	38 ms
Neljas katse	153 ms	21 ms	30 ms
Viies katse	172 ms	18 ms	39 ms
Kuues katse	178 ms	19 ms	37 ms
Seitsmes katse	170 ms	15 ms	35 ms
Kaheksas katse	183 ms	22 ms	35 ms
Üheksas katse	169 ms	21 ms	38 ms
Kümnes katse	189 ms	22 ms	38 ms
Keskmine	**171ms**	**20 ms**	**36 ms**

Nagu tabelist on näha, ORM-i päringud võtavad mitmekordselt rohkem aega. Samas, kuna numbrid on nii väiksed, siis ühekordne päringu tegemine ei mõjuta kusataja kogemust ning ei ole märgatav. Lisades 1, 2 ja 3 on ORM-i, pooliku SQL ja täieliku SQL koodiosad päringutest.

3.3 C# Linq päringud

Järgmistes näidetes ma muutsin sisu, kuna see on konfidentsiaalne informatsioon, mille avaldamiseks mul puuduvad õigused, kuid üritasin loogilist sisu maksimaalselt alles jätta.

Probleemiks oli liiga aeglane lehe laadimine, mille tõttu hakkasid laekuma klientide kaebused. Peale uurimist selgus, et lehe laadimise tegid aeglaseks tuhanded päringud, mida põhjustas ORM-i vale kasutamine.

Esiteks:

```
items = items.Where(
    i => customers.Contains(i.CustomerVID) || i.CustomersMaps != null &&
    t.CustomersMaps.Any(m => !m.Deleted && customers.Contains(m.CustomerVID)));
```

Joonis 13. Vale ORM-i kasutamine

See osa koodist tekitas järgmise SQL päringu, mis kasutab EXISTS lauset:

```
WHERE (EXISTS(
SELECT NULL AS [EMPTY]
FROM [CustomersMap] AS [t6]
WHERE (NOT ([t6].[Deleted] = 1))))
AND (EXISTS(
SELECT NULL AS [EMPTY]
FROM [CustomersMap] AS [t7]
WHERE (NOT ([t7].[Deleted] = 1)) AND ([t7].[CustomerVID] IN @p3)))
```

Joonis 14. Tekitatud SQL päring

EXISTS kasutamine küll aeglustab päringuid, kuid vahe ei ole nii suur, ehk see ei olnud peamine mõjufaktor.

Teiseks:

```
public void SpecialItems(Item givenItem)
{
    var newItem = new Item
    {
        RelatedCustomers = Db.Items.Where(i => !i.Deleted && i.VID == givenItem.VID)
                            .Select(
                                i => new KeyValue
                                {
                                    Vid = i.Customer.VID,
                                    Name = i.Cusomer.Name
                                }).Distinct()
                            .Union(
                                givenItem.CustomersMap.Where(m => !m.Deleted).
                                          Select(
                                              m => new KeyValue
                                              {
                                                  Vid = m.Customer.VID,
                                                  Name = m.Customer.Name
                                              })).Distinct().ToList(),
        Model = givenItem.ItemModel.Model
    };
}
```

Joonis 15. Halb ORM-i näide

Andtud näide tekitas järgmiseid päringuid iga givenItem-i kohta:

```sql
SELECT DISTINCT [t5].[VID] AS  [Vid], [t5].[Name]
FROM (
SELECT  [t4].[VID], [t4].[Name]
FROM (
SELECT DISTINCT [t1].[VID], [t1].[Name]
FROM [dbo].[Item] AS [t0]
INNER JOIN [dbo].[Customer] AS [t1] ON [t1].[VID] = [t0].[CustomerVID]
WHERE (NOT ([t0].[Deleted] = 1)) AND ([t0].[VID] = @x1)
UNION
SELECT  [t3].[VID],   [t3].[Name]
FROM [dbo].[CustomersMap] AS [t2]
INNER JOIN [dbo].[Customer] AS [t3] ON [t3].[VID] = [t2].[CustomerVID]
WHERE (NOT ([t2].[Deleted] = 1)) AND ([t2].[ItemVID] = @x1)
) AS [t4]
) AS [t5]
```

