TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Arvutiteaduse instituut
Informaatika eriala

SIIM KINKS

SQLITE ORM TEEK ANDROIDILE

Magistritöö

Juhendaja: dots. Juhan Ernits

Autor: .... "....." jaanuar 2016

Juhendaja: .... "....." jaanuar 2016

Kinnitaja: .... "....." jaanuar 2016

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.

....	....
(kuupäev)	(allkiri)

Annotatsioon

Käesoleva töö põhieesmärgiks oli luua Android SQLite ORM teek, mille töökiirus, mälukasutus, rakendamise ja API kasutuslihtsus oleks paremad, kui olemasolevatel sarnastel lahendustel. Sooviti luua teek, milles ei kasutataks ORM teekide seas levinud *reflectionit* ning mis massist eristumiseks pakuks kogukonna tänapäeva trendidele vastavaid lisafunktsionaalsusi, mida konkurentidel pole.

Töö tulemusena kaardistati olemasolevate lahenduste põhilised omadused, koostati detailne nõuete nimekiri ning realiseeriti sellele vastav teek nimega SqliteMagic. Töös antakse ülevaade valdkonnast ja motivatsiooni tagamaadest ning analüüsitakse erinevaid olemasolevaid lahendusi andmebaasi suhtluse korraldamiseks. Kirjeldatakse realiseeritud teegi funktsionaalsusi ning nende teostusdetaile. Lisaks valideeritakse töö tulemusi, võrreldes loodud teegi mälukasutust ja operatsioonide kiirust põhikonkurentidega.

Teegi jõudlustestidega valideerimisel selgub, et loodud teek vastab eesmärkidele võrdluses alternatiivsete lahendustega ning on konkurentidega võrreldes stabiilsem ning paremini optimeeritud reaalelulisteks olukordadeks. Lisaks, tänu realiseeritud API-le ja kasutuselevõtmise lihtsusele, tagatakse väiksem rakenduse arendamisele kuluv aeg.

Abstract

The main purpose of the current thesis is to create an Android SQLite ORM library, which is better in operation speed, memory usage, integration to projects and API usability than any other similar solution. The desired result is a library, where reflection, which is very common in ORM libraries, is not used at all. In addition, in order to differentiate from other approaches, offers extra features that correspond to latest community trends, which competitors are lacking.

As a result, a library called SqliteMagic was implemented, which meets all the specified requirements. The thesis gives an overview of the domain and motivation background; also, analyzes different existing solutions for organizing interaction with a database. The implemented library features are detailed and accompanied with explanations and justifications. In addition, the results of the study are validated by comparing memory usage and operation speeds of the created library with the main competitors.

As the performance analysis shows, the created library meets all the given requirements and compared to competitors, is much more stable and optimized for real life scenarios. In addition, thanks to the simplicity of the implemented API and integration to projects, much smaller application development times are ensured.

Sisukord

Sissejuhatus **8**

1 Olemasolevad Android SQLite ORM lahendused **12**
1.1 Sugar ORM . . . 12
1.2 greenDAO . . . 14
1.3 DBFlow . . . 16

2 Realiseeritud teegi kirjeldus ja analüüs **20**
2.1 Teegi paigaldus . . . 21
2.2 Andmebaasi defineerimine . . . 23
2.2.1 Java andmetüübid . . . 25
2.2.2 Kasutaja defineeritud andmetüübid . . . 26
2.2.3 Tüübiteisendajad . . . 27
2.2.4 Muutumatud objektid . . . 28
2.3 Androidi annotatsioonide tugi . . . 30
2.4 API disain ja põhimõtted . . . 30
2.5 Andmete salvestamine ja muutmine . . . 31
2.5.1 Salvestamine . . . 33
2.5.2 Uuendamine . . . 35
2.5.3 Kustutamine . . . 36
2.6 Andmete pärimine . . . 38
2.7 Paralleeltöötlus . . . 40
2.7.1 Tabeli andmete muudatuste teavitamine . . . 41
2.7.2 RxJava mugavusoperaatorid . . . 42
2.8 Kasutaja abistamine . . . 42
2.9 Andmebaasi migratsioon . . . 43

3 Teostusdetailid **44**
3.1 Annotatsiooniprotsessor . . . 44
3.2 *Reflectioni* kõrvaldamine . . . 44
3.2.1 Teegi ja genereeritud koodi ühendamine . . . 45
3.2.2 Rakenduse ja genereeritud koodi ühendamine . . . 46
3.2.3 Lahenduse töölesaamine Android Studios . . . 48
3.3 Androidispetsiifilised optimeeringud . . . 49
3.3.1 SqliteMagic teegis rakendatud optimeeringud . . . 50

4 Loodud teegi jõudluse analüüs võrdluses alternatiividega **52**
4.1 Kiirus . . . 53
4.1.1 Sisestus . . . 53
4.1.2 Andmete uuendamine . . . 55
4.1.3 Tundmatu sisestus . . . 56
4.1.4 Pärimine . . . 58
4.2 Mälukasutus . . . 60
4.2.1 Sisestus . . . 61
4.2.2 Andmete uuendamine . . . 62
4.2.3 Tundmatu sisestus . . . 63
4.2.4 Pärimine . . . 64

4.3 Analüüsi tulemused 66

5 Teegi tulevik ja edasised arendused **67**
5.1 Tüübiohutu päringute API 67
5.2 Automaatne andmebaasi migratsioon 67

Kokkuvõte **68**

Viited **69**

A Lisa: meetodite jälgimisvaated **71**

B Lisa: mälueraldusvaated **74**

Joonised

1 SqliteMagic teegi genereeritud meetodid vaadatuna Android Studios. . . . 33
2 SqliteMagic teegi genereeritud meetodid vaadatuna Android Studios enne ja peale SqliteMagic IntelliJ plugina paigaldamist. 49
3 Neljaastmelise sõltuvusega andmetüüp. 53
4 Andmete sisestus. 54
5 Andmete uuendus. 55
6 Andmete tundmatu salvestamine tühja tabelisse. 56
7 Andmete tundmatu salvestamine täis tabelisse. 57
8 Andmete salvestamine, kui pole teda, kas tuleb uuendada või lisada. 57
9 Tabeli esimese kirje pärimine. 58
10 Kõikide andmete pärimine. 59
11 Andmete sisestuse mälukasutus. 61
12 Andmete uuenduse mälukasutus. 62
13 Andmete salvestamise mälukasutus, kui pole teada kas tuleb uuendada või sisestada. 63
14 Tabeli esimese kirje pärimise mälukasutus. 64
15 Kõikide andmete pärimise mälukasutus. 65
16 SqliteMagic, DBFlow ja greenDAO uuendamisoperatsiooni meetodite jälgimisvaade. 71
17 SqliteMagic, DBFlow ja greenDAO sisestamisoperatsiooni meetodite jälgimisvaade. 72
18 SqliteMagic, DBFlow ja greenDAO kõikide andmete pärimise operatsiooni meetodite jälgimisvaade. 73
19 Sisestamisoperatsiooni allokeeritud objektide vaade. 74
20 Uuendamisoperatsiooni allokeeritud objektide vaade. 74
21 Esimese kirje pärimise allokeeritud objektide vaade. 75
22 Kõikide kirjete pärimise allokeeritud objektide vaade. 76

Sissejuhatus

Iga vähegi keerukam rakendus koosneb mitemetest autonoomsetest osadest ehk moodulitest. Nendeks võivad olla näiteks serveriga suhtlemisemoodul, piltide laadimismoodul, andmete serialiseerimismoodul, andmebaasiga suhtlemismoodul jne. Tagamaks, et kõik osad teevad õiget asja korrektselt tuleb neid valideerida ja põhjalikult testida. Kõik eelpool nimetatu on väga suur ja aeganõudev töö, mis parema lahenduse puudumisel tuleks iga rakenduse jaoks uuesti teha. Lahenduseks on kindlat probleemi lahendavad teegid, mida saab laiemas perspektiivis jaotada kaheks – vabavaralised ja tasulised. Kogu Androidi platvorm on rajatud vabavaralisele tarkvarale, mistõttu on ka Androidi arendajate kogukonnas saanud tavaks vabavaraliste teekide loomine ja kasutamine. Tänu kirjeldatule ning asjaolule, et käesolev töö keskendub Androidi platvormile tarkvara loomisele, siis järgnevalt keskendutakse ainult vabavaralisele tarkvarale.

Kahjuks pole probleemi lahendava vabavaralise teegi valik kerge. Enamasti on valikus mitmeid projekte – näiteks antud töö kirjutamise hetkel on Android ORM[1] teeke veidi alla kolmekümne. Et tiheda konkurentsi seas esile tõusta, tuleb teha, kas midagi paremini, kui kõik teised või läheneda probleemile hoopis uue nurga alt. Sellest tulenevalt on ühele probleemile mitmeid erinevaid lahendusi.

Hea avatud lähetekoodiga projekti tunnused

Kuna vabavaraliste teekide projekte on üpriski palju siis nende seast sobiva valiku tegemine võib olla küllaltki aeganõudev ja keeruline tegevus. Android kogukonnas võtab head vabavaraliste projektide tavad kokku näiteks järgmine arvamusliidrite plenaarettekanne ühel tüüpilisel Androidi arendajatele suunatud konverentsil DroidCon [30]. Järgnevalt on välja toodud nimetatud ettekandest mõned tähtsamad aspektid, mis iseloomustavad nii-öelda „head" projekti ja mida kogukond jälgib teegi valimisel. Järgnevalt kirjeldatut üritatakse ka antud töö teostamisel järgida.

Lihtsus

Väga olulist rolli teegi valikul mängib selle API[2] lihtsus ja intuitiivsus. Mida lihtsam on API, seda vähem peab kulutama aega õppimisele ja seda lihtsam on seda projektis kasutada. Lisaks võib hästimõistetav API olla otsustav faktor teegi kaasamiseks projekti. Näiteks, kui on valikus kaks teeki – ühel keeruline API, mis teeb oma tööd kiiremini ja teisel lihtsam API, kuid aeglasem töökiirus, siis enamasti võetakse kasutusele lihtsama APIga teek. Viimase põhjuseks on aeg – kõik tahavad oma projekte võimalikult kiirelt valmis saada ning kolmanda osapoole teegi API keerulisus kindlasti aeglustab töö valmimiskiirust.

Teisalt mängib olulist rolli projekti üleüldine keerulisus ehk see, kui kiiresti saadakse aru, mis „kapoti" all toimub. Näiteks, kui otsustatakse investeerida aeg mingi teegi õppimiseks,

[1] *Object-Relational Mapping* – objekt-relatsiooniline kaardistamine ehk objektorienteeritud domeeni mudeli sidumine relatsioonilise andmebaasiga.

[2] *Application Programming Interface* – programmiliides ehk reeglistik olemasoleva valmisprogrammiga või teegiga suhtlemiseks.

et see projekti kaasata ja kolme kuu pärast ilmneb mingi kummaline viga, siis on väga oluline, et selle aja jooksul on suudetud selgeks saada, kuidas teegis asjad töötavad, et leida õige lahendus probleemile. Kui isegi aasta peale teegi kasutusele võtmist tundub selle sisene töö musta kastina, on väga raske iseseisvalt probleeme lahendada. Viimane omakorda, võib kogu projekti viia olukorda, kus sõltutakse kolmandast osapoolest nii palju, et kui teegi projekti omanikud ei ole aktiivsed ja leia piisavalt kiiresti lahendusi, võivad tähtajad nihkuda ning halvimal juhul kogu projekt välja surra.

Adekvaatse fookusega funktsionaalsus

Nagu iga alammoodul heas projektis, nii ka iga teek peaks tegema ainult ühte konkreetset asja ja seda korrektselt ning kõrvalmõjudeta. Seetõttu on väga oluline, et teek lahendaks ainult probleemi, milleks see loodi ja mitte erinevaid kõrvalprobleeme. Näiteks, kui tegemist on sõltuvusi süstiva teegiga (*dependency injection framework*), siis selle ülesandeks on hallata sõltuvusi, mitte serialiseerida JSON[3] andmeobjekte.

Lisaks peab teek rahuldama konkreetse projekti vajadusi. Analoogina võib vaadelda punktist A, punkti B liikumist – kui ühel inimesel on vaja läbida vaid mõned kilomeetrid, siis enamasti on kõige mõistlikum valik jalgratas, mitte veoauto; samas kui ülesandeks on transportida mõned tonnid metalli, on valik vastupidine. Seetõttu tulekski valida teek, mis rahuldaks konkreetse projekti vajadusi ning ei teeks väga palju üleliigset. Iga funktsionaalsus on mingis sõltuvuses mõne eelneva funktsionaalsusega, seega suure funktsionaalsuste pagasiga kaasneb enamasti ka suur keerukus. Lisaks võib muudatus ühes teegi funktsionaalsuses teha katki teise funktsionaalsuse jne. Kokkuvõttes tuleb tähelepanelik olla, et kõik vajalik oleks olemas ja kõik, mis on üleliigne, ei omaks väga suurt mõju ülejäänud projektile – vastasel juhul võib kogu projekt kannatada.

Õigsus

Lisaks sellele, et teek peab tegema vaid seda, milleks ta loodi, tuleb seda teha ka korrektselt. Väga hea indikaator on automaattestide olemasolu ja nende kvaliteet. Näiteks, kui suvalise meetodi sisu välja kommenteerida ja mõni test annab läbikukkuva tulemuse, siis on arvestatava tõenäosusega tegemist üpriski hästi testitud projektiga.

Lisaks on väga oluline testide arusaadavus ja see, kui lihtne on mõnel kolmandal osapoolel neid käivitada. Pole paremat viisi projekti valideerimiseks, kui jooksutada olemasolevaid teste või kirjutada mõni uus, et vaadata kas antud teek täidab õigesti parasjagu käsilolevas projektis nõutud tegevusi.

Kaasatus

Erinevalt tasulistest teekidest, ei maksta üldjuhul vabavaralise projekti autorile personaalse abi ega toetuse eest, mistõttu võib probleemi ilmnemisel lahenduse leidmine raskeks osutuda. Selliste projektide puhul muutub väga oluliseks kogukonna panus ja abi. Mida rohkem inimesi teeki kasutab, seda lihtsam on probleemidele lahendusi leida.

[3] *JavaScript Object Notation* – tekstvormingus ja programmeerimiskeelest sõltumatu lihtsustatud andmevahetusvorming, mis põhineb JavaScripti programmeerimiskeele alamhulgal.

Kaua eksisteerinud, kuid väikese või väheaktiivse kasutajaskonnaga projekti puhul, suure tõenäosusega ainult internetiotsinguga lahendusi ei leia. Kogukonna kaasatusest annab üpriski hea ülevaate näiteks Stackoverflow otsing teegi nimega – mida rohkem on küsimusi ja vastuseid, seda edukam projekt ja parem toetus.

Samas, väga noorte projektide puhul, kus pole veel tekkinud kaasatud kogukonda, on äärmiselt oluline autori aktiivsus. Viimase tuvastamiseks sobib vaadata avatud ja suletud probleemide sektsiooni projekti kodulehel. Kui seal on palju avatud probleeme, millel puudub autori tagasiside, siis ilmselt on tegemist, kas surnud projektiga või mitteaktiivse autoriga.

Uue SQLite ORM teegi loomise motivatsioon

Mobiilsetel seadmetel on internetikasutus üpris kulukas – esiteks peab lõppkasutaja selle eest maksma; teiseks raadio kiip, mis teostab internetiga suhtlust, on üks kõige enam akut tarbivaid riistvara komponente. Selleks, et iga kliki järgselt ei peaks internetist andmeid uuesti küsima, tuleks kasutada puhverdamist.

Üldjoontes on andmete puhverdamiseks kaks võimalust – salvestada andmed muutmällu või kettale. Muutmällu salvestamisel saab andmed kordades kiiremini kätte kui kettalt. Kuna serverites on mälu üpriski odav ja kättesaadav, siis puhverdamiseks kasutatakse muutmälu. Mobiilseadmetes on mälu väga piiratud ressurss, mistõttu hästi kirjutatud rakendused kasutavad puhverdamiseks, kas andmebaasi või tavalisi faile. Kuna andmebaasil on tugi andmete struktureeritud salvestamiseks ja pärimiseks, eelistatakse seda tavalistele failidele. Samas, et puhverdamine säilitaks oma eelised peab andmebaasiga suhtlus olema nii kiire kui ka optimeeritud efektiivsele mälu ja aku kasutusele.

Androidi operatsioonisüsteemis on esimese osapoole toetatud andmebaasiks SQLite. Sellega suhtlemiseks on Android SDKs madalataseme SQLite API ümbriskood (*wrapper code*). Sellest tulenevalt on tegu üpriski primitiivse mooduliga, mille kasutamiseks tuleb kirjutada palju korduvat koodi (*boilerplate code*). Teadagi, on tänu kordustele vead üpriski lihtsad tekkima ning andmebaasiga suhtluse haldamine võib vähegi suurema projekti juures muutuda vägagi aeganõudvaks.

Lahendust nimetatud probleemile pakuvad ORM teegid, mis kujutavad endast programmeerimiskeelepõhist abstraheeritud andmebaasiga suhtlemise liidest. Teisisõnu, võetakse programmeerimiskeele andmeobjektid ja nende andmeväljad ning seostatakse need üksühele vastavalt andmebaasi tabelite ja nende veergudega. Viimast tehakse automaatselt – kui muutub andmeobjekt, muutub ka andmebaasi tabel jne. Tekib n-ö „virtuaalne objektide andmebaas", millega saab suhelda programmeerimiskeele vahendusel, ilma et peaks otseselt kirjutama SQL päringuid.

Nagu eespool mainitud, on olemasolevaid Android ORM teeke väga palju. Siit tulenevalt tekib loogiline küsimus – miks luua veel üks? Lähemal lahenduste uurimisel selgub, et enam-vähem igaüks teeb midagi hästi, kuid puudub teek, mis teeks kõike hästi. Viimast näitab ka fakt, et Android ORM teekide seas puudub kogukonna standard. Seega on keskkond sobiv uue teegi tekkeks, mis lahendaks enamuse probleemidest ning teeks asju vähemalt sama hästi või paremini kui olemasolevad lahendused.