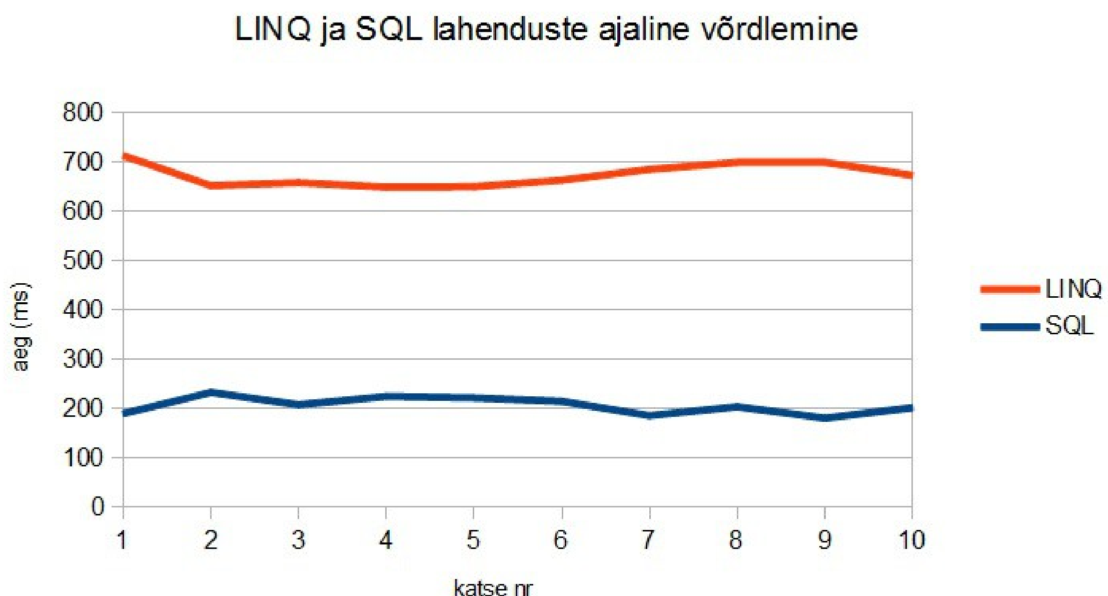
Joonis 16. Korduv SQL päring

Ehk see põhjustas sadu, või mõnel juhul ka tuhandeid sarnaseid päringuid.

3.4 C# ORM vs SQL

Järgmisena võtsin ette juba olemasoleva koodiosa, kus lahendus on kirjutatud LINQ abil. Kirjutasin samaväärse koodi, kuid lahendasin seda puhta SQL abil. Käivitasin korduvalt mõlemat ja võrdlesin tulemusi.

Päringu idee on välja võtta mudelid koos tootja ja kategooria andmetega, millel ei ole ühtegi toodet küljes.



Joonis 17. LINQ ja SQL lahenduste ajalise võrdlemise graafik

LINQ abil lahenduse keskmine töötamise aeg on 675 ms.

SQL abil lahenduse keskmine töötamise aeg on 207 ms.

LINQ ja SQL ajaline vahe selles näites on kolmekordne. Tasub küll mainimist, et tegu ei ole väga suure andmehulgaga. Päringus võetakse 100 esimest järjestatud tulemust, kuid terve päring tagastab veidi üle selle arvu ridu. SQL päringus võetakse kõik väljad välja, et see oleks võrdväärne LINQ päringuga, kuigi tegelikult vaja läheb palju vähem andmeid. Samas LINQ abil tehtud lahendus on arusaadavam ja kergemini hallatav.

4 Kokkuvõte

Töö põhieesmärgiks oli hinnata ORM-i otstarbekust tarkvaraprojektides ja võrrelda alternatiivlahendustega.

Tulemuseks võib öelda, et ORM võimaldab väga palju, kuid vajab ka palju oskusi keerukamatel juhtudel. Kas kasutada ORM-i või mitte sõltub väga palju konkreetsest olukorrast. On juhtumeid, kus ORM sobib paremini ja on ka neid, kus ORM ei sobi üldse. Samas, tihti saab kasutada osaliselt ORM-i, ning nendes osades, kus ORM ei sobi kasutada näiteks puhast SQL-i.

ORM sobib väga hästi, kui on vaja lihtsat asja teha, ning projekti alguses tõstab see arendamiskiirust. Samuti on ORM-i palju kergem hallata. Teiselt küljest, kui kiirus on väga kriitiline, siis puhas SQL sobib palju paremini. Lisaks, väga keerulised päringud võivad ORM-i abil kirjutamisel olla väga ajakulukad, eriti kui ei ole piisavalt kogemusi.

Kasutatud kirjandus

[1] http://mindref.blogspot.com.ee/2013/02/sql-vs-orm.html (23.05.2016)

[2] http://martinfowler.com/bliki/OrmHate.html (23.05.2015)

[3] http://www.kenneth-truyers.net/2014/11/15/how-to-ditch-your-orm/ (23.05.2016)

[4] http://java.dzone.com/articles/orm-offensive-anti-pattern (23.05.2016)

[5] http://stackoverflow.com/questions/145131/whats-the-best-strategy-for-unit-testing-database-driven-applications (23.05.2016)

[6] http://en.wikipedia.org/wiki/Object-relational_impedance_mismatch (23.05.2016)

[7] http://www.odbms.org/wp-content/uploads/2013/11/023.01-Shusman-The-Impedance-Mismatch-2002.pdf (23.05.2016)

[8] http://blogs.msdn.com/b/adonet/archive/2008/02/04/exploring-the-performance-of-the-ado-net-entity-framework-part-1.aspx (23.05.2016)

[9] http://blogs.tedneward.com/post/the-vietnam-of-computer-science/ (23.05.2016)

[10] https://www.fusonic.net/en/blog/3-steps-for-fast-entityframework-6.1-code-first-startup-performance/ (23.05.2016)