Uurimisküsimused

Kuna enamus olemasolevatest lahendustest kasutavad mingil viisil *reflectionit*[4] ning on teoreetiliselt teada, et see on aeglane[17], siis uuritakse antud töös, kas on võimalik realiseerida selline teek, milles *reflectioni* vajadus täielikult puudub või äärmisel juhul kasutab seda minimaalselt. Seega uuritakse, kas nimetatud optimisatsioon tagab parema jõudluse ja efektiivsema mälu kasutuse ning seega ka efektiivsema aku kasutuse, võrreldes *reflectionit* kasutavate alternatiividega. Lisaks uuritakse, mis optimeeringuid oleks võimalik veel teostada, et jõudluses konkureerida parimatega.

Enamustel olemasolevatel rakendustel puudub intuitiivne, paindlik ja lihtsasti kasutatav API andmebaasi operatsioonide mitmelõimeliseks käivitamiseks. Seega uuritakse, kuidas kõige optimaalsemal ja arusaadavamal viisil mitmelõimelist tööd (*concurrency*) korraldada. Lisaks uuritakse, kuidas realiseerida kõige arusaadavamal viisil andmebaasi tabeli andmete muudatustest teadaandvat süsteemi.

Olemasolevaid Android ORM teeke on üpriski palju ning massist eristumiseks tuleks teha midagi erilist. Muutumatutel objektidel (*immutable objects*) on palju häid omadusi [6, Item 15], mistõttu on need rakendustes palju kasutust leidnud. Seega uuritakse, kuidas toetada muutmatute objektide salvestamist ORM teegis. Lisaks, kuna andmemudelite vahel esineb tihti sõltuvusi, siis uuritakse, kuidas kõige sujuvamalt ja mugavamalt seda korraldada.

Tulenevalt hea avatud lähtekoodiga projekti tunnustest peab teegi API olema arusaadav, intuitiivne ja lihtsasti kasutatav. Seega uuritakse, kuidas nimetatud omadused tagada kõigi optimeeringute ja funktsionaalsuste juures. Lisaks, et tagada projekti edu kogukonna seas, peab teegi ülesseadmine ja kasutuselevõtmine olema võimalikult lihtne.

Viimaseks, uuritakse viise, kuidas teegi kasutajat suunata ja abistada selle kasutamisel. Viimase ülesandeks on vältida või ennetada rakenduse tööhetkel tekkivaid vigu – lõpptulemusena vähendada arendamiseks ja vigade otsimiseks kuluvat aega.

Töö sisu ülevaade

Töö esimeses peatükis kirjeldatakse ja analüüsitakse olemasolevaid Android SQLite ORM lahendusi. Teises peatükis tuuakse välja loodud teegi nõudmised ja kirjeldatakse teegis realiseeritud funktsionaalsusi ning nende disaini otsuseid ja põhjuseid. Järgnevalt kirjeldatakse teegi tehniliste lahenduste detaile ning põhjendatakse lahenduste tagamaid. Lisaks antakse ülevaade teegis rakendatud androidispetsiifilistest optimeeringutest. Neljandas peatükis tehakse põhjalik analüüs realiseeritud teegi jõudluse ja mälukasutuse võrdlusest põhikonkurentidega. Töö viimases peatükis kirjeldatakse teegi olukorda antud töö valmimise hetkel ning selle tulevikku ja edasiseid arengusuundi.

[4]Võimalus rakenduse tööajal programmikoodi uurida ning modifitseerida selle struktuuri ja käitumist.

1 Olemasolevad Android SQLite ORM lahendused

Olemasolevaid Android SQLite ORM teeke on üpriski palju ning kõikide kirjelduste ja analüüsi tulemuste väljatoomine poleks antud töö kontekstis mõistlik. Viimasest tulenevalt jaotati teegid realiseerimisviiside järgi gruppidesse ning valiti igast grupist kirjeldamiseks ja analüüsimiseks kõige silmapaistvam teek. Suures plaanis on gruppideks järgnevad: *reflectionil*, annotatsiooniprotsessoril ja erilahendusel põhinevad teegid. Järgnevalt on välja toodud kolm Androidile suunatud SQLite ORM teeki, millest autori arvates igaüks esindab oma lahendusgrupis parimat realisatsiooni. Iga teek esindab üht või mitut hästi realiseeritud aspekti, kuid millel on vähemal või suuremal määral puudujääke ülejäänud aspektides.

1.1 Sugar ORM

Sugar ORM teegi[28] suureks plussiks on selle lihtsus ja API kerge kasutatavus. Projekti kasutamiseks tuleb lisada teek build.gradle failis sõltuvuste hulka (vt Koodinäide 1) ja failis AndroidManifest.xml defineerida andmebaasi versioon ning nimi (vt Koodinäide 2).

```
1 dependencies {
2     compile "com.github.satyan:sugar:1.4"
3 }
```

Koodinäide 1: SugarORM sõltuvuse lisamine.

```
1 <application
2     android:label="@string/app_name"
3     android:icon="@drawable/icon"
4     android:name="com.orm.SugarApp">
5
6 <meta-data android:name="DATABASE" android:value="sugar_example.db" />
7 <meta-data android:name="VERSION" android:value="2" />
8
9 </application>
```

Koodinäide 2: SugarORM andmebaasi versiooni ja nime defineerimine.

Andmeobjektide andmebaasi lisamiseks tuleb teegi poolt etteantud baasobjekti laiendada (vt Koodinäide 3, rida 1), defineerida kõik andmeväljad minimaalselt paketi privaatse (*package private*) nähtavusega ja defineerida argumentideta konstruktor (vt Koodinäide 3, rida 6). Tänu nimetatud objekti laiendamisele tekivad otse andmeobjektile meetodid selle andmebaasi talletamiseks, uuendamiseks, kustutamiseks jne. Kõik see tagab arusaadava ja lihtsa teegi kasutuse.

Teisalt, piirab nimetatud muster andmeobjektide hierarhiat ja struktuuri. Lisades teegi olemasolevasse projekti ei pruugi sealne mudelite struktuur võimaldada uue objekti laiendamist. Kuna tavaliselt on andmeväljade nähtavus privaatne (*private*), tuleks lisaks kõikide andmeväljade nähtavusi muuta. Lisaks võib andmebaasi tekkida tahtmatuid tabeleid hooletu andmeobjekti laiendamise tulemusena. Kõik nimetatu võib potentsiaalselt tekitada rakenduses vigu ja lisatööd.

```java
public class Book extends SugarRecord<Book> {
    String title;
    String edition;
    Author author;

    public Book(){
    }

    public Book(String title, String edition){
        this.title = title;
        this.edition = edition;
    }
}
```

Koodinäide 3: Andmebaasi salvestatava andmeobjekti defineerimine SugarORM teegis.

Andmeväljadena toetatakse kõiki Java primitiivseid andmetüüpe. Lisaks on võimalik defineerida kompleksandmetüüpe (vt Koodinäide 3, rida 4) – kasutaja poolt defineeritud objekte, mis samuti laiendavad teegi baasobjekti. Teisalt ei võimalda teek ise defineerida tüübiteisendajaid. Viimast oleks vaja, kui tahta salvestada mõnda objekti, mille autoriks on kolmas osapool – näiteks objektid Locale, Date, mõni enum, jne.

Teegis on „puhast koodi" (*clean code*) ja loetavust säilitav ehitusmustriga (*builder pattern*) päringute API (vt Koodinäide 4). Tänu nimetatule asendatakse päringute tegemiseks vajalik „toores" SQL, keelepõhise API vastandiga. Kõik see väldib potentsiaalseid lahtise tekstina kirjutatavast SQList tulenevaid kirja-, reegli- ja tüübivigu.

```java
List<TestRecord> records = Select
    .from(TestRecord.class)
    .where(
        Condition.prop("test").eq("satya"),
        Condition.prop("prop").eq(2))
    .list();
```

Koodinäide 4: SugarORM ehitusmustriga päringu API.

Kuigi päringute API on üldjoontes üpriski selgesti arusaadav, saaksid osad aspektid veelgi lihtsamad olla. Näiteks võttes järgneva koodijupi (vt Koodinäide 5), siis on ilmselgelt meetodi esimese parameetrina sisestatav *Book.class* üleliigne, kuna meetodi välja kutsumiseks juba kasutatakse *Book* klassi.

```java
Book book = Book.findById(Book.class, 1);
```

Koodinäide 5: SugarORM kirje leidmine id järgi.

Kuigi antud teek on äärmiselt kergelt kasutatav ja lihtsasti mõistetava APIga, on sel väga suur puudus – see on üpriski aeglane (vt mõõtmistulemusi ptk 4.1). Nimelt kasutatakse sisemiselt väga ohtralt Java *reflectionit*, mis on teatavalt aeglasem, kui tavakood ning eriti aeglane Androidis. Operatsioonide kiirendamiseks hoitakse *reflectioniga* saadud objekte mälus, mis omakorda suurendab n-ö asjatut mälukasutust.

Suureks miinuseks on ka konkurentse töötluse täielik puudumine. Viimase lisamiseks tuleks näiteks ehitada oma ümbriskood, mida peaks potentsiaalselt uuendama peale igat teegi versiooni uuendust. Kõik see jällegi nõuab aega ning on aldis vigade tekkimisele.

1.2 greenDAO

GreenDAO teegi[11] eeliseks on kiirus – seda peetakse kõige kiiremaks Android ORM teegiks (vt mõõtmistulemusi ptk 4.1). Kahjuks kiiruse juures selle teegi positiivsed aspektid lõpevad – tegemist on suhteliselt keeruka, raskesti kasutatava, hallatava ning mõistetava projektiga.

Projekti sõltuvuse lisamine on tavapärane (vt Koodinäide 6). Edasine läheb keerukaks – lisaks tavalisele rakenduse moodulile tuleb luua Java moodul koodigeneraatori jaoks. Viimases tuleb luua klass, kus on tavaline *main* meetod, mille sisse tuleb kirjutada kood, mille tulemusena genereeritakse reaalsed Java objektid, mida saab juba rakenduses kasutada (vt Koodinäide 7). Koodi reaalseks genereerimiseks tuleb antud *main* meetod ise käivitada või see kuidagi ise oma rakenduse ehitusprotsessi integreerida.

```
1 dependencies {
2     compile "de.greenrobot:greendao:2.1.0"
3 }
```

Koodinäide 6: greenDAO sõltuvuse lisamine.

Kogu nimetatud protseduur on esiteks üleliigselt keeruline ning teiseks äärmiselt paindlikkust piirav. Olemasolevasse projekti antud teegi lisamisel tuleks, kas kõik andmebaasi mudelid ümber kirjutada või luua uued objektid generaatoriga ja kirjutada konverteerimist teostavad meetodid. Kõik see nõuab üpriski palju aega ning on aldis vigade tekkimisele.

```
1  package de.greenrobot.daogenerator.gentest;
2
3  import ...;
4
5  public class ExampleDaoGenerator {
6
7    public static void main(String[] args) throws Exception {
8      Schema schema = new Schema(1000, "de.greenrobot.daoexample");
9
10     final Entity author = addAuthor(schema);
11     final Entity book = addBook(schema);
12
13     book.addToOne(author, author.getPkProperty());
14
15     new DaoGenerator()
16         .generateAll(schema, "../DaoExample/src/main/java");
17   }
18
19   private static Entity addAuthor(Schema schema) {
20     Entity author = schema.addEntity("Author");
21     author.addIdProperty();
22     author.addStringProperty("name").notNull();
23     return author;
24   }
25
26   private static Entity addBook(Schema schema) {
27     Entity book = schema.addEntity("Book");
28     book.addIdProperty();
29     book.addStringProperty("title").notNull();
30     book.addStringProperty("edition");
31     return book;
32   }
33 }
```

Koodinäide 7: greenDAO mudelite generaator.

Kuigi teegi koduleht väidab, et nende API on ülimalt lihtne ja intuitiivne, siis reaalsuses on asi vastupidine – operatsioonide käivitamiseks tuleb palju „tseremooniat" läbida ning ohtralt on ebaselgust objektide olemuse või eesmärgi kohta. Koodinäide 8 näitab, mis sammud tuleb läbida, et saada reaalne objekt (rida 5), millega andmemudelite operatsioone teostada. Objektide *DaoMaster.DevOpenHelper*, *DaoMaster*, *DaoSession* olemus ja eesmärk jäävad esmapilgul selgusetuks. Teegi kasutamisel tekivad küsimused – kas *DaoMaster.DevOpenHelper* objekt tuleb initsialiseerida ühe korra või iga kord ja mis väärtused tuleb konstruktorisse ette anda; mis eesmärk on *DaoMaster* objektil; kas *DaoSession* objekt tuleb kuidagi sulgeda ning mida *session* üldse endast kujutab; kas sessiooni sees tehtud toimingud toimuvad transaktsiooni sees? Loetletud küsimustele vastuste saamiseks tuleb põhjalikult lugeda dokumentatsiooni või uurida genereeritud koodi. Lisaks, et tagada teegi korrektne kasutamine ning vähendada korduvat koodi (read 1-3), tuleks kirjutada ümbriskood, mis on just see, mille vajaduse ORM teegid peaksid kõrvaldama.

```
1 DaoMaster.DevOpenHelper helper = new DaoMaster.DevOpenHelper(context, "notes-db", null);
2 DaoMaster daoMaster = new DaoMaster(helper.getWritableDatabase());
3 DaoSession daoSession = daoMaster.newSession();
4
5 BookDao bookDao = daoSession.getBookDao();
```

Koodinäide 8: greenDAO ülesseadmine operatsioonide teostamiseks.

Positiivse aspektina on teegis päringute võimalus ehitusmustriga (vt Koodinäide 9).

```
1 List<Book> books = bookDao.queryBuilder()
2   .where(Properties.Title.eq("Game of Thrones"))
3   .orderAsc(Properties.Edition)
4   .list();
```

Koodinäide 9: greenDAO ehitusmustriga päringu API.

Konkurentse töötluse võimalus on antud töö kirjutamise hetkel beeta funktsionaalsus ning suuresti dokumenteerimata. Kõiki konkurentseid operatsioone töötleb järjestikku üks lõim ning kasutatakse tagasikutse (*callback*) mehhanismi. Kuna tegemist on beeta funktsionaalsusega, siis ilmselt võib paljugi muutuda, kuid antud kujul on see üpriski keeruline ja segane. Konkurentse töötluse operatsioonidele juurdepääsuks tuleb olemasolevast *DaoSession* objektist saada *AsyncSession* objekt (vt Koodinäide 10, rida 1). *AsyncSession* objektiga saab teostada kõiki konkurentseid operatsioone, kuid iga operatsiooni tulemusena tagastatakse *AsyncOperation* (vt Koodinäide 10, read 2 ja 4), mis töötab kui Java *Future* objekt – reaalse tulemuse saamiseks tuleb, kas tõkestada jooksvat lõime ja oodata tulemust kasutades meetodit *waitForCompletion()*, mis kokkuvõttes on sama hea, kui teha operatsioon kohe järjestikuselt; või defineerida operatsioonide „kuulaja" (vt Koodinäide 10, rida 6). Viimane defineeritakse kogu sessiooni peale, seega kõikide operatsioonide tulemused jõuavad tagasikutse meetodisse. Kirjeldatud mehhanism on üpriski segane ja aldis vigade tekkeks – esiteks on operatsiooni väljakutse ja selle reaktsioon teineteisest eraldatud, mistõttu pole koodile pealevaadates koheselt selge, mida tahetakse teha; teiseks peab tulemuste töötlemiseks neid sõeluma (vt Koodinäide 10, read 11 ja 13) ning kolmandaks tuleb operatsiooni tulemus õigesse tüüpi teisendada, kuna tagasikutsesse jõuab *Object* tüüp (vt Koodinäide 10, rida 9).

```java
final AsyncSession asyncSession = daoSession.startAsyncSession();
final AsyncOperation countOp = asyncSession.count(Book.class);
final int countSeqNr = countOp.getSequenceNumber();
final AsyncOperation deleteOp = asyncSession.delete(book);
final int deleteOpSequenceNumber = deleteOp.getSequenceNumber();
asyncSession.setListener(new AsyncOperationListener() {
    @Override
    public void onAsyncOperationCompleted(AsyncOperation operation) {
        final Object result = operation.getResult();
        final int seqNr = operation.getSequenceNumber();
        if (seqNr == countSeqNr) {
            // handle count
        } else if (seqNr == deleteOpSequenceNumber) {
            // handle delete
        }
    }
});
```

Koodinäide 10: greenDAO konkurentne töötlus.

Kirjeldatud näitest on näha, et konkurentse töötlemise API on antud teegis alles väga toores ning üleliigset „tseremooniat” nõudev, mistõttu muutub rakenduse kood keerulisemaks, raskemini arusaadavamaks ning vigadele vastuvõtlikumaks.

1.3 DBFlow

DBFlow[9] on olemasolevatest lahendustest kõige küpsem – üpriski kiire[5] ja lihtsa APIga. Teegi keskseks osaks on Java annotatsioonid, mille põhjal genereeritakse andmebaasiga suhtlust teostav Java kood.

Projekti sõltuvuse lisamine on tavapärasest veidi erinev. Esiteks tuleb lisada kolmanda osapoole Gradle plugin (vt Koodinäide 11, rida 7 ja 12), mis võimaldab annotatsiooni protsessorite sõltuvuse õige konfigureerimise ja teeb genereeritud koodi projektile nähtavaks. Teiseks tuleb lisada teegi kolm sõltuvust (vt Koodinäide 11, read 15 - 17) – annotatsiooni protsessor, põhimoodul, milles on annotatsioonid, konverterid jne; ning töömoodul, milles on kogu teegi funktsionaalsust teostav kood.

```groovy
// top level build.gradle
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        "com.neenbedankt.gradle.plugins:android-apt:1.4"
    }
}

// project level build.gradle
apply plugin: "com.neenbedankt.android-apt"

dependencies {
    apt "com.raizlabs.android:DBFlow-Compiler:2.2.1"
    compile "com.raizlabs.android:DBFlow-Core:2.2.1"
    compile "com.raizlabs.android:DBFlow:2.2.1"
}
```

Koodinäide 11: DBFlow sõltuvuse lisamine.

[5]DBFlow jääb enamasti kiiruselt siiski greenDAO-le alla, vt ptk 4.1.

Teegi kasutamiseks tuleb see initsialiseerida Androidi *Application* klassis (vt Koodinäide 12, rida 5). Lisaks tuleb defineerida andmebaas suvalises klassis *@Database* annotatsiooniga (rida 9).

```
1  public class ExampleApplication extends Application {
2      @Override
3      public void onCreate() {
4          super.onCreate();
5          FlowManager.init(this);
6      }
7  }
8
9  @Database(name = BookDatabase.NAME, version = BookDatabase.VERSION)
10 public class BookDatabase {
11
12     public static final String NAME = "Books";
13
14     public static final int VERSION = 1;
15 }
```

Koodinäide 12: DBFlow teegi initsialiseerimine.

Sarnaselt Sugar ORM-le tuleb andmeobjektide andmebaasi lisamiseks ainult teegi poolt etteantud objekti laiendada (vt Koodinäide 13, rida 2), mille tulemusena päritakse meetodid andmebaasiga suhtluseks. Kõik sellega seonduvad peatükis 1.1 nimetatud positiivsed ja negatiivsed aspektid kehtivad ka antud teegi puhul. Lisaks on võimalik implementeerida ka ainult *Model* objekti, kuid sel juhul tuleb andmebaasiga suhtluseks vajaminevad meetodid ise defineerida või kopeerida *BaseModel* klassist. Erinevalt SugarORM-le tuleb lisaks defineerida ka antud andmekirjet kirjeldavad annotatsioonid. Minimaalselt tuleb lisada *@Table* annotatsioon (vt Koodinäide 13, rida 1) defineerides andmebaasi, kuhu antud tabel kuulub ja `allFields = true` , et kõik tabeli väljad kuuluksid salvestamisele; ning *@PrimaryKey*, *@Column* annotatsiooni paari (read 3 - 4), mis defineerib antud tabeli primaarse võtme. Lisaks peab objektil olema vähemalt *package private* nähtavusega konstruktor ja väljad, kuid viimased võivad ka olla *private* nähtavusega, kui neile on defineeritud juurdepääsu meetodid (*access methods*).

```
1  @Table(databaseName = BookDatabase.NAME, allFields = true)
2  public class Book extends BaseModel {
3      @Column
4      @PrimaryKey(autoincrement = true)
5      long id;
6      String title;
7      String edition;
8
9      @ForeignKey(references = {
10         @ForeignKeyReference(
11                     columnName = "author_id",
12                     columnType = Long.class,
13                     foreignColumnName = "id")},
14         saveForeignKeyModel = false)
15     @Column
16     Author author;
17
18     Book(){
19     }
20 }
```

Koodinäide 13: Andmebaasi salvestatava andmeobjekti defineerimine DBFlow teegis.

Andmeväljadena toetatakse kõiki Java primitiivseid andmetüüpe. Lisaks on võimalik defineerida kompleksandmetüüpe (vt Koodinäide 13, read 9 - 16). Nagu koodinäitest võib välja lugeda, siis kasutaja defineeritud andmetüüpide sidumine on üpriski keeruline ja tekstimahukas. Lisaks, kui näiteks peaks viidatava tabeli rea nimi muutuma (vt Koodinäide 13, rida 11), siis *Book* objektil olev viga ilmneb alles rakenduse töö ajal. Kogu ridades 9 - 13 defineeritud info oleks võimalik välja lugeda mudelite struktuurist juba kompileerimise ajal.

Kõiki mitte-primitiivseid Java objekte saab andmebaasi salvestada ka tüübiteisendajate vahendusel, mis muudavad mitte-primitiivseid objektid primitiivseteks (vt Koodinäide 14). Olles defineerinud tüübiteisendaja mingi mitte-primitiivse tüübi jaoks saab seda kasutada samal viisil tabeli väljana, kui primitiivset objekti (vt Koodinäide 14, rida 29).

```java
1  // type converter
2  @com.raizlabs.android.dbflow.annotation.TypeConverter
3  public class LocationConverter extends TypeConverter<String,Location> {
4
5      @Override
6      public String getDBValue(Location model) {
7          return model == null ? null : String.valueOf(
8                  model.getLatitude()) + "," + model.getLongitude();
9      }
10
11     @Override
12     public Location getModelValue(String data) {
13         String[] values = data.split(",");
14         if(values.length < 2) {
15             return null;
16         } else {
17             Location location = new Location("");
18             location.setLatitude(Double.parseDouble(values[0]));
19             location.setLongitude(Double.parseDouble(values[1]));
20             return location;
21         }
22     }
23 }
24
25 // usage
26 @Table(...)
27 public class SomeTable extends BaseModel {
28     @Column
29     Location location;
30 }
```

Koodinäide 14: Tüübiteisendaja defineerimine DBFlow teegis.

Positiivse aspektina on teegis päringute võimalus ehitusmustriga (vt Koodinäide 15).

```java
1  List<Book> books = new Select()
2      .from(Book.class)
3      .where(Condition.column(Book_Table.TITLE).eq("Game of Thrones"))
4      .queryList();
```

Koodinäide 15: DBFLow ehitusmustriga päringu API.

Teegis on ka tugi operatsioonide paralleelseks käivitamiseks. Lisamise, kustutamise ja uuendamise paralleelseks käivitamiseks tuleb meetodi parameetrina ette anda *true* väärtus (vt Koodinäide 16, rida 2). Päringute paralleelseks käivitamiseks (vt Koodinäide 16) tuleb *TransactionManager* objektile (rida 9) lisada uus transaktsioon, mille parameetriteks on teostatav päring ja tagasikutse klass (rida 11). Kõik konkurentsed operatsioonid

teostatakse ühes taustalõimes. Antud API on üpriski lihtne, kuid pole kuigi intuitiivne. Lisaks pole koheselt arusaadav, mis lõimes tagasikutse meetod jookseb – Androidis võib kasutajaliidese objekte muuta ainult pealõimes (*main thread*), seega, kui päringu tulemust tahetakse kasutajale näidata, peaks ise leidma viisi kuidas tulemus sinna lõime viia.

```
1  final Book book = ...;
2  book.save(true);
3
4  final ModelQueriable<Book> query = new Select()
5          .from(Book.class)
6          .where(Condition
7              .column(Book_Table.TITLE)
8              .eq("Game of Thrones"))
9  TransactionManager.getInstance()
10         .addTransaction(new SelectListTransaction<>(query,
11                 new TransactionListenerAdapter<List<Book>>() {
12                     @Override
13                     public void onResultReceived(List<Book> books) {
14                         // retrieved here
15                     }
16                 }
17         ));
```

Koodinäide 16: DBFLow konkurentne töötlus.

Kuid kõige hea juures on ka miinuseid – genereeritud klasside juurdepääsuks kasutatakse *reflectionit*, mis on aeglane. Viimase kiirendamiseks puhverdatakse *reflectioniga* saadud objekte, mis kulutab aga niigi vähest mälu. Lisaks kasutatakse üpriski palju mälu, et erinevate sisemiste loogikate üle „raamatupidamist” korraldada (vt ptk 4.2).

2 Realiseeritud teegi kirjeldus ja analüüs

Lähtudes olemasolevate teekide hästi ja halvasti realiseeritud aspektidest ning Androidi kogukonna trendidest seati loodava teegi realiseerimisel aluseks järgnevad nõuded.

Funktsionaalsed nõuded:

1. Andmebaasi struktuur defineeritakse läbi Java andmemudelite.
2. Võimalus muuta andmemudeli salvestamiseks kasutatavaid omadusi.
3. Võimalus defineerida andmebaasi kuuluvates andmemudelites väljadena Java primitiivseid andmetüüpe ning nende pakendatud (*boxed*) vasteid, kasutaja defineeritud muutuvaid andmetüüpe, kasutaja defineeritud muutumatuid andmetüüpe ja kolmanda osapoole andmetüüpe.
4. Võimalus defineerida tüübiteisendajaid.
5. Võimalus teostada andmebaasi CRUD[6] operatsioone.
6. Võimalus teostada andmebaasi CRUD operatsioone asünkroonselt.
7. Võimalus teostada transaktsioonis CRUD operatsioone itereeritaval objektide kogumil.
8. Võimalus pärida andmeid nende täielikus sügavuses. Teisisõnu, kui andmeobjekt sisaldab väljadena teisi salvestavaid andmeobjekte ja need omakorda järgmisi jne, siis esialgse andmemudeli täielikul päringul peaks tagastatama kõik objektid, kuni võimaliku suurima sügavuseni.
9. Võimalus salvestada andmeid nende täielikus sügavuses.
10. Võimalus andmete pärimisel täpsustada tagastusviisi – kõikide päringule vastavate andmete kogu, esimene päringule vastav andmeobjekt, päringu tulemusele vastav kirjete arv või päringule vastavad andmed parsimata kujul Android *Cursor* objektis.
11. Võimalus luua aktiivseid päringuid, kus teavitatakse väljakutsujat, kui päringule vastavates tabelites toimuvad muudatused.
12. Võimalus teostada andmebaasi migratsioone.
13. Võimalus defineerida erinevatele ehitusvariantidele ja -maitsetele erineva struktuuri, nime ja versiooniga andmebaase.

Mittefunktsionaalsed nõuded:

1. Teek peab olema avatud lähtekoodiga ja avalikult kättesaadav.
2. Teek peab toetama Android Studio IDEs[7] arendamist.
3. Teek peab toetama Gradle ehitussüsteemi.
4. Teegis arendatav funktsionaalsus peab olema fokuseeritud ainult andmebaasi operatsioonidele.

[6] *Create, Read, Update, Delete* – andmete loomine, pärimine, uuendamine ja kustutamine.
[7] *Integrated Development Environment* – tarkvararakendus, mis pakub laiaulatuslikke vahendeid tarkvara arendamiseks.

5. Teek ei tohi kasutada *reflectionit*.
6. Teegi kasutuselevõtmine olemasolevasse rakendusse peab olema võimalikult lihtne ning sellega ei tohi kaasneda andmestruktuuride hierarhia muudatusi.
7. Teegi sõltuvuse lisamine rakenduse projekti peab olema võimalikult lihtne.
8. Kui andmete või toimingudetailide tuletamine on võimalik, peab teek võimaldama viisi selle kasutaja eest automaatselt ära teha.
9. Teek peab leidma tüübiteisendajaid ja andmebaasi kuuluvaid mudeleid automaatselt, ilma et kasutaja peaks eraldi midagi selle jaoks lisaks seadistama.
10. Andmebaasi kuuluva mudeli defineerimine peab olema võimalik minimaalselt ühe koodireaga.
11. Avalikus API-s peab andmebaasioperatsioonide teostamine toimuma ilma kõrvaliste sõltuvusobjektideta[8].
12. Avalik API peab järgima puhta koodi soovitusi [6, 14, 21].
13. Avalik API peab olema võimalikult arusaadav, intuitiivne ja lihtsasti kasutatav.
14. Teegi kasutamine peab olema nii märkamatu, et selle poolt pakutud funktsionaalsused tunduvad, kui Java keele osad.
15. Teek peab kasutajat kõikvõimalikel viisidel selle kasutamise juures abistama ning suunama.
16. Teegi jõudlus peab olema vähemalt samaväärne jõudluselt parimate konkurentidega.
17. Teegi mälukasutus peab olema vähemalt sama optimaalne, kui mälukasutuselt kõige efektiivsematel konkurentidel.

Antud töö tulemusena valmis kirjeldatud nõuetele vastav SQLite ORM teek Androidile nimega SqliteMagic. Töö kirjutamisel on eeldatud, et kasutatakse Android Studio IDE ja Gradle ehitussüsteemi kombinatsiooni, mis alates 2014 aasta lõpust on Androidi ametlik arenduskeskkond [1].

2.1 Teegi paigaldus

Tagamaks projekti edu kogukonna seas, peab teegi ülesseadmine ja kasutuselevõtmine rakendusse olema võimalikult lihtne. Tulenevalt antud projekti teostusdetailidest (vt ptk 3) ja modulaarsetest osadest tuleks kasutajal paigaldada järgnevad sõltuvused: IntelliJ plugin, annotatsioonide moodul, annotatsiooniprotsessori moodul, rakenduse tööaja moodul ja kolmanda osapoole sõltuvused[9]. Kõik kirjeldatu on ilmselgelt liiga palju, et tagada ülesseadmise lihtsus ja kiirus. Tulenevalt kirjeldatud probleemist ja teegi teostusdetailidest

[8] Kõrvaliste sõltuvusobjektide all mõeldakse objekte, mis pole seotud operatsiooni detailide defineerimisega, kuid oleks operatsiooni teostamiseks hädavajalikud ning mille initsialiseerimine ja haldamine peaks olema kasutaja ülesanne. Näiteks parameetrina etteantav andmebaasi ühenduse objekt või andmebaasioperatsioone koondav DAO objekt, jne.

[9] Võimaldavad baitkoodi ja AST muudatused ning tagavad annotatsiooniprotsessorite õige töö Androidi projektis.

osutus kõige paremaks lahenduseks kirjutada Gradle plugin. Esiteks, annab see võimaluse kogu keerulise seadistuse kasutaja eest ära teha. Viimase tulemusena saab kogu teegi sõltuvuse vaid kahe koodireaga projekti lisada (vt Koodinäide 17, read 8 ja 20). Teiseks, pakub võimaluse modifitseerida rakenduse ehitustsüklit vastavalt vajadusele, mida ka antud projekti puhul on vaja, et realiseerida püstitatud eesmärkidest tulenevaid lahenduste teostusdetaile (vt ptk 3.2.1).

Andmebaasi nime ja versiooni defineerimine käib läbi Android Gradle plugina ehitusvariandi konfigureerimise süsteemi [10]. Versiooni jaoks on vajalik *int* tüüpi, *DB_VERSION* nimega *buildConfigField* väli ja nime jaoks *String* tüüpi *DB_NAME* nimega väli, millest ainult versiooni defineerimine on kohustuslik (vt Koodinäide 17, read 25 ja 27). Andmebaasi nime vaikimisi väärtus on *database.db*. Nimetatud süsteemi abil saab konfigureerida erinevatele ehitusvariantidele ja -maitsetele erinevaid andmebaasi nimesid ja versioone. Näiteks, kui rakendusel on „demo" ja „täisversiooni" ehitusvariandid ning need kasutavad erinevaid andmeid ja omavad erinevaid väljalaske perioode, siis oleks vajalik, et neile vastavad andmebaasid on nimetatud näiteks *demo.db* ja *full.db* ning omavad versioone 44 ja 5. Ehitusvariandi konfigureerimise süsteemi kasutades on kõik nimetatu võimalik lisapingutusi tegemata, mistõttu valitigi antud süsteem koodis või manifestis defineerimise asemel.

```
1  // top level build.gradle
2  buildscript {
3      repositories {
4          jcenter()
5      }
6      dependencies {
7          classpath 'com.android.tools.build:gradle:1.3.0'
8          classpath 'com.siimkinks.sqlitemagic:sqlitemagic-plugin:0.3.1'
9      }
10 }
11
12 allprojects {
13     repositories {
14         jcenter()
15     }
16 }
17
18 // project level build.gradle
19 apply plugin: 'com.android.application'
20 apply plugin: 'com.siimkinks.sqlitemagic'
21
22 android {
23     defaultConfig {
24         ...
25         buildConfigField "int", "DB_VERSION", "1"
26         // Optional (defaults to "database.db")
27         buildConfigField "String", "DB_NAME", "\"sample.db\""
28     }
29 }
30 dependencies {
31     compile 'io.reactivex:rxjava:1.0.16'
32 }
```

Koodinäide 17: SqliteMagic sõltuvuse lisamine.

Teegi tööks on vajalik Androidi rakenduse kontekst, mille vahendusel luuakse ühendus andmebaasiga. Seetõttu tuleb enne teegi kasutamist see initsialiseerida meetodiga *init*, mille parameetriks on *Application* objekt (vt Koodinäide 18, rida 5). Üheks väga heaks kohaks, kus teek initsialiseerida, on näiteks *Application* objekti *onCreate* meetod, kuid antud asukoht pole kohustuslik.

```java
1 public final class MyApplication extends Application {
2     @Override
3     public void onCreate() {
4         super.onCreate();
5         SqliteMagic.init(this);
6     }
7 }
```

Koodinäide 18: SqliteMagic teegi initsialiseerimine.

Kokkuvõttes tuleb teegi töölesaamiseks paigaldada IntelliJ plugin (vt ptk 3.2.3), lisada kaks rida sõltuvuse defineerimiseks, ühe rea andmebaasi versiooni defineerimiseks ning ühe rea teegi töö initsialiseerimiseks.

2.2 Andmebaasi defineerimine

Üldjoontes on olemasolevastes ORM laendustes kasutusel kolm andmebaasi defineerimise viisi. Esiteks, iserealiseeritud erilahendus, kus kirjeldatakse kogu andmebaas otse Java koodis, mis genereerib andmebaasiga suhtluseks vajaliku koodi ettenäidatud kausta (vt ptk 1.2). Antud lahenduse plussiks on asjaolu, et kuna kogu andmebaasi suhtlust teostav kood genereeritakse arendamise käigus, siis pole vajadust kasutada *reflectionit* rakenduse töötamise ajal. Lisaks saab genereeritavat koodi optimeerida ning tulevastes teegi versioonides vastavalt vajadusele muuta. Samas, tulenevalt iserealiseeritusest puudub vaikimisi integratsioon rakenduse ehitusprotsessis – koodi genereerimine tuleb käsitsi käivitada. Nimetatud probleemi lahendaks Gradle plugin, mis integreeriks koodi genereerimise rakenduse ehitustsükliga. Samas antud lahendus poleks ideaalne, kuna puuduks täielik integratsioon Java annotatsioonide töötlemise süsteemiga ja kolmandate osapoolte lahendustega, mistõttu poleks nende toetamine võimalik. Lisaks on iserealiseeritud erilahenduse kõige suuremaks miinuseks andmebaasi defineerimise viis – andmeobjekte luuakse kolmanda osapoole API-ga, mis ei pruugi esmasel vaatlusel tunduda intuitiivne ja arusaadav ning see asub põhiprojektist eraldi. Lisaks ei pruugi andmeobjetide defineerimise API toetada kõiki rakenduse jaoks vajalikke funktsionaalsusi, mistõttu tuleb valida, kas oodata järgmist teegi versiooni, kus on vajalik funktsionaalsus lisatud, või leiutada kõrvalisi „häkke” probleemi lahendamiseks. Kõik nimetatu halvendab arusaadavust, lisab keerukust ja arendamiseks kuluvat aega, mistõttu pole antud lahendus sobiv käesoleva töö tulemusena valmivas teegis.