[11] http://stackoverflow.com/questions/2739461/how-to-reduce-entity-framework-4-query-compile-time (23.05.2016)

[12] http://stackoverflow.com/questions/3891125/entity-framework-first-query-slow (23.05.2016)

[13] https://msdn.microsoft.com/en-us/library/cc853327(v=vs.100).aspx (23.05.2016)

Lisa 1 – ORM-i koodi näide

```
data = request.env['sale.order'].search([('state', 'not in', ('draft',
'sent','cancel')), ('date_order', '>=', '2015-01-01'), ('date_order', '<=',
'2015-02-01')])
data_2 = request.env['purchase.order'].search([('state', 'not in', ('draft',
'sent','cancel')), ('date_order', '>=', '2015-01-01'), ('date_order', '<=',
'2015-02-01')])
first = 0
for solo in data:
	for one in solo.order_line:
		first = first + one.product_uom_qty;
second = 0
for solo in data_2:
	for one in solo.order_line:
		second = second + one.product_qty;
```

Joonis 18. ORM-i koodi näide.

Lisa 2 – Pooliku SQL näide

```
SELECT  sum(qty_sold) as qty_sold, sum(qty_ordered_manually) as qty_ordered
FROM (
	SELECT
	sum (sl.product_uom_qty) as qty_sold,
	0 as qty_ordered,
	FROM sale_order_line sl
	INNER JOIN sale_order o ON o.id = sl.order_id
	INNER JOIN product_product p ON p.id = sl.product_id
	INNER JOIN stock_warehouse wh ON wh.id = o.warehouse_id
	WHERE o.state not in ('draft', 'sent', 'cancel') {0}
UNION ALL
	SELECT
	0 as qty_sold,
	sum (pl.product_qty) as qty_ordered,
	FROM purchase_order_line pl
	INNER JOIN purchase_order o ON o.id = pl.order_id
	INNER JOIN product_product p ON p.id = pl.product_id
	INNER JOIN stock_location l ON l.id = o.location_id
	INNER JOIN stock_warehouse wh
	INNER JOIN stock_location loc ON loc.id  = wh.view_location_id
	on loc.parent_left <= l.parent_left AND loc.parent_right >= 
l.parent_right
	WHERE o.state in ('approved', 'except_picking', 'except_invoice', 
'done') {0}
) d
```

Joonis 19. Pooliku SQL näide

Lisa 3 – Täieliku SQL näide

```
SELECT date, shop_code, shop_name, product_code, product_name, sum(qty_sold)
as qty_sold, sum(qty_ordered_manually) as qty_ordered_manually,
sum(qty_ordered_by_VMI) as qty_ordered_by_VMI
FROM (
	SELECT
	to_char(date_trunc('day', o.date_order), 'YYYY-MM-DD') as Date,
	wh.code as Shop_code,
	wh.name as Shop_name,
	p.default_code as Product_code,
	p.name_template as Product_name,
	sum (sl.product_uom_qty) as qty_sold,
	0 as qty_ordered_manually,
	0 as qty_ordered_by_VMI
	FROM sale_order_line sl
	INNER JOIN sale_order o ON o.id = sl.order_id
	INNER JOIN product_product p ON p.id = sl.product_id
	INNER JOIN stock_warehouse wh ON wh.id = o.warehouse_id
	WHERE o.state not in ('draft', 'sent', 'cancel') {0}
	GROUP BY to_char(date_trunc('day', o.date_order), 'YYYY-MM-DD'),
	shop_code, Shop_name, Product_code, Product_name
```

Joonis 20. Täieliku SQL näide esimene osa

```
UNION ALL
	SELECT
	to_char(date_trunc('day', o.date_order), 'YYYY-MM-DD') as Date,
	wh.code as Shop_code,
	wh.name as Shop_name,
	p.default_code as Product_code,
	p.name_template as Product_name,
	0 as qty_sold,
	sum (CASE WHEN o.type = 'manual' THEN pl.product_qty ELSE 0 END) as
ty_ordered_manually,
	sum (CASE WHEN o.type = 'automatic' AND o.src_system = 'VMI' THEN
pl.product_qty ELSE 0 END) as qty_ordered_by_VMI
	FROM purchase_order_line pl
	INNER JOIN purchase_order o ON o.id = pl.order_id
	INNER JOIN product_product p ON p.id = pl.product_id
	INNER JOIN stock_location l ON l.id = o.location_id
	INNER JOIN stock_warehouse wh
	INNER JOIN stock_location loc ON loc.id  = wh.view_location_id
	on loc.parent_left <= l.parent_left AND loc.parent_right >=
l.parent_right
	WHERE o.state in ('approved', 'except_picking', 'except_invoice',
'done') {0}
	GROUP BY to_char(date_trunc('day', o.date_order), 'YYYY-MM-DD'),
shop_code, Shop_name, Product_code, Product_name
) d

GROUP BY date, shop_code, shop_name, product_code, product_name
ORDER BY date desc, shop_code, product_code
```

Joonis 21. Täieliku SQL näide teine osa