Teiseks võimaluseks on defineerida andmebaas välises failis või failide kogumis, kas puhtal SQL kujul tabelite loomislausetega või mõnes struktureeritud andmeid kirjeldavas keeles nagu *JSON* või *Protocol Buffers*. Nimetatud lahendus on sarnane esimesele – failis defineeritu põhjal genereeritakse määratud kausta andmebaasiga suhtluseks vajalik kood. Erinevalt esimesest lahendusest, annab see, aga rohkem võimalusi erinevateks realisatsioonideks. Näiteks saaks failide põhjal genereerida lahendusi teistesse andmebaasidesse erinevates keskkondades nagu server või veebibrauser ja seda kõike programmeerimiskeeltest sõltumatult. Lisaks, kui kasutada näiteks *Protocol Buffersit*, saaks serverist tulevaid bitijadasid Java koodi deserialiseerimata otse andmebaasi salvestada. Kuna antud lahenduse plussid ja miinused on sarnased esimesele lahendusele, siis pole ka see lahendus sobiv käesoleva töö tulemusena valmivas teegis.

Kolmandaks lahenduseks, mida realiseerib ka valminud teek, on kasutada annotatsioonide

töötlust, kus andmebaasi kuuluvaid Java objekte märgistatakse vajadusel metaandmeid sisaldavate annotatsioonidega. Tulenevalt faktist, et andmebaasi metaandmed on defineeritud otse Java objektil, tagab antud lahendus parima loetavuse, arusaadavuse ja lihtsuse ning sellest tulenevalt ka kiireima arendusaja. Vastavalt realisatsiooni tüübile, kasutatakse annotatsioone, kas rakenduse tööhetkel või genereeritakse nende põhjal andmebaasiga suhtluseks vajalik kood kompileerimisel. Kuna valminud teegi üheks põhieesmärgiks on täielik *reflectioni* kõrvaldamine, siis annotatsioonide käsitlemiseks sobib ainult kompileerimisaegne töötlemine (vt ptk 3.1). Viimase suureks plussiks on täielik integratsioon rakenduse ehitusprotsessiga[10]. Antud lahenduse ainsaks miinuseks on fakt, et annotatsiooniprotsessoriga pole võimalik olemasolevat koodi muuta – seetõttu peaks kasutaja genereeritud koodi ise välja kutsuma. Kirjeldatud miinus on aga antud teegis kõrvaldatud (vt ptk 3.2.2).

Andmebaasi defineerimiseks on teegis järgnevad annotatsioonid (vt Koodinäide 19):

- *@Table* – defineerib tabeli andmebaasis. Vaikimisi võetakse tabeli nimeks väikeste tähtedega klassi nimi, kuid seda on võimalik muuta parameetri *value* abil. Annotatsiooni parameetri *persistAll* väärtusega *true* saab teegile öelda, et salvestamisele kuuluvad kõik objekti väljad, mis pole annoteeritud *@IgnoreColumn* annotatsiooniga. Vastasel korral kuuluvad salvestamisele ainult *@Column* annotatsiooniga väljad, mis on ka vaikimisi käitumine. Vaikimisi kasutab teek objekti väärtuste juurdepääsuks otse väljamuutujaid (vt ptk 3.3.1). Viimast käitumist on võimalik muuta kõikide väljade jaoks parameetriga *useAccessMethods*, kus väärtuse *true* korral, kasutatakse väärtuste juurdepääsuks *JavaBeans* stiilis meetodeid [15].
- *@Id* – defineerib tabeli primaarvõtme. Vaikimisi on kõik primaarvõtmed seadistatud automaatse kasvamise peale, mida saab muuta parameetriga *autoIncrement*. Igal tabelil peab olema üks *@Id* annotatsiooniga andmeväli, kuid selle defineerimata jätmisel genereeritakse „maagiline" *long* tüüpi andmeväli nimega „id"[11] (vt ptk 3.2.2).
- *@Column* – defineerib veeru tabelis. Vaikimisi võetakse veeru nimeks väikeste tähtedega andmevälja nimi, kus *camelcase*[12] on asendatud alakriipsudega, kuid seda on võimalik muuta parameetri *value* abil. Annotatsiooni parameetriga *handleRecursively*, mille vaikimisi väärtus on *true*, defineeritakse, kas mitteprimitiivset andmevälja käsitletakse täies mahus või osaliselt (vt ptk 2.2.2). Lisaks on võimalik mitteprimitiivsetele andmeväljadele lisada SQL välisvõtme kitsendus *ON DELETE CASCADE* parameetriga *onDeleteCascade*. Sarnaselt *@Table* annotatsiooniga, on võimalik konkreetse andmevälja piires määrata andmevälja väärtuse juurdepääsu tüüp parameetriga *useAccessMethods*.
- *@IgnoreColumn* – defineerib salvestamisele mittekuuluva andmevälja. Antud annotatsioonil on mõju ainult siis, kui *@Table* parameetri *persistAll* väärtus on *true*.

Tähtis on ka ära mainida järgnevad nõudmised ja lõdvendused andmemudelitele:

- Erinevalt konkurentide nõudmistele andmeobjektide kohta, ei pea antud teegis kasutatavad andmeklassid ühtegi baasklassi laiendama ega implementeerima.

[10]Alates Java 6-st, mida kasutatakse ka Androidis, on keeles tugi annotatsiooniprotsessoritele [19].
[11]Ainsaks erinevuseks on muutumatud objektid, kus kirjeldatud käitumist ei teostata (vt ptk 2.2.4).
[12]Kirjutusstiil, kus ühendsõnad või fraasid kirjutatakse kokku viisil, kus iga järgnev sõna algab suure tähega. Näiteks *ListItem*.

- Ühegi annotatsiooni ükski parameeter pole kohustuslik – kõik parameetrid on vaikimisi seadistatud enamkasutatud omadustele.
- Igal andmeklassil peab olema minimaalselt *package private* nähtavusega parameetriteta konstruktor, mida kasutatakse andmete pärimisel uue tagastatava väärtuse loomisel (vt Koodinäide 19, rida 13).
- Kui annotatsiooni *@Table* või *@Column* parameetri *useAccessMethods* väärtus on *true*, siis iga salvestamisele kuuluva välja kohta peab olema kaks meetodit – väärtuse lugemiseks meetod stiilis *getXxx()* või primitiivse *boolean* väärtuse korral *isXxx()* ning väärtuse kirjutamiseks meetod stiilis *setXxx()* (vt Koodinäide 19). Parameetri väärtuse *false* korral peab iga salvestamisele kuuluva välja nähtavus olema minimaalselt *package private*.
- Eksperimentaalse funktsionaalsusena toetatakse ka *private* nähtavusega konstruktoreid ja salvestatavaid välju, kus AST[13] manipulatsiooniga muudetakse vastavaid nähtavusi (vt ptk 3.2.2). Samas, kuna antud funktsionaalsuse vajadus ja turvalisus on küsitava väärtusega, siis vajab selle edendamine põhifunktsionaalsuseks põhjalikumat uurimist ja kogukonna tagasisidet.
- Kui annotatsiooni *@Table* parameetri *persistAll* väärtus on *true*, siis primaarvõtme defineerimiseks pole *@Id*-le lisaks vaja *@Column* annotatsiooni defineerida (vt Koodinäide 19, rida 4). Lisaks, iga välja vaikimisi salvestamisstrateegia on võrdne *@Column* annotatsiooni vaikimisi väärtustest tuleneva salvestusstrateegiaga. Viimase muutmiseks tuleb igal soovitud väljal defineerida *@Column* annotatsioon soovitud käitumisega (vt Koodinäide 19, read 8 ja 10).

```
1  @Table(persistAll = true)
2  public final class Author {
3      @Id
4      long id;
5      String firstName;
6      @IgnoreColumn
7      String lastName;
8      @Column(value = "phone")
9      String phoneNumber;
10     @Column(useAccessMethods = true)
11     private int age;
12
13     Author() {}
14
15     public int getAge() {
16         return age;
17     }
18
19     public void setAge(int age) {
20         this.age = age;
21     }
22 }
```

Koodinäide 19: SqliteMagic teegis andmebaasi kuuluva andmeobjekti defineerimine.

2.2.1 Java andmetüübid

Salvestamiseks on toetatud kõiki Java primitiivseid andmetüübid ja nende objektiks pakendatud (*boxed*) vasted, sealhulgas ka baitide massiiv ja *String*. Kõik Java andmetüübid

[13] *Abstract Syntax Tree* – süntaksipuu.

teisendatakse SQLite poolt toetatud andmetüüpidesse:

- NULL – kõik objektid, mis pole väärtustatud
- INTEGER – long, Long, int, Integer, short, Short, boolean ja Boolean
- REAL – double, Double, float ja Float
- TEXT – String
- BLOB – byte, Byte, byte[] ja Byte[]

2.2.2 Kasutaja defineeritud andmetüübid

Mobiilsete rakenduste andmebaasides, nagu ka tavapärastes relatsioonilistes andmebaasides, on väga tavaline mitme tabeli omavahelised seosed. Kuna tavapäraste süsteemi poolt pakutud meetoditega on relatsioonide loomine ja haldamine tülikas, siis antud probleem on sobiv ORM teegile lahendamiseks. Lisaks, kuna olemasolevastes lahendustes (vt ptk 1) antud funktsionaalsus, kas puudub täielikult või on seda üpriski tülikas kasutada, siis selle toetamine ja lihtsustamine annab tugeva eelise konkurentide ees.

Kompleksseid ehk kasutaja defineeritud andmetüüpide andmevälju käsitletakse teegis, kui tavalisi Java andmetüüpe (vt Koodinäide 20, rida 4 ja 6). Ainuke nõudmine on, et nii juurandmeklass (vt Koodinäide 20, rida 1) kui ka kompleksvälja andmeklass (vt Koodinäide 20, rida 9) peavad olema annoteeritud *@Table* annotatsiooniga. Muid lisasamme tegema ei pea, kuna kõik relatsiooni teostamiseks vajalikud metaandmed tuletatakse andmete struktuurist.

Sisemiselt salvestatakse juurtabelisse viidatud tabeli kirje identifikaator. Vaikimisi salvestatakse juurobjekti salvestamisel ka kompleksväljad nende vastavatesse tabelitesse. Nimetatud funktsionaalsus on muudetav annotatsiooni *@Column* parameetriga *handleRecursively*, mille *false* väärtuse korral salvestatakse ainult viidatava tabeli kirje identifikaator. Lisaks on võimalus lisada tabelitevahelisele seosele SQL välisvõtme piirang *ON DELETE CASCADE* annotatsiooni *@Column onDeleteCascade* parameetriga. Vaikimisi on viimase väärtus *false*.

```
1  @Table(persistAll = true)
2  public final class Book {
3      @NonNull
4      String title;
5      @NonNull
6      Author author;
7  }
8
9  @Table(persistAll = true)
10 public final class Author {
11     @NonNull
12     String firstName;
13     @NonNull
14     String lastName;
15 }
```

Koodinäide 20: SqliteMagic teegis kasutaja defineeritud andmeobjekti kasutamine andmeväljana.

2.2.3 Tüübiteisendajad

Tihti on vaja salvestada andmeobjekte, mis pole programmi valduses – pole primitiivsed Java objektid ega kasutaja enda defineeritud objektid. Nendeks võivad olla näiteks Java Date, Instant, ZoneOffset jne objektid või mõne muu kolmanda osapoole andmeobjektid, mille signatuuri muutmine pole võimalik.

Nimetatud probleemile pakuvad lahendust tüübiteisendajad, mis konverteerivad SQLite jaoks tundmatu andmetüübi mõneks toetatud andmetüübiks. SqliteMagic teegis on nende defineerimine äärmiselt lihtsaks tehtud – tuleb ainult klassile, mis teostab konverteerimist, lisada *@ObjectTransformer* annotatsioon (vt Koodinäide 21, rida 1) ning defineerida kaks staatilist meetodit, mille nimed pole olulised, kuid millel on *@ObjectToDbValue* ja *@DbValueToObject* annotatsioonid (vt Koodinäide 21, rida 3 ja 8). Viimastes meetodites tuleb vastavalt teostada objekti konverteerimine andmebaasi väärtuseks ja vastupidi. Järgnevalt leiab kompileerimise käigus annotatsiooniprotsessor (vt ptk 3.1) kõik tüübiteisendajad *@ObjectTransformer* annotatsioonide järgi üles ning kasutab neis defineeritud staatilisi meetodeid vajalikes kohtades andmete konverteerimisel.

Tähtis on siinkohal märkida, et iga *@ObjectTransformer* annotatsiooniga klass peab sisaldama ainult ühte teisendust teostavat meetodite paari ning iga andmetüübi kohta saab olla ainult üks tüübiteisendaja. Nimetatud reegli rikkumisel loetakse rakenduse ehitus ebaõnnestunuks ning teavitatakse kasutajat informatiivse sõnumiga. Lisaks teostatakse rida erinevaid kontrolle, mis valideerivad tüübiteisendaja korrektsust.

Nagu eelnevalt mainitud, peavad tüübiteisendust teostavad meetodid olema staatilised, mis annab kaks optimisatsiooni. Esiteks, tüübiteisendajate objekte pole vaja initsialiseerida, mistõttu väheneb mälukasutus, kuid sellest tulenevalt on neid ka lihtsam kasutada. Teiseks, on Androidis staatiliste meetodite käivitamine kiirem, kui virtuaalsete meetodite (vt ptk 3.3.1). Kuigi enamustes SQLite ORM teekides on tüübiteisendajad toetatud, siis antud töö autorile teadaolevalt, ei kasuta ükski neist viimati mainitud optimeeringut.

Vähendamaks teegikasutaja tööd on realiseeritud mitmed tüübiteisendajad enamkasutatud kolmanda osapoole andmetüüpide jaoks:

- boolean ja Boolean (kuna SQLite ei toeta boolean väärtusi, siis kõige mõistlikum oli realiseerida selle tugi tüübiteisendajaga)
- java.util.Date (vt Koodinäide 21)
- java.sql.Date
- java.util.Calendar

```
1  @ObjectTransformer
2  public final class DateTransformer {
3      @ObjectToDbValue
4      public static Long objectToDbValue(Date javaObject) {
5          return javaObject == null ? null : javaObject.getTime();
6      }
7
8      @DbValueToObject
9      public static Date dbValueToObject(Long dbObject) {
10         return dbObject == null ? null : new Date(dbObject);
11     }
12 }
```

Koodinäide 21: SqliteMagic teegi andmetüübi teisendaja *java.util.Date* näitel.

2.2.4 Muutumatud objektid

Muutumatu klass (*immutable class*) on tavaline klass, mille initsialiseeritud objekti väärtusi ei saa muuta. Java platvormis on mitmeid klasse, mis on oma olemuselt muutumatud – *String*, primitiivsete andmetüüpide objektidest vastandid, *BigInteger*, jne. Muutumatuid klasse saab ka ise defineerida – klassil ei tohi olla ühtegi meetodit, mis muudaks objekti olekut; klassi ei tohi saada laiendada; kõik klassisisesed väljad peaksid olema muutumatud (*final*) ja privaatse nähtavusega. Muutumatutel klassidel on palju häid omadusi – neid on lihtsam disainida, implementeerida ja kasutada, kui muutuvaid klasse. Lisaks on muutumatud objektid vähem aldid vigadele, turvalisemad ja olemuselt mitmelõimelise töö suhtes ohutud – nad ei vaja sünkroniseerimist ning neid saab jagada lõimede vahel kõrvalmõjudeta. [6, Item 15]

Antud töö autorile teadaolevalt, ei toeta ükski konkureeriv Android SQLite ORM teek sisseehitatult kasutaja defineeritud muutumatuid objekte. Seetõttu, on lisaks kirjeldatud klasside headele omadustele antud funktsionaalsuse toetamine ka hea võimalus konkurentidest eristumiseks.

Kuigi muutumatute klasside defineerimiseks on üpriski konkreetsed juhised ja nõudmised, võib iga programmeerija lahendus olla erinev. Sellest tulenevalt, lähenedes muutumatute klasside toetamise probleemile vaatenurgast, kus kõiksugused implementatsioonid on lubatud, selgub, et probleemi lahendus võib muutuda, kas liiga keeruliseks, raskesti arusaadavaks, ebamugavalt kasutatavaks või ebavajalikult pikaks. Vähegi mõistlikuma tulemuse saamiseks tuleks välja töötada standardne viis muutumatute klasside defineerimiseks, mis rakendaks kõiki nõudmisi korrektselt. Lisaks tuleks loodud lahendus kasutajale selgeks õpetada ning selle kasutamine piisavalt atraktiivseks teha, et asi kasutust leiaks. Kirjeldatu tulemusena sisaldaks loodav projekt kahte omavahel eraldatavat funktsionaalsust, mida saaks lahku lüüa kaheks eraldiseisvaks teegiks – andmebaasi ORM teek ja muutumatute objektide standardse defineerimise teek. Mõistlikum oleks uurida probleemi lahendavaid olemasolevaid projekte. Hea õnne korral leidub mõni teek, mis on nii laia kasutust leidnud, et seda võib nimetada kogukonna standardiks, mistõttu saab suure enesekindlusega projekti toe teeki integreerida.

Osutub, et Javas tuleb iga muutumatu klassi realiseerimiseks palju korduvat koodi kirjutada, mistõttu pakuvad nimetatud probleemile lahendust mitmed teegid. Androidi kogukonnas üheks populaarsemaks on Google'i poolt arendatava AutoValue projekti [5] laiendus AutoParcel [4]. Kuna nimetatud teegid on suure kasutajaskonnaga ning lahendavad muutumatute klasside defineerimise probleemi süstemaatilisel viisil, otsustati antud töö tulemusena valminud teegis muutumatuid objekte toetada nende vahendusel.

Olemuselt käsitleb SqliteMagic muutumatuid objekte, kui tavalisi kasutaja defineeritud objekte (vt ptk 2.2.2). Muutumatu klass defineeritakse vastavalt AutoValue teegi poolt nõutud reeglitele (vt Koodinäide 22). Andmebaasi salvestamiseks tuleb realiseeritud klass markeerida SqliteMagic teegi annotatsioonidega sarnaselt tavalisele viisile (vt ptk 2.2), kus ainsaks erinevuseks on tabeliveergude kirjeldajad – klassiväljade asemel kirjeldavad veergusid abstraktsed meetodid.

Kõik eelnevalt kirjeldatud muudetava objekti kohta käivad nõudmised ja lõdvendused kehtivad ka muutumatute objektide kohta paari väikse erinevusega:

- Tulenevalt AutoValue teegi eripärast peavad vaikimisi kõik objekti väärtused olema

täidetud. Antud käitumist saab muuta annotatsiooniga *@Nullable* (vt Koodinäide 22, rida 9). Kõike kirjeldatut arvestab ka SqliteMagic teek lisades genereeritavale koodile täidetavuse kontrolle ainult seal, kus vaja.

- Erinevalt muudetavatest klassidest, kus primaarvõtmele vastava välja puudumisel genereeritakse see „maagiliselt”, siis muutumatute klasside puhul kirjeldatud käitumist ei toetata. Sellest tulenevalt peab igal klassil olema üks *@Id* annotatsiooniga *long* tüüpi abstraktne meetod (vt Koodinäide 22, read 7 - 8). Kirjeldatud käitumise kasuks otsustati, et säilitada AutoValue teegi kasutamise loomulikkus.

```
1  @Table(persistAll = true)
2  @AutoParcel
3  public abstract class User {
4      User() {
5      }
6
7      @Id(autoIncrement = false)
8      public abstract long id();
9      @Nullable
10     public abstract String pictureUrl();
11     public abstract String username();
12     public abstract String firstName();
13     public abstract String lastName();
14
15     @NonNull
16     @CheckResult
17     public static Builder builder() {
18         return new AutoParcel_User.Builder();
19     }
20
21     @AutoParcel.Builder
22     public static abstract class Builder {
23         @CheckResult
24         public abstract Builder id(long id);
25         @CheckResult
26         public abstract Builder pictureUrl(@Nullable String url);
27         @CheckResult
28         public abstract Builder username(@NonNull String username);
29         @CheckResult
30         public abstract Builder firstName(@NonNull String firstName);
31         @CheckResult
32         public abstract Builder lastName(@NonNull String lastName);
33
34         @CheckResult
35         public abstract User build();
36     }
37 }
```

Koodinäide 22: SqliteMagic teegis andmebaasi kuuluva muutumatu andmeobjekti defineerimine koos ehitusklassiga.

AutoValue teek on esialgselt disainitud kasutamiseks tavalistes Java rakendustes ning omab suhteliselt väikest, kui väga konkreetset funktsionaalsuste komplekti. Seetõttu on kogukond teeki laiendanud ning lisanud erinevate kasutusjuhtude jaoks uusi funktsionaalsusi. Kirjeldatud fakti adresseerimiseks saab SqliteMagic teeki konfigureerida suvalist AutoValue laiendust kasutama. Rakenduse build.gradle failis tuleb defineerida SqliteMagic Gradle plugina konfiguratsiooni objekt, milles tuleb välja *autoValueAnnotation* väärtuseks omistada AutoValue teegi laienduse täispikkuses baasannotatsiooni nimi (vt Koodinäide 23).

```
1 // project level build.gradle
2 ...
3 sqlitemagic {
4     autoValueAnnotation = "auto.parcel.AutoParcel"
5 }
```

Koodinäide 23: AutoValue teegi laienduse konfigureerimine SqliteMagic teegis.

2.3 Androidi annotatsioonide tugi

Android Studio toetab koodi inspekteerimist eriliste koodi kohta metaandmeid lisavate annotatsioonide vahendusel [29]. Teisisõnu toetatakse lepingupõhist disaini (*Design by Contract* [14, Tip 31]), kus eel- ja järeltingimused on kirjeldatud annotatsioonidega. Kui kirjutatud kood ei vasta lepingu tingimustele teavitatakse programmeerijat rikkumisest infosõnumiga ning vastavalt rikkumise astmele värvitakse vigane kood teist värvi või tõmmatakse punane joon alla. Kõik kirjeldatu aitab ennetada programmeerimisel tekkivaid vigu võimalikult vara – lõpptulemusena kulub vähem aega vigade parandamisele.

Ennetamaks ebakorrektsest kasutamisest tekkivaid vigu on kogu teegi avalik API markeeritud Androidi annotatsioonidega. Lisaks saab annotatsioonide vahendusel teegile andmeväljade olemuse kohta vihjeid anda, tänu millele on võimalik andmebaasiga suhtluseks genereeritud koodi optimeerida. Näiteks andmeväljade puhul, mis on alati täidetud, pole väärtuse täidetavuse kontrolli (*null check*) vaja teostada. Markeerides nimetatud välju *@NonNull* annotatsiooniga (vt Koodinäide 20) oskab teek sellega arvestada ja jätab genereeritud koodis vastavate väljade puhul täidetavuse kontrolli tegemata, mis võib suurte andmehulkade korral operatsiooni teostamise kiirust märkimisväärselt suurendada.

2.4 API disain ja põhimõtted

Kogu teegi API on disainitud lihtsust ja intuitiivsust silmas pidades, mistõttu on palju aega pühendatud klasside, meetodite ja parameetrite nimetamise peale [21, ptk 2]. Üldine puhta koodi soovitus ütleb, et mida vähem on meetoditel parameetreid, seda lihtsam on neist aru saada ja neid kasutada [6, lk 189][21, lk 40]. Viimasest tulenevalt on kogu API ulatuses püütud iga avaliku meetodi parameetrite arv hoida võimalikult madalal ning alati maksimaalselt kahe parameetri juures. Kirjeldatu saavutamiseks kohtades, kus enam parameetreid on vajalik, kasutati tehnikat, kus parameetriks on abiklassi objekt, mille konstrueerimiseks kasutatakse ehitusmustrit [6, lk 190].

Teegi kasutamise lihtsustamiseks, vigade vältimiseks ja kasutaja suunamiseks kasutatakse kogu avaliku API ulatuses tehnikat, kus operatsiooni teostamiseks ehitatakse selle detaile sisaldav objekt kokku ehitusmustriga [6, Item 2]. Tulemusena saadakse muutumatu objekt, millel on defineeritud operatsiooni käivitamiseks kaks meetodit – üks teostab operatsiooni sünkroonselt, teine annab alustuskoha asünkroonse API maailma (vt ptk 2.7) [6, lk 190]. Kuna ehitatud operatsiooni objekt on muutumatu, siis saab seda kõrvalmõjudeta korduvalt käivitada, mistõttu ei pea seda iga kord uuesti ehitama [6, Item 15].

2.5 Andmete salvestamine ja muutmine

Teegi teostusdetailidest tulenevalt genereeritakse andmebaasiga suhtlust teostav kood annotatsiooniprotsessoriga (vt ptk 3.1). Teatavasti pole aga nimetatud viisil võimalik olemasolevat koodi muuta, mistõttu puudub märkamatu viis ühendamaks rakenduse ja genereeritud koodi. Seetõttu tuleks näiteks andmete salvestamiseks, kas teegi kasutajal genereeritud kood ise välja kutsuda või peaks teek pakkuma abiklasse vajalike operatsioonide teostamiseks. Esimest lahendust pakub näiteks muutumatute objektide defineerimise teek AutoValue (vt Koodinäide 22, rida 18). Teist lahendust pakuvad enamus annotatsiooniprotsessorite põhjal töötavatest Android SQLite ORM teekidest - sunnitakse laiendama baasklassi, millel on defineeritud CRUD operatsioonide meetodid, kuid mis reaalse implementatsiooni leiavad kasutades *reflectionit*. Kolmandaks lahenduseks oleks viia eelnevalt kirjeldatud baasklassi meetodid eraldiseisvatesse abiklassidesse.

Kokkuvõtlikult on probleemiks genereeritud koodi ühendamine rakenduse koodiga viisil, mis tunduks kõige intuitiivsem. Eesmärgiks seati olukord, kus teegi kasutamine oleks nii märkamatu, et selle poolt pakutud funktsionaalsused tunduvad, kui Java keele osad. Kirjeldatu saavutamiseks sobiks näiteks olukord, kus operatsioone teostavad meetodid asuvad otse andmeobjekti küljes. Näiteks, kui andmeobjekti saaks salvestada viisil `author.insert()` , poleks vaja meelde jätta abiklasside nimesid ega muretseda kõiksugu sõltuvusklasside instanside pärast ning unustades teegi poolt pakutavad operatsioonid või nende konkreetsed nimed, pakuks IDE koodivihjete funktsionaalsus (*code completion*) kiire ja intuitiivse võimaluse nimetatu meeldetuletamiseks.

Lahendus, kus genereeritud klasside meetodite väljakutsumine jäetakse teegi kasutaja hooleks võiks põhimõtteliselt sobida – poleks *reflectionit*; oleks väga selgelt aru saada, kus ja kuidas midagi tehakse ning ei seataks piiranguid andmeobjekti hierarhiale. Kahjuks nõuaks kirjeldatud lahendus kasutajalt palju korduva koodi kirjutamist, mida loodav teek peaks kõrvaldama. Isekirjutatavat koodi oleks tühiselt vähe võrreldes kogusega, mida tuleks tavaolukorras andmebaasiga suhtluse teostamiseks kirjutada, kuid töö autor leidis, et peab leiduma veelgi parem lahendus.

Lahendus, kus sunnitakse laiendama baasklassi vähendaks küll korduva koodi kirjutamist, kuid ei sobi mitmel põhjusel. Esiteks, kuna väga oluliseks peetakse teegi lihtsat kasutuselevõttu olemasolevasse projekti ilma andmeobjekte ümberkirjutamata või nende hierarhiat muutmata, siis kirjeldatud lahendus ei sobiks ilmselgetel põhjustel. Teiseks, kuna üheks teegi teostusdetailide nõudmiseks ja uurimisküsimuseks oli *reflectioni*-vaba realisatsioon, siis igasugune lahendus, kus operatsiooni teostavat genereeritud meetodit või klassi otsitakse *reflectioni* vahendusel, on täielikult välistatud. Kuigi konkurentide teegid püüavad *reflectioni* kasutamisest tulenevat kiiruskaotust kompenseerida realisatsiooniklasside puhverdamisega, siis laenatakse mälu arvelt, mida on mobiilsetes seadmetes niigi napilt.

Lahendus, kus CRUD operatsioone teostavad meetodid asuvad eraldiseisvates abiklassides lahendaks andmeobjektide ümberkirjutamise ja hierarhia probleemid. Samas, tänu faktile, et meetodid pole andmeobjekti küljes, ei tundu rakenduse arendusel kirjeldatud lahendus nii intuitiivsena, kui soovitud.

Kõigi kirjeldatud nõudmiste realiseerimiseks ainsaks lahenduseks sobiksid laiendusmeetodid (*extension methods*) nagu C# või Javascriptis, kuid Javas nimetatud funktsionaalsus puudub. Kuna kõik teised kirjeldatud lahendused ei sobinud otsustati „sohki” teha

ja „automaagiliselt” genereerida andmebaasioperatsioonide meetodid otse andmeobjektile (vt Koodinäide 24, rida 13 ja ptk 3.2.2). Kirjeldatud lahendus seob genereeritud koodi rakenduse koodiga kõige märkamatumal viisil säilitades kõik nõudmised ja head omadused – puudub *reflection* ja mõju mälule; ei seata piiranguid andmeobjektide hierarhiale; säilitatakse operatsioonide leidmise lihtsus.

Tulenevalt API põhimõtetest (vt ptk 2.4) on genereeritud meetodid alustuskohad operatsiooni detaile sisaldavatele ehitusklassidele (vt Koodinäide 24, rida 14). Operatsiooni reaalseks läbiviimiseks on kaks meetodit – *execute* ja *observe*. Meetod *execute* täidab operatsiooni koheselt ehk sünkroonselt (vt Koodinäide 25, read 1 - 4) ning tagastab operatsiooni tulemuse. Meetod *observe* on juurdepääsuks asünkroonse API maailma, mida realiseerib RxJava (vt Koodinäide 25, read 6 - 13 ja ptk 2.7).

```
1  @Table(persistAll = true)
2  public final class Author {
3      @NonNull
4      String firstName;
5      @NonNull
6      String lastName;
7
8      @Invokes(
9          value = "SqliteMagic_Author_Handler.PersistBuilder#create",
10         useThisAsOnlyParam = true
11     )
12     @CheckResult
13     public final PersistBuilder persist() {
14         return PersistBuilder.create(this);
15     }
16     ...
17 }
```

Koodinäide 24: SqliteMagic teegi poolt genereeritud meetod andmeobjektil.

Kokkuvõtlikult, tuleb iga operatsiooni reaalseks teostamiseks alati genereeritud meetodi poolt tagastaval objektil mingi meetod välja kutsuda. Kuna antud asjaolu on kerge ununema lisatakse kasutaja abistamiseks genereeritud meetoditele *@CheckResult* Androidi toetusannotatsioon (vt Koodinäide 24, rida 12), mille puhul jätkumeetodi puudumisel tõmmatakse koodile Android Studios punane joon alla.

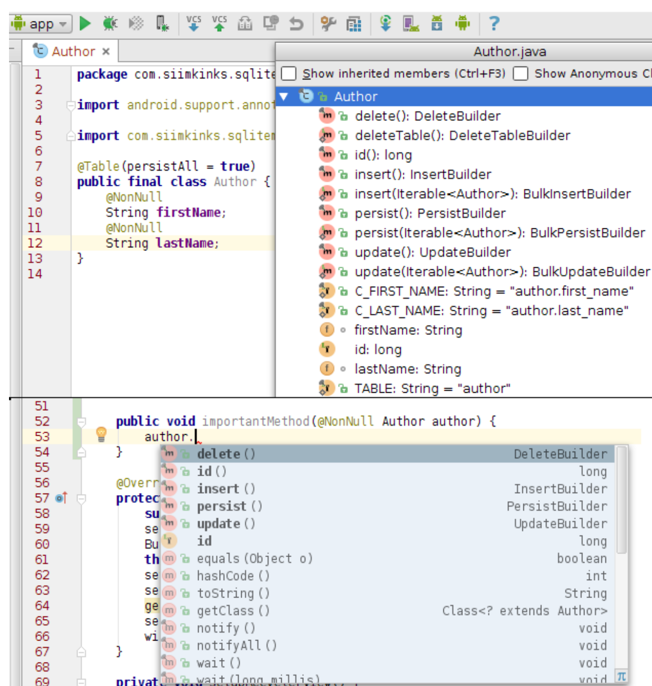
```
1  final long id = author
2          .persist()
3          .ignoreNullValues()
4          .execute();
5
6  final Subscription subscription = author
7          .persist()
8          .ignoreNullValues()
9          .observe()
10         .subscribeOn(Schedulers.io())
11         .observeOn(AndroidSchedulers.mainThread())
12         .subscribe(id -> infoTextView
13                 .setText("Saved author with id " + id));
```

Koodinäide 25: SqliteMagic teegi vahendusel andmeobjekti salvestamine.

Realiseeritud lahenduse ainsaks miinuseks on selle „maagilisus” ning Java keelereeglitest kõrvalekaldumine – genereeritud meetodeid pole lähtefaili avamisel näha. Teisalt on neid jällegi näha IDE vahendusel (vt Joonis 1). Lisaks, teegi pikemal kasutusel tunduvad genereeritud meetodid sama iseenesestmõistetavad, kui Java igas objektis alati olemasolevad

meetodid *equals*, *toString*, jne. Viimasega täidetakse ka püstitatud eesmärk, mis nägi ette olukorra, kus teegi funktsionaalsused tunduvad, kui Java keele osad.

Kogukonnas ollakse väga skeptilised igasuguste baitkoodi muudatuste osas. Viimane on enamasti põhjustatud olukorrast, kus inimesed ei näe konkreetset koodi, mida baitkoodi muudatused toodavad – vea tekkimisel on raske aru saada, millest see tuleneb. Lisaks, tulenevalt eelmisest, kurdetakse et baitkoodi muudatustega genereeritud koodi on võimatu siluda, kuna puudub inimloetav lähtekood. Kirjeldatud probleemide adresseerimiseks kasutab SqliteMagic teek baitkoodi muudatusi minimaalselt – lisatakse ainult n-ö „liimkood", mis ei oma muud loogikat, kui seob rakenduse koodi ja genereeritud inimloetava koodi (vt ptk 3.2.1). Tulemusena saab vaadelda „maagilist" meetodit (vt Koodinäide 24, rida 13), kui väljakutsujat reaalsele inimloetavale meetodile (vt Koodinäide 24, rida 14). Tänu kirjeldatule saab siluris realt-reale ja meetodi sisse liikudes jõuda inimloetava genereeritud lähtekoodini, mille silumine käib tavalisel viisil.



Joonis 1: SqliteMagic teegi genereeritud meetodid vaadatuna Android Studios.

2.5.1 Salvestamine

Andmete salvestamiseks vajaliku operatsiooni ehitusobjekti juurdepääs käib läbi andmemudelile „maagiliselt" genereeritavate meetodite:

- *insert()* – defineerib operatsiooni, kus objekt salvestatakse kasutades INSERT lauset (vt Koodinäide 26, rida 13). Tulemusmeetodid produtseerivad salvestatud kirje primaarvõtme.
- *insert(Iterable<T>)* – staatiline meetod, mis defineerib operatsiooni, kus salvestatakse parameetrina etteantav itereeritav objektide kogum transaktsioonis kasutades INSERT lauset (vt Koodinäide 26, rida 28). Sünkroonne tulemusmeetod tagastab

tõeväärtuse, näitamaks operatsiooni õnnestumist. Asünkroonne tulemusmeetod annab allavoolu *OperationResult* objekte, milles sisaldub salvestatud objekti primaarvõti ja objekt ise.

- *persist()* – defineerib operatsiooni, kus objekt salvestatakse kasutades INSERT lauset, kui selle primaarvõtmele vastav kirje pole veel andmebaasis. Vastasel korral uuendatakse primaarvõtmele vastavat kirjet UPDATE lausega (vt Koodinäide 26, rida 22). Tulemusmeetodid produtseerivad salvestatud kirje primaarvõtme (vt Koodinäide 25, read 1 ja 12).
- *persist(Iterable<T>)* – staatiline meetod, mis defineerib operatsiooni, kus salvestatakse parameetrina etteantav itereeritav objektide kogum transaktsioonis kasutades INSERT lauset, kui konkreetse itereeritava objekti primaarvõtmele vastav kirje pole veel andmebaasis. Vastasel korral uuendatakse primaarvõtmele vastavat kirjet UPDATE lausega (vt Koodinäide 26, rida 34). Sünkroonne tulemusmeetod tagastab tõeväärtuse (vt Koodinäide 27, rida 2), näitamaks operatsiooni õnnestumist. Asünkroonne tulemusmeetod annab allavoolu *OperationResult* objekte, milles sisaldub salvestatud objekti primaarvõti ja objekt ise (vt Koodinäide 27, read 5 - 12).

```
1  @Table(persistAll = true)
2  public final class Author {
3      @NonNull
4      String firstName;
5      @NonNull
6      String lastName;
7
8      @Invokes(
9          value = "SqliteMagic_Author_Handler.InsertBuilder#create",
10         useThisAsOnlyParam = true
11     )
12     @CheckResult
13     public final InsertBuilder insert() {
14         return InsertBuilder.create(this);
15     }
16
17     @Invokes(
18         value = "SqliteMagic_Author_Handler.PersistBuilder#create",
19         useThisAsOnlyParam = true
20     )
21     @CheckResult
22     public final PersistBuilder persist() {
23         return PersistBuilder.create(this);
24     }
25
26     @Invokes("SqliteMagic_Author_Handler.BulkInsertBuilder#create")
27     @CheckResult
28     public static BulkInsertBuilder insert(Iterable<Author> var0) {
29         return BulkInsertBuilder.create(var0);
30     }
31
32     @Invokes("SqliteMagic_Author_Handler.BulkPersistBuilder#create")
33     @CheckResult
34     public static BulkPersistBuilder persist(Iterable<Author> var0) {
35         return BulkPersistBuilder.create(var0);
36     }
37     ...
38 }
```

Koodinäide 26: SqliteMagic teegi poolt genereeritud salvestamismeetodid andmeobjektil.

```
final List<Author> authors = ...
final boolean success = Author
        .insert(authors)
        .execute();
final Subscription subscription = Author
        .insert(authors)
        .observe()
        .map(opResult -> opResult.entity)
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(author -> infoTextView
                .setText("Inserted author " + author));
```

Koodinäide 27: SqliteMagic teegi vahendusel andmeobjektide hulgi lisamine.

Kõikide andmeid salvestavate operatsioonide puhul üritatakse kõikjal, kus võimalik korrigeerida primaarvõtme väärtust objektil automaatselt. Näiteks, kui primaarvõtmel on defineeritud automaatne kasvamine (`@Id(autoincrement = true)`) ning tegemist on muudetava klassiga, siis muudetakse peale salvestamist primaarvõtme väärtust ka objektil. Vastunäiteliselt, kui tegemist on muutmatu objektiga, siis uut objekti salvestatud primaarvõtmega ei looda. Mälu kokkuhoiu mõttes jäetakse viimane tegevus kasutaja teha.

2.5.2 Uuendamine

Andmete uuendamiseks on kaks võimalust – konkreetsete andmeobjektide uuendamine operatsiooni ehitusobjektide vahendusel, mis genereeritakse andmemudelile ning teegi eeldefineeritud SQL UPDATE lauset konstrueeriva ehitusobjekti vahendusel.

```
@Table(persistAll = true)
public final class Author {
    @NonNull
    String firstName;
    @NonNull
    String lastName;

    @Invokes(
        value = "SqliteMagic_Author_Handler.UpdateBuilder#create",
        useThisAsOnlyParam = true
    )
    @CheckResult
    public final UpdateBuilder update() {
        return UpdateBuilder.create(this);
    }

    @Invokes("SqliteMagic_Author_Handler.BulkUpdateBuilder#create")
    @CheckResult
    public static BulkUpdateBuilder update(Iterable<Author> var0) {
        return BulkUpdateBuilder.create(var0);
    }
    ...
}
```

Koodinäide 28: SqliteMagic teegi poolt genereeritud uuendamismeetodid andmeobjektil.

Andmemudelile genereeritakse järgnevad uuendusoperatsioonide ehitusobjektide juurdepääsu meetodid:

- *update()* – defineerib operatsiooni, kus objekt uuendatakse kasutades UPDATE lauset (vt Koodinäide 28, rida 13). Tulemusmeetodid produtseerivad tõeväärtuse, mis väljendab operatsiooni õnnestumist (vt Koodinäide 29, read 1 ja 9).

- *update(Iterable<T>)* – staatiline meetod, mis defineerib operatsiooni, kus uuendatakse parameetrina etteantav itereeritav objektide kogum transaktsioonis kasutades UPDATE lauset (vt Koodinäide 28, rida 19). Sünkroonne tulemusmeetod tagastab tõeväärtuse, näitamaks operatsiooni õnnestumist; asünkroonne tulemusmeetod annab allavoolu uuendatud objekte.

```
1  final boolean success = author
2          .update()
3          .conflictAlgorithm(SQLiteDatabase.CONFLICT_ROLLBACK)
4          .execute();
5  final Subscription subscription = author
6          .update()
7          .conflictAlgorithm(SQLiteDatabase.CONFLICT_ROLLBACK)
8          .observe()
9          .map(success -> success ? "succeeded" : "failed")
10         .subscribeOn(Schedulers.io())
11         .observeOn(AndroidSchedulers.mainThread())
12         .subscribe(status -> infoTextView
13                 .setText("Updating " + status));
```

Koodinäide 29: SqliteMagic teegi vahendusel andmeobjekti uuendamine.

Andmebaasi tabelite uuendamiseks konkreetsest objektist sõltumatult pakub teek n-ö „anonüümse” SQL UPDATE lause ehitamise liidest (vt Koodinäide 30, read 2 - 6). Viimane järgib teegi API põhimõtteid (vt ptk 2.4) – operatsiooni detailid ehitatakse kokku ehitusmustriga ning operatsiooni käivitamiseks on üks meetod sünkroonseks (rida 8) ning üks asünkroonseks täidesaatmiseks (read 9 - 14). Mõlema meetodi tulemusväärtuseks on *int* andmetüüp, mis väljendab muudetud ridade arvu. Erinevalt konkreetsete objektide kohta käivatest operatsiooni ehitusobjektidest, saab anonüümset uuendamise operatsiooni kompileerida (rida 6) – konstrueerida muutumatu objekt, mida saab korduvalt käivitada ja lõimede vahel jagada.

```
1  public static final CompiledUpdate FIX_ALL_ROWLING_FIRST_NAMES =
2          Update
3          .table(Author.class)
4          .set(Column.withName(Author.C_FIRST_NAME).to("Joanne"))
5          .where(column(Author.C_LAST_NAME).is("Rowling"))
6          .compile();
7  ...
8  final int affectedRows = FIX_ALL_ROWLING_FIRST_NAMES.execute();
9  final Subscription subscription = FIX_ALL_ROWLING_FIRST_NAMES
10         .observe()
11         .subscribeOn(Schedulers.io())
12         .observeOn(AndroidSchedulers.mainThread())
13         .subscribe(affectedRows -> infoTextView
14                 .setText("Fixed " + affectedRows + " author names"));
```

Koodinäide 30: SqliteMagic teegis uuendamislause defineerimine ja käivitamine.

2.5.3 Kustutamine

Sarnaselt uuendamisele (vt ptk 2.5.2) on kustutamiseks kaks võimalust – konkreetse andmeobjekti kustutamine genereeritud operatsiooni ehitusobjekti vahendusel ning teegi eeldefineeritud DELETE lauset konstrueeriva ehitusobjekti vahendusel.

Andmemudelile genereeritakse järgnevad kustutusoperatsioonide ehitusobjektide juurdepääsu meetodid:

- *delete()* – defineerib operatsiooni, kus objekt kustutatakse kasutades DELETE lauset (vt Koodinäide 31, rida 13). Tulemusmeetodid produtseerivad *int* andmetüübi, mis väljendab kustutatud ridade arvu (vt Koodinäide 32, read 1 ja 7).
- *deleteTable()* – staatiline meetod, mis defineerib operatsiooni, kus terve andmetüübi tabel kustutatakse (vt Koodinäide 31, rida 19). Tulemusmeetodid produtseerivad *int* andmetüübi, mis väljendab kustutatud ridade arvu.

```
1  @Table(persistAll = true)
2  public final class Author {
3      @NonNull
4      String firstName;
5      @NonNull
6      String lastName;
7
8      @Invokes(
9          value = "SqliteMagic_Author_Handler.DeleteBuilder#create",
10         useThisAsOnlyParam = true
11     )
12     @CheckResult
13     public final DeleteBuilder delete() {
14         return DeleteBuilder.create(this);
15     }
16
17     @Invokes("SqliteMagic_Author_Handler.DeleteTableBuilder#create")
18     @CheckResult
19     public static DeleteTableBuilder deleteTable() {
20         return DeleteTableBuilder.create();
21     }
22     ...
23 }
```

Koodinäide 31: SqliteMagic teegi poolt genereeritud kustutamismeetodid andmeobjektil.

```
1  final int affectedRows = author
2          .delete()
3          .execute();
4  final Subscription subscription = author
5          .delete()
6          .observe()
7          .map(affectedRows -> affectedRows > 0 ? "succeeded" : "failed")
8          .subscribeOn(Schedulers.io())
9          .observeOn(AndroidSchedulers.mainThread())
10         .subscribe(status -> infoTextView
11                 .setText("Deleting " + status));
```

Koodinäide 32: SqliteMagic teegi vahendusel andmeobjekti kustutamine.

Andmebaasi ridade kustutamiseks konkreetsest objektist sõltumatult pakub teek n-ö „anonüümse" SQL DELETE lause ehitamise liidest (vt Koodinäide 33, read 2 - 6). Viimane on peaaegu identne tabelite uuendamise liidesega (vt ptk 2.5.2).

```java
private static final CompiledDelete DELETE_ALL_GEORGE_MARTINS =
        Delete
        .from(Author.class)
        .where(column(Author.C_LAST_NAME).is("Martin"))
        .and(column(Author.C_FIRST_NAME).like("George%"))
        .compile();
...
final int affectedRows = DELETE_ALL_GEORGE_MARTINS.execute();
final Subscription subscription = DELETE_ALL_GEORGE_MARTINS
        .observe()
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(affectedRows -> infoTextView
                .setText("Deleted " + affectedRows + " author names"));
```

Koodinäide 33: SqliteMagic teegis kustutamislause defineerimine ja käivitamine.

2.6 Andmete pärimine

Tekstina sõnesse kirjutatud SQL laused on üpriski aldid vigadele – tihti tekivad, kas vähestest teadmistest või näpuvigadest tulenevad süntaksi ja loogikavead. Eriti halvaks läheb asi andmebaasi struktuuri muudatustega, kus tuleks kogu rakenduse kood üle käia, et käsitsi kõik uuendused sisse viia ning tihti jääb kuskil midagi märkamata, mis võib tekitada vigu rakenduse elu hiljemates staadiumites jne.

Kirjeldatud puudustest tulenevalt otsustati antud töös kasutada ehitusmustriga päringute konstrueerimise mehhanismi (vt Koodinäide 34). Lisaks ohutumale muudatuste läbiviimisele võimaldab mehhanism kasutajat suunata korrektset SQLi kirjutama. Näiteks on viga panna LIMIT klausel enne ORDER BY klauslit – ehitusmustriga on võimalik korraldada olukord, kus peale LIMIT klausli defineerimist pole võimalik muid klausleid järgnevalt defineerida. Üleüldiselt on Java koodis defineeritud SQL vähema müra tõttu ka loetavam, kui sõnena defineeritud SQL.

```java
final List<Author> authors = Select
        .columns(Author.C_LAST_NAME)
        .from(Author.class)
        .where(column(Author.C_FIRST_NAME).like("George%"))
        .and(column(Author.C_LAST_NAME).isNot("Martin"))
        .order(by(Author.C_FIRST_NAME))
        .limit(10)
        .execute();
```

Koodinäide 34: SqliteMagic teegis päringu defineerimine ja käivitamine.

Päringute konstrueerimise mehhanismis arvestatakse teegi API põhimõtetega (vt ptk 2.4) – päring defineeritakse ehitusmustriga, mille tulemuseks on muutmatu ja korduvalt käivitatav objekt. Sarnaselt kõigele eelnevale on päringu käivitamiseks kaks meetodit – *execute* ja *observe*.

Vaikimisi konstrueeritakse operatsioon, mis tagastab päringule vastava loendi „pealiskaudsetest” objektidest. Viimaseks nimetatakse objekte, mille kõik olemasolevad primitiivsed väljad on täidetud vastavalt salvestatusele, kuid mille kompleksväljad on täidetud nii vähe kui võimalik – muutuvatel objektidel on alati täidetud ainult identifikaator ning muutumatud objektid on täidetud nii madalalt, kui andmestruktuur lubab. Kirjeldatu on vaikimisi käitumiseks valitud, et halvimal juhul teamatusest kogemata tervet andmebaasi päringuga ei tagastataks.

Päringu konstrueerimisel on kaks faasi – operatsiooni defineerimine ja kompileerimine. Defineerimisfaasis kirjeldatakse päringule vastav SQL lause teegi meetoditega. Kompileerimisfaasis ehitatakse kokku konkreetne SQL lause, kogutakse selle argumendid, tehakse analüüs objektidest, mis tuleb tagastada ning vajadusel täiustatakse SQLi, et pärida sügavamaid objekte ning kaardistatakse veerud, mis kuuluvad pärimisele. Kompileerimisfaasi tulemusena saadakse muutumatu päringu operatsiooni objekt, mida saab korduvalt kõrvalmõjudeta käivitada ja lõimede vahel jagada. Päringu operatsiooni kompileerimist saab eraldi välja kutsuda meetodiga *compile*, kuid operatsiooni käivitamismeetodid ja meetodid *takeFirst*, *count*, *toCursor* teevad seda ka automaatselt, mistõttu eraldi väljakutsumine pole kohustuslik.

Kuna teek toetab kasutaja defineeritud andmetüüpide salvestamist, siis on ka päringute puhul neile erilist tähelepanu pööratud. Kui andmemudel sisaldab kompleksvälju, siis on ilmne, et kogu andmemudeli pärimiseks tuleb ka need täita, mis tähendab, aga kirje pärimist mõnest teisest tabelist. Kuna iga kompleksvälja eraldi pärimine mõjuks kiirusele laastavalt, siis päritakse kõikidest nõutud tabelitest andmed korraga ühe päringuga. Viimase saavutamiseks tuleb kõik tabelid päringus ühendada JOIN klauslitega. Kuna vähegi keerukamate andmete korral (vt Koodinäide 35) oleks kirjeldatu käsitsi kirjutamine äärmiselt tülikas ning aldis vigadele, siis päringute kompileerimisfaasis täiustatakse ehitatavat SQLi (vt Koodinäide 35, read 29 - 32) puuduvate klauslitega (vt Koodinäide 36). Kirjeldatud käitumine arvestab kasutaja defineeritud päringu osadega lisades puuduvaid klausleid vastavalt sellele.

```
1  @Table(persistAll = true)
2  public final class Country {
3      @NonNull
4      String name;
5      @NonNull
6      String code;
7  }
8
9  @Table(persistAll = true)
10 public final class Author {
11     @NonNull
12     String firstName;
13     @NonNull
14     String lastName;
15     @NonNull
16     Country birthCountry;
17 }
18
19 @Table(persistAll = true)
20 public final class Book {
21     @NonNull
22     String title;
23     @NonNull
24     Author author;
25     @NonNull
26     Author coAuthor;
27 }
28
29 final CompiledSelect<Book> compiledSelect = Select
30         .from(Book.class)
31         .queryDeep()
32         .compile();
```

Koodinäide 35: SqliteMagic teegis kompleksobjekti struktuur ning päringu defineerimine.

```sql
1 SELECT *
2 FROM book
3 LEFT JOIN author AS FGyqNc ON book.author=FGyqNc._id
4 LEFT JOIN country AS otabMl ON FGyqNc.birth_country=otabMl._id
5 LEFT JOIN author AS peIzzK ON book.co_author=peIzzK._id
6 LEFT JOIN country AS LpXICB ON peIzzK.birth_country=LpXICB._id;
```

Koodinäide 36: SqliteMagic teegis kompleksobjekti päringu põhjal ehitatav SQL.

Vaikimisi päringu käitumise muutmiseks on mehhanismis erilised meetodid:

- *queryDeep()* – seadistab operatsiooni pärima andmeid terves ulatuses vastavalt defineeritud päringule – kui andmeobjekt sisaldab kasutaja defineeritud komplektseid väljasid, siis päritakse ka need ja omakorda nende sõltuvused jne, kuni võimaliku maksimaalse sügavuseni, kuid mitte rohkem, kui FROM klausel ette näeb. Meetod tagastab operatsiooni ehitamisobjekti.
- *takeFirst()* – seadistab operatsiooni pärima ainult esimest päringule vastavat objekti. Meetod tagastab muutumatu operatsiooni objekti.
- *toCursor()* – seadistab operatsiooni tagastama päringu tulemustele vastavat Androidi *Cursor* objekti. Meetod tagastab muutumatu operatsiooni objekti, millel on meetod *getFromCurrentPosition*, mille vahendusel on võimalik kursori hetkeasukohast saada reaalne päritud objekt.
- *count()* – seadistab operatsiooni tagastama *long* andmetüübi, mis väljendab päringu tulemusele vastavate kirjete arvu. Meetod tagastab muutumatu operatsiooni objekti.

2.7 Paralleeltöötlus

Vaadates olemasolevaid SQLite ORM lahendusi kerkib esile nähtus, kus iga teek, kas üritab paralleeltöötlust omal viisil teostada või on sellest üldse loobutud. Lõpptulemuseks on ebamäärane API, mis on kas piiratud funktsionaalusega, liiga keerukas, vähe testitud või nende kõikide kombinatsioon.

Viimasel ajal on kogukonna standardiks muutunud Netflixi poolt arendatav avatud lähtekoodiga projekt RxJava [25]. Projekt kirjeldab end kui teeki koostamaks asünkroonseid ja sündmustel põhinevaid rakendusi Java virtuaalmasinas kasutades jälgitavaid jadasid (*observable sequences*). Kuna antud teek on väga laialdaselt kasutuses, pakub põhjalikult testitud, väga paindlikku ja võimast APIt, siis antud töös otsustati andmebaasi operatsioonide asünkroonset teostust toetada RxJava vahendusel.

Kuigi RxJava võimaldab paralleeltöötlust on selle operatsioonid vaikimisi sünkroonsed. Paralleeltöötluseks tuleb teegile eraldi öelda, kus lõimes operatsioone jooksutatakse meetodiga *subscribeOn* ning kus lõimes operatsioone vaadeldakse meetodiga *observeOn*. Tänu kirjeldatud mehhanismile, on esiteks koodi lugedes väga selgelt näha, mis lõimes teatud töö toimub ning teiseks annab võimaluse vajaduse korral kergelt tööd teostavaid lõimi vahetada. Kõik kirjeldatu on üpriski oluline lihtsustus Androidis, kus näiteks kasutajaliidest muutev kood peab alati jooksma põhilõimes (vt Koodinäide 37, rida 16).

```java
final Subscription subscription = RxTextView
    .textChanges(searchInputView)
    .map(CharSequence::toString)
    .debounce(2, TimeUnit.SECONDS)
    .filter(notEmptyString())
    .map(searchInput -> "%" + searchInput + "%")
    .switchMap(searchCondition -> Select
            .from(Author.class)
            .where(column(Author.C_FIRST_NAME).like(searchCondition))
            .or(column(Author.C_LAST_NAME).like(searchCondition))
            .order(by(Author.C_LAST_NAME))
            .observe()
            .runQueryOnce())
    .distinctUntilChanged()
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(view::renderSearchResults);
```

Koodinäide 37: Kohaliku andmebaasi otsingu näide kasutades SqliteMagic ja RxJava teekide kombinatsiooni.

Vastavalt teegi API põhimõtetele (vt ptk 2.4) on igal ehitatud operatsioonil meetod *observe* (vt Koodinäide 37, rida 12), mis on alustuskohaks jälgitavate jadade defineerimisele, mida pakub RxJava. Lisaks saab loodud operatsioone kombineerida teiste jälgitavate jadadega, mistõttu on võimalused avarad (vt Koodinäide 37).

2.7.1 Tabeli andmete muudatuste teavitamine

RxJava teegi kasutamine annab võimaluse realiseerida reaktiivsete voogude (*reactive streams*) semantika päringuoperatsioonides[14]. Teisisõnu, avab päringu *observe* meetod n-ö „kraani" päringuga seotud tabelite andmetele, kus muudatuste tekkimisel antakse allavoolu teavitusi. Kirjeldatud andmevoo käitumist nimetatakse „kuumaks" vooks – andmeallikas teavitab kuulajaid lõpmatult, kuni kuulamine lõpetatakse. Vastavalt päringu operatsiooni tüübile antakse allavoolu päringu objekte, millel on meetodid reaalseid andmeid tagastava päringu sooritamiseks. Viimasest tulenevalt on kasutajal võimalus vastavalt olukorrale, kas teavituse peale andmeid pärida, teavitust ignoreerida või rakendada mõnda muud äriloogikat.

Koodinäide 38 demonstreerib tabelite andmete muudatustest teavitamist. Päritakse *Book* tüüpi objekte selle täies mahus (sisaldab *Author* tüüpi kompleksväljasid; vt Koodinäide 35) ning iga teavituse peale suurendatakse loendajat. Peale esialgset voogu ühendumist (rida 6) teavitatakse kuulajat kohese teavitusega (rida 9), et oleks võimalus viivitamatult päringu tulemusi saada. Iga tabelite *Author* (rida 11) ja *Book* (read 13 ja 16) muudatuse peale teavitatakse andmevoo kuulajat (read 14 ja 17). Transaktsioonide puhul teavitatakse peale transaktsiooni õnnestumist kuulajat ainult ühe korra (rida 25). Teavitusi saadakse seni, kuni andmevoogu tellitakse - tellimuse katkestamisel (rida 27) teavituste edastamine lõpetatakse (rida 30).

[14]Realisatsiooni idee pärineb SQLBrite teegist [26].

```
final AtomicInteger queries = new AtomicInteger();
final Subscription subscription = Select
        .from(Book.class)
        .queryDeep()
        .observe()
        .subscribe(bookListQuery -> {
            queries.getAndIncrement();
        });
System.out.println(queries.get()); // Prints 1

createRandomAuthor().insert().execute();
final Book randomBook = createRandomBook();
randomBook.insert().execute();
System.out.println(queries.get()); // Prints 3

randomBook.update().execute();
System.out.println(queries.get()); // Prints 4

try (Transaction transaction = SqliteMagic.newTransaction()) {
    for (int i = 0; i < 15; i++) {
        createRandomBook().insert().execute();
    }
    transaction.markSuccessful();
}
System.out.println(queries.get()); // Prints 5

subscription.unsubscribe();
createRandomBook().insert().execute();
createRandomBook().insert().execute();
System.out.println(queries.get()); // Prints 5
```

Koodinäide 38: SqliteMagic teegis päringu kuulajale tabelite andmete muudatustest teavitamine.

2.7.2 RxJava mugavusoperaatorid

Vastavalt päringu operatsiooni tüübile annab *observe* meetod allavoolu päringu objekte, kus reaalsete andmete saamiseks tuleb objektil teatud meetod välja kutsuda. Kuna kirjeldatud tegevust tuleks erilise äriloogika puudumisel korduvalt teha, siis pakub teek RxJava operaatoreid koodi lühendamiseks.

Järgnevalt on toodud nimekiri SqliteMagic teegis realiseeritud mugavusoperaatoritest:

- *runQuery()* – muudab andmevoogu andes allavoolu teavitusest tulnud päringu reaalsed tulemused.
- *runQueryOnce()* – muudab andmevoogu lõpetama peale esimese teavituse saabumist ning annab allavoolu teavitusest tulnud päringu reaalsed tulemused.
- *isZero()* – muudab loenduspäringu andmevoogu andes allavoolu tõeväärtustüübi, mille tõene väärtus tähendab, et päringu loendustulemus on võrdne nulliga.
- *isNotZero()* – sarnane operaatoriga *isZero*, kuid mille tõene väärtus tähendab, et päringu loendustulemus on nullist erinev.

2.8 Kasutaja abistamine

Tänu annotatsiooniprotsessorile saab valesti kirjutatud koodist kasutajale teada anda juba kompileerimise ajal. Esiteks vähendab viimane valest konfiguratsioonist tulenevaid vigu, teiseks vähendab vigade leidmisele kuluvat aega – kogu arenduseks vajaminev aeg

väheneb. Sõltuvalt vea tõsidusest saab kasutajat teavitada, kas sõbraliku hoiatussõnumiga või isegi ehitustsükli katkestamisega märkides ehituse ebaõnnestunuks.

On mitmeid sorti probleeme, mida saab kasutajale teavitada kirjeldatud meetodiga. Näiteks võib vaadelda andmebaasi struktuuri, kui hulka graafidest. Teadagi ei tohiks olla ringsõltuvusi tabelite vahel – siit tulenevalt kasutatakse topoloogilist sorteerimisalgoritmi [8, ptk 22.4] ning kontrollitakse andmebaasi struktuuri korrektsust. Tulevikus saaks andmebaasi struktuurist mõne graafijoonistusprogrammiga ka inimloetava pildi tekitada.

Lisaks teavitatakse kasutajat API valest kasutusest. Näiteks, kui nõutakse, et *@Table* annotatsiooni kasutatakse ainult klassi failidel, kuid mitte liidestel, siis sellest antakse kasutajale teada kompileerimise hetkel. Teiseks teavitatakse puudulikust või valesti konfigureeritud andmestruktuurist. Näiteks *@Id* annotatsiooni puudumisel muutumatu klassi definitsioonis märgitakse ehitustsükkel ebaõnnestunuks, kuna andmebaasiga suhtlust teostavat koodi pole võimalik kirjeldatud andmete põhjal genereerida.

2.9 Andmebaasi migratsioon

Olemasolevates lahendustes on mitmeid erinevaid andmebaasi migratsiooni teostamise mustreid – otse Java koodi kirjutades, SQL skriptid jne. Tulenevalt Androidis kasutatavale DalvikVM arhitektuurile, on ühes rakenduses lubatud maksimaalselt 65,536 meetodit [7]. Selle limiidi ületamisel tuleb kasutusele võtta erilised meetmed, et rakendus kompileeruks ja suudaks käivituda. Viimase tulemusena suureneb rakenduse ehitusaeg ja ka startimisaeg lõppkasutaja seadmes. Kõigest nimetatust tulenevalt tasub kokku hoida uute meetodite kasutamisest, kus mõistlik ja võimalik. Seetõttu pole hea iga uue andmebaasi migratsiooni jaoks kirjutada Java koodi.

Paremaks lahenduseks on käivitada SQL faile, kus migratsioonisammud defineeritakse SQL lausetena, mida antud teek rakendabki. Lisaks eelmainitud probleemile parandab antud lahendus ka loetavust ja arusaadavust – konkreetselt on aru saada, mis operatsioone teostatakse ning vajadusel saab päringuid käsitsi modifitseerida.

Antud teegis toimub migratsioon järgnevalt:

- Iga andmebaasi versiooni kohta peab Androidi *assets* kaustas olema defineeritud migratsiooni fail kujul *<andmebaasi versioon>.sql.*
- Kui rakendus käivitub uuendatud andmebaasi versiooniga, siis jooksutatakse transaktsioonis iga uuendatud versiooni kohta sellele vastav migratsiooni fail.

3 Teostusdetailid

Järgnevalt kirjeldatakse teegi funktsionaalsuste realiseerimise võtmedetaile.

3.1 Annotatsiooniprotsessor

Kogu andmebaasiga suhtlust teostav kood genereeritakse Java annotatsiooniprotsessori vahendusel. Viimane on ehitustööriist *javac*-s (Java kompilaatoris), mis on mõeldud annotatsioonide skaneerimiseks ja protsessimiseks kompileerimise ajal. Valminud teegis skaneeritakse *@Table* ja *@ObjectTransformer* annotatsioone. Esimese vahendusel leitakse kõik andmebaasi kuuluvad tabelid ning kaardistatakse kogu andmebaasi struktuur. *@ObjectTransformer* annotatsiooni vahendusel leitakse kõik defineeritud tüübiteisendajad, mida kasutatakse hiljem genereeritavas koodis vastavate andmetüüpide teisendamiseks.

Skaneeritud andmete põhjal genereeritakse iga tabeli kohta kaks klassi - andmemudeliga samas paketis asuv andmete lugemise, kirjutamise ja loomisega tegelev klass[15]; ning teegi juurpaketis asuv andmebaasiga suhtlust teostav ja tabeli loomislauset sisaldav klass[16]. Lisaks genereeritakse klass nimega *GeneratedClassesManager*, mis koondab kõikide tabelite kohta käivad andmed. Viimane töötab viidana, mis oskab konkreetse tabeli andmete või operatsioonide otsija kokku viia reaalsete andmetega või operatsiooni realisatsiooniga. Viimast kasutatakse näiteks andmebaasi loomisel, andmeobjekti klassi nime järgi tabeli nime leidmisel, päringutes reaalset operatsiooni teostava meetodi leidmisel jne.

Erinevalt annotatsioone rakendavatest konkureerivatest teekidest, kus annotatsioonide vahendusel tuleb andmete kohta üpriski palju informatsiooni käsitsi defineerida (vt Koodinäide 13), nõuab loodud teek kasutajalt andmeid võimalikult minimaalselt. Näiteks piisab kogu tabeli defineerimiseks ainult `@Table(persistAll = true)` annotatsioonist. Kirjeldatu on võimalik tänu põhjalikule andmete analüüsile - eriomadusteta tabeli loomiseks ja andmebaasiga suhtluse korraldamiseks vajalik informatsioon on täielikult tuletatav Java andmemudeli omadustest. Ainsad andmed, mille defineerimine on möödapääsmatu on need, mille semantikat pole võimalik ainult andmestruktuuri teades tuletada nagu välisvõtme tegevused (*foreign key constraint actions*) või tabeli primaarvõti. Teine põhjus rohkemate andmete defineerimiseks on olukord, kus soovitakse vaikmisi käitumisest erinevaid tulemusi – nimetada tabeleid või veergusid erilisel viisil; defineerida väljasid, mis ei kuulu salvestamisele või kompleksväljasid, mida ei tuleks salvestada täies mahus.

3.2 *Reflectioni* kõrvaldamine

Annotatsiooniprotsessori vahendusel andmebaasiga suhtlust teostava koodi genereerimine on suur samm *reflectioni* kõrvaldamises. Kuid nagu peatükis 2.5 mainitud, on siiski probleemiks genereeritud koodi ühendamine rakenduse koodiga. Parimaks, teegi kasutusmugavust suurendavaks ja *reflectioni*-vabaks lahenduseks kujunes andmebaasioperatsioonide meetodite genereerimine tabelina defineeritud andmetüübile (vt ptk 2.5). Kahjuks Java keelereeglid ei võimalda laiendusmeetodite kasutamist, mistõttu nõudis lahenduse realiseerimine erilist lähenemist.

[15]Gereeritud klassi nimi on stiilis SqliteMagic_*tabeli klassi nimi*_Dao.
[16]Gereeritud klassi nimi on stiilis SqliteMagic_*tabeli klassi nimi*_Handler.

Lisaks laiendusmeetoditele nõuab *reflectioni* kõrvaldamine genereeritud koodi sidumist teegisisese koodiga. Näiteks kasutatakse päringu sooritamiseks teegi poolt eeldefineeritud meetodeid, kuid nende sees on vaja viidet vastava tabeli reaalset päringut teostavale genereeritud meetodile.

Realiseeritud lahendus koosneb mitmest osast. Järgnevalt on detailselt kirjeldatud iga osa ning viis kuidas kõik omavahel kokku sobitub ja tööle hakkab.

3.2.1 Teegi ja genereeritud koodi ühendamine

Esiteks loodi süsteem, kus annotatsioonide vahendusel on võimalik defineerida anonüümseid väljakutsumissihtmärke ja väljakutsujaid, mis kompileerimisel baitkoodi muudatusi tehes ühendatakse. Väljakutsumissihtmärgid defineeritakse annotatsiooniga *@InvocationTarget*, mille parameetriks on sõnest võti, mis peab projekti piires olema unikaalne[17] (vt Koodinäide 39, rida 2). Väljakutsujad defineeritakse annotatsiooniga *@Invokes*, mille parameetriks on väljakutsutava meetodi *@InvocationTarget* annotatsiooni parameetri väärtus (rida 19). Rakenduse kompileerimisel asendatakse väljakutsuva meetodi sisu väljakutsutava meetodi väljakutsega (rida 30). Siinkohal on oluline mainida, et väljakutsutav meetod peab olema staatiline. Nimetatud piirang on nõutud esiteks lihtsuse huvides, teiseks tuleneb androidispetsiifilisest optimeeringust (vt ptk 3.3.1).

```
1  public final class GeneratedClassesManager {
2      @InvocationTarget("GeneratedClassesManager#getTableName")
3      public static String getTableName(Class<?> clz) {
4          final String name = clz.getName();
5          switch (name) {
6              case "com.siimkinks.sqlitemagic.sample.model.Book":
7                  return "book";
8              case "com.siimkinks.sqlitemagic.sample.model.Author":
9                  return "author";
10             default:
11                 throw new RuntimeException("Class " + name +
12                     " is not annotated with @Table");
13         }
14     }
15 }
16
17 // before
18 public final class SqlUtil {
19     @Invokes("GeneratedClassesManager#getTableName")
20     public static String getTableName(Class<?> tableClass) {
21         // filled with magic
22         throw new RuntimeException(ERROR_PROCESSOR_DID_NOT_RUN);
23     }
24 }
25
26 // after
27 public final class SqlUtil {
28     @Invokes("GeneratedClassesManager#getTableName")
29     public static String getTableName(Class<?> var0) {
30         return GeneratedClassesManager.getTableName(var0);
31     }
32 }
```

Koodinäide 39: SqliteMagic teegis koodi sidumine baitkoodi muudatustega.

Loodud süsteem on mõeldud teegisiseseks kasutamiseks, mistõttu realiseeritud funktsionaalsused piirduvad ainult vajalikega. Süsteemi rakendamisel kasutatakse põhimõtet, kus

[17] Selguse huvides kasutab teek võtmeid stiilis "*klassiNimi#meetodiNimi*".

väljakutsuv meetod omab väljakutsutava meetodiga enamasti sama signatuuri, kuid see pole kohustuslik. Baitkoodi muudatusi tehes jälgitakse väljakutsuva meetodi tagastustüüpi – *void*-i korral jäetakse väljakutsutava meetodi tulemus tagastamata. Lisaks antakse väljakutsuva meetodi parameetrid sisendiks väljakutsutavale meetodile. Viimast käitumist saab muuta annotatsiooni *@Invokes* parameetriga *useThisAsOnlyParam*, mille tõese väärtuse korral antakse väljakutsutavale meetodile ainsaks parameetriks *this* võtmesõna.

Baitkoodi muudatuste läbiviimiseks sobiks iga teek, mis toetab kompileerimisaegset töötlust – AspectJ, ASM, DexMaker, Javassist jne. Kuna kirjeldatud süsteem pole väga keerulise funktsionaalsusega, siis baitkoodi muudatusi teostava teegi valikul peeti oluliseks järgnevaid aspekte – lihtsat kasutust, head dokumenteeritust, projekti aktiivset haldamist ja lihtsat integreerimist rakenduse ehitusprotsessi. Valituks osutus Javassist, kuna see vastas kõigile nõudmistele, selle funktsionaalsuste pagas on kõige sobivam ülesande teostamiseks ning selle kasutamine tundus kõige lihtsam – põhiargumendiks oli võimalus muuta baitkoodi omamata sügavaid teadmisi sellest ning võimalus kirjutada lahtise teksitiga Java koodi, mis hiljem kompileeritakse muudetavasse klassi.

Baitkoodi transformaatori töölesaamiseks tuli kirjutada Gradle plugin[18], mis modifitseerib kompileerimisprotsessi, seadistades transformaatori töölepaneku peale Java koodi kompileerimist. Kuid kirjeldatust üksi veel ei piisa teegi ja genereeritud koodi ühendamiseks. Jättes kõik nagu mainitud jookseks transformaator ainult rakenduse ja annotatsiooniprotsessori genereeritud koodil, mis enamasti sisaldab ainult väljakutsumissihtmärke (*@InvocationTarget* annotatsiooniga meetodid). Kõige vajaliku ühendamiseks, tuleb enne transformeerimist, rakenduse sõltuvusena lisatud SqliteMagic täitmisaegses *jar* failis sisalduvad kompileeritud failid kopeerida Java kompileerimisülesande tulemuskausta. Tänu kirjeldatule on baitkoodi transformeerimisülesande lähtefailideks nii rakenduse ja annotatsiooniprotsessori genereeritud kompileeritud failid, kui ka teegi täitmisaegse koodi kompileeritud failid.

3.2.2 Rakenduse ja genereeritud koodi ühendamine

Teiseks loodi süsteem, kus *@Table* annotatsiooniga klassidele genereeritakse andmebaasioperatsioonide meetodid. Loodud süsteemi võib vaadelda, kui laiendusmeetodite lisamist Java keelde väga primitiivsel ja piiratud kujul.

Java lähtekoodi kompileerimine toimub laias plaanis kolmes staadiumis [16]:

1. Kõik lähtefailid kogutakse ning parsitakse süntaksipuudeks ehk AST-deks (*abstract syntax tree*).
2. Käivitatakse kõik defineeritud annotatsiooniprotsessorid. Uute failide genereerimisel korratakse esimest sammu ning käivitatakse uuesti annotatsiooniprotsessorid. Kirjeldatu toimub kuni ühtegi uut faili ei looda.
3. Loodud süntaksipuid analüüsitakse ning vigade puudumisel genereeritakse nende põhjal baitkood. Analüüsi käigus võib ilmneda viiteid lisaklassidele, mille korral kompilaator otsib nende lähte- või kompileeritudkoodi asukohti. Lähtekoodi leidmisel need failid kompileeritakse, kuid neile ei rakendata annotatsioonide töötlust.

[18]Teostuse idee ja baaskood pärineb Morpheus projektist[23].

Kompileerimise viimases staadiumis enne baitkoodi genereerimist otsitakse katkiseid viiteid (näiteks puuduvad klassid ja meetodid), teostatakse tüübikustutamine (*type erasure*), eemaldatakse süntaktiline liiasus jne. Kirjeldatust tulenevalt ei ole näiteks baitkoodi muudatustega võimalik genereerida meetodeid, mida saaks rakenduse koodist tavapärasel viisil välja kutsuda – puuduvate viidete korral baitkoodi ei genereerita, mistõttu ei ole asja, mida muuta.

Ainsaks autorile teadaolevaks viisiks rakenduse koodist väljakutsutavate meetodite genereerimiseks on läbi AST muudatuste. Viimase teostamiseks on üldjoontes kaks võimalust – kasutades Groovy kompileerimisaegseid AST transformatsioone[13, ptk *Compile-time metaprogramming*] või Lombok teeki[20]. Groovy kasutamine ei sobi väga olulisel põhjusel – selle kasutamine nõuaks, et ka rakendus seda kasutaks. Kirjeldatud nõue suurendaks APK (*Android application package*) meetodite arvu märkimisväärselt[19] ning teadagi on Androidis meetodite arv limiteeritud (vt ptk 2.9). Tulemusena peaks teegi kasutaja lisama rakendusse väga suure sõltuvuse, mida potentsiaalselt üldse ei kasutata. Parimaks valikuks osutus Lombok, kuna see on ennast tõestanud, suhteliselt kaua eksisteerinud, kuid aktiivses arenduse olev teek, millel on palju kasutajaid. Lisaks pole Lomboki baasfunktsionaalsusel rakenduse tööaja sõltuvusi – kogu töö tehakse kompileerimise ajal.

Lombok ühendub Java kompileerimisprotsessi olles annotatsiooniprotsessor. Erinevalt tavalistest protsessist, kus lubatud on ainult failide genereerimine, kuid mitte modifitseerimine, kasutab Lombok n-ö „tagaukse" APIsid, et modifitseerida esimeses kompileerimisfaasis loodud süntaksipuid. Tänu AST muudatustele teises kompileerimise faasis, on näiteks klassile lisatud meetodid nähtavad kolmandale faasile, mistõttu on rakenduse koodist võimalik välja kutsuda sünteetiliselt lisatud meetodeid. Loodud teek ühendub Lomboki teeki, kui selle laiendus – defineeritakse käsitlejaid (*handlers*), mis ühenduvad annotatsioonide vahendusel Lomboki töötlusprotsessi.

```
1  // before
2  @Table(persistAll = true)
3  public final class Author {
4      ...
5      @Invokes(
6          value = "SqliteMagic_Author_Handler.InsertBuilder#create",
7          useThisAsOnlyParam = true)
8      public final InsertBuilder insert() {
9          // filled with magic
10         throw new RuntimeException(ERROR_PROCESSOR_DID_NOT_RUN);
11     }
12     ...
13 }
14 // after
15 @Table(persistAll = true)
16 public final class Author {
17     ...
18     @Invokes(
19         value = "SqliteMagic_Author_Handler.InsertBuilder#create",
20         useThisAsOnlyParam = true)
21     public final InsertBuilder insert() {
22         return InsertBuilder.create(this);
23     }
24     ...
25 }
```

Koodinäide 40: SqliteMagic teegis Lomboki abiga genereeritud meetod.

[19] Groovy Androidi Gradle plugina lehel esitatud probleemipileti andmetel nõuab Groovy kasutamine umbes 28000 meetodit[12].

Lomboki vahendusel genereeritakse vajalikele klassidele andmebaasioperatsioonide meetodid. Kuna eelnevalt loodi teegi ja genereeritud koodi ühendamiseks süsteem (vt 3.2.1), siis töö kokkuhoiu mõttes kasutati seda ka genereeritud meetodite ja annotatsiooniprotsessoriga genereeritud koodi ühendamiseks. Sellest tulenevalt on genereeritud meetodil *@Invokes* annotatsioon (vt Koodinäide 40, read 5 - 7) ning kogu vajalik signatuur (rida 8), kuid mille sisu asendatakse baitkoodi muudatustega (read 10 ja 22).

Lisaks meetoditele genereeritakse andmebaasi tabeli veergude nimedele (vt Koodinäide 41, read 4 - 5) ja tabeli nimele (rida 3) vastavad *String* konstandid, et neile oleks kerge viidata näiteks päringute sooritamisel. Muudetavates klassides genereeritakse puuduva primaarvõtme välja puhul *long* tüüpi „id" nimega väli (read 6 - 8) ning selle väärtuse juurdepääsuks sama nimega meetod (rida 10).

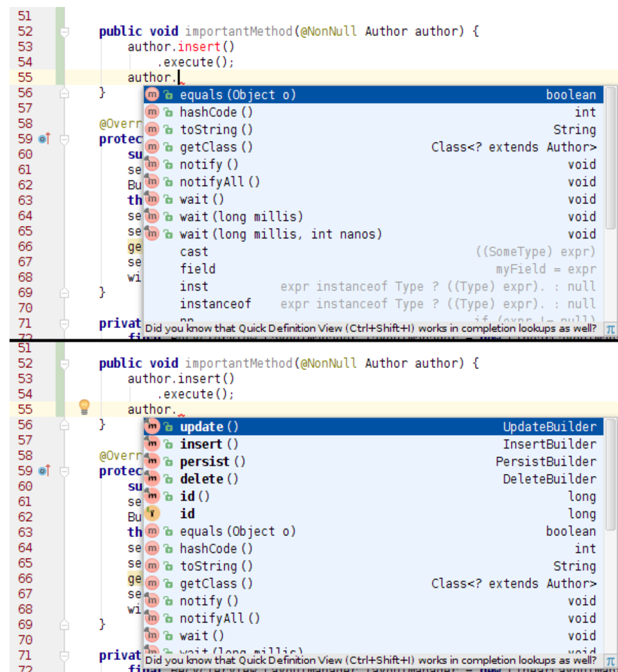
```
1  @Table(persistAll = true)
2  public final class Author {
3      public static final String TABLE = "author";
4      public static final String C_FIRST_NAME = "author.first_name";
5      public static final String C_LAST_NAME = "author.last_name";
6      @Id
7      @Column("_id")
8      long id;
9
10     public final long id() {
11         return this.id;
12     }
13     ...
14 }
```

Koodinäide 41: SqliteMagic teegis Lomboki abiga genereeritud väljad.

Eksperimentaalse funktsionaalsusena toetatakse ka privaatse nähtavusega andmemudeli väljade ja konstruktori defineerimist. Viimane saavutatakse samuti Lomboki vahendusel muutes AST-s vastavaid väärtusi. Kuna aga kirjeldatud funktsionaalsus on küsitava vajadusega ning turvariske tekitav, siis selle võimaldamine vajab põhjalikumat uurimist ning kogukonna tagasisidet.

3.2.3 Lahenduse töölesaamine Android Studios

Siiani kirjeldatud baitkoodi ja AST muudatused ühendavad kogu genereeritud koodi n-ö kapotialuselt. Kõik töötab, kui rakendus käsurealt kompileerida, kuid IDE värvib genereeritud meetodite väljakutsed punaseks viidates meetodite puudumisele (vt Joonis 2). Kirjeldatud probleemi lahendamiseks tuli kirjutada IntelliJ plugin, mis põhimõtteliselt annab IDE-le teada, mis meetodeid ja välju genereeritakse. Kõige tulemusena pole andmeklassi lähtekoodi vaatamata võimalik aru saada, et meetodid on sünteetiliselt lisatud (vt Joonis 2).



Joonis 2: SqliteMagic teegi genereeritud meetodid vaadatuna Android Studios enne ja peale SqliteMagic IntelliJ plugina paigaldamist.

3.3 Androidispetsiifilised optimeeringud

Kuigi Androidi rakendused kirjutatakse Javas, ei saa neid kindlasti kohelda, kui tavalisi Java rakendusi. Esiteks jooksevad Androidi rakendused kordades aeglasematel, vähema mäluga ja teise arhitektuuriga riistvaral, kui serveri või töölaua rakendused. Teiseks, olenemata, et rakendusi kirjutatakse Javas ei jookse need Java virtuaalmasinal, vaid hoopis Dalvikul[20] või ART-il[21][3]. Viimasest tuleneval täidetakse rakenduse tööajal Java baitkoodi põhjal loodud Dalviku baitkoodi. Kolmandaks põhineb Androidis kasutatav Java modifitseeritud Apache Harmony projektil[2]. Viimasest tulenevalt on Androidis Java standardteekide kogum suuresti vähendatud ning optimeeritud mobiilsetes seadmetes kasutamiseks.

Tulenevalt kirjeldatud keskkonnast on soovituslik Androidi rakendusi arendades silmas pidada mitmeid optimeeringuid, mis tava Java rakendustes ei kehti, ei oma erilist jõudluse võitu või millest on lihtsalt mööda vaadatud [24]. Siiski ignoreeritakse rakenduste arenduses tihti mitmeid optimeeringuid, kuna need vähendaks koodist arusaamist, nende rakendamine võtaks rohkem aega, kui esialgse tulemuse tarbeks vaja või ei olda lihtsalt piisavalt pädevad või asjast teadlikud. Samas, teegi kirjutamisel on iga optimeeringu rakendamine äärmiselt oluline, sest kunagi ei saa ette näha kõiki situatsioone, kus seda võidakse kasutada. Lisaks, kuna teek töötab rakenduse osana, siis võiks öelda, et parim teek on märkamatu teek. Rakenduse kirjutamisel soovitakse keskenduda selle äriloogikale ja funktsionaalsustele, mitte teekide kiiruse optimeerimisele või halvast mälukasutusest tingitud mäluprobleemidele.

[20] Kuni Android 4.4-ni.
[21] Alates Android 5.0-st.

3.3.1 SqliteMagic teegis rakendatud optimeeringud

Järgnevalt on loetletud mõned märkimisväärsed Androidi spetsiifilised optimeeringud, mida loodud teegi arendamisel silmas peeti. Kuigi enamus nendest on n-ö mikrooptimisatsioonid, mis üksi kasutades erilist jõudlusvõitu ei anna, siis nende massilisel kasutamisel väiksed kiirusvõidud summeeruvad andes kokkuvõttes siiski märkimisväärseid tulemusi (vt ptk 4.1 ja ptk 4.2).

Kõikjal, kus võimalik, eelistatakse staatilisi meetodeid virtuaalsetele. Esiteks, staatiliste meetodite väljakutsed on umbes 15%-20% kiiremad [24, ptk *Prefer Static Over Virtual*]. Teiseks, lihtsustab kirjeldatu koodi genereerimist ning selle kasutamist – pole vaja luua uusi objekte, et vajalikke meetodeid välja kutsuda ning hallata objektide taaskasutamist jne.

Mälu kokkuhoidmiseks ning tihedate prahikoristussündmuste vältimiseks taaskasutatakse objekte nii palju kui võimalik [24, ptk *Avoid Creating Unnecessary Objects*]. Lisaks parandab nimetatu ka operatsioonide kiirust – kuigi mälu eraldamine on võrdlemisi odav, siis selle allokeerimine on alati kallim, kui allokeerimata jätmine. Veelenam, mäluprahi koristamine nõuab teatud aega ning mida vähem seda tegema peab, seda rohkem hoitakse aega kokku. Kirjeldatud optimisatsioonil on otsene mõju ka rakenduse kasutajakogemusele – tihedad mälukoristused tekitavad väikseid „jõnkse” rakenduse töös ning mida vähem neid on, seda sujuvam on rakenduse töö.

Meetodite vahendusel või klassiväljadest saadud korduvalt kasutatavad objektiviidad puhverdatakse lokaalsetes muutujates. Iga virtuaalse meetodi ja väljamuutuja väljakutsel on mingi kulu. Näiteks virtuaalse meetodi väljakutsumiseks peab virtuaalmasin teadma, kuhu mäluaadressile hüpata. Selleks käiakse kogu klassi hierarhia läbi ning luuakse meetodite tabel, mis põhimõtteliselt sisaldab iga meetodi nime ja selle asukohta mälus ehk mäluaadressi. Et leida tabelist mäluaadress, kuhu hüpata, tuleb teha tabeli otsing (*lookup*), mis põhimõtteliselt kujutab endast meetodi nimel hashimisfunktsiooni rakendamist ning tulemuse põhjal väärtuse leidmist. Kuigi kirjeldatud funktsiooni rakendamine on väga kiire, on sel mingi kulu. Viimast on mõistlik maksta ainult ühe korra, eriti kui väljakutse on näiteks iteratsiooni sees, mida teostatakse mitutuhat korda. Kuigi JIT (*Just-In-Time*) ja AOT (*Ahead-Of-Time*) kompileerijad[3] aitavad leevendada kirjeldatud kulu, pole ka need probleemi täielik lahendus. Kui näiteks JIT asendab meetodi väljakutse otsese väljamuutuja väljakutsega, tuleb viimase leidmiseks ikkagi teha mäluotsing (*memory lookup*), mis pole samuti tasuta.

Ollakse teadlikud Android SDK ja Java standardteegi klasside „hingeelust” ning kasutatakse neid võimalikult optimaalselt. Näiteks andmete pärimisel, kui on teada, et n objekti tuleb tagastada *ArrayList* andmetüübina, siis allokeeritakse uus andmestruktuur konstruktoriga, kus sisendparameetriks on n. Kirjeldatud käitumine elimineerib vajaduse *ArrayList*-i sees oleva primitiivse massiivi pideva uuesti allokeerimise ning viitade kopeerimise uude massiivi, kuna õige suurusega massiiv allokeeritakse juba objekti loomisel. Sama tehakse näiteks *Stringide* allokeerimisega *StringBufferi* vahendusel. Lisaks välditakse mugavusmeetodite kasutamist ning kutsutakse otse juurmeetodit operatsiooni sooritamiseks. Kirjeldatu väldib asjatuid meetodite väljakutseid.

Välditakse *autoboxingut* ning eelistatakse primitiivseid andmetüüpe nende objektidest vastanditele. Iga primitiivse andmetüübi muutmine selle objektist vastandiks tekitab uue

objekti. Kirjeldatu loob asjatult uusi objekte, mis omakorda raiskab mälu ning kutsub esile asjatuid mälukoristusi.

Java standardteegis olevad andmestruktuurid[22] on optimeeritud kiiretele operatsioonidele suurtel andmekogustel, kuid seejuures raiskavad ka palju mälu. Kirjeldatud probleemi adresseerimiseks on Androidis mälukasutust optimeerivad kollektsioonid[23]. Viimased jäävad suurte andmekoguste juures kiiruselt Java vastavatele andmestruktuuridele märgatavalt alla, kuid mõistlike suuruste juures on samaväärsed ning kohati kiiremadki. Kirjeldatust tulenevalt kasutataksegi kõikjal, kus võimalik ja mõistlik mälu säästmiseks Java andmestruktuuride asemel Android vastandeid.

Kuna päringute kiirus on loodud teegis olulise tähtsusega otsustati realiseerida optimeeritum versioon Androidi *SQLiteCursor* klassist[24]. Kuna *SQLiteCursor* on loodud kasutamiseks kõiksugustes kasutusjuhtudes, tehakse sisemiselt palju raamatupidamist nagu veerunimede ja neile vastavate indeksite hoidmine *HashMap* objektis, API õige kasutamise kontrollimine jne. Teades, kuidas andmeid päritakse, saab lõviosa nimetatud arvepidamisest ära jätta – veerunimede ja nende indeksite üle pole vaja järge pidada, kuna päringu parsimise hetkel on teada, mis positsioonil päritavad veerud asuvad; pole vaja kontrollida API õiget kasutust, kuna objekti kasutatakse vaid teegisiseselt jne. See omakorda tagab palju kiirema andmete kättesaamise (vt ptk 4.2.4).

[22]HashMap, HashSet.
[23]ArrayMap, ArraySet, Sparse perekond jne.
[24]Optimeeringu idee pärineb greenDAO teegist.

4 Loodud teegi jõudluse analüüs võrdluses alternatiividega

Analüüsi eesmärgiks on välja selgitada, kas eelnevates peatükkides loetletud teoreetilised eeldused ja lahendused peavad paika ka reaalsuses. Selleks võrreldakse omavahel peatükis 1 loetletud olemasolevaid lahendusi ja antud töö tulemusena valminud SqliteMagic teeki. Mõõtmises vaadeldakse mobiilseadmete seisukohast olulisi aspekte.

Esiteks mõõdetakse operatsioonide kiirust, mis mõjutab protsessori tsüklite kaudu ka aku kestvust. Teiseks mõõdetakse efektiivset mälu kasutust. Kuna mobiilseadmetel on mälu väga piiratud koguses, siis mida enam võtab teek mälu, seda vähem jääb seda rakenduse kasutada. Ebavajalikult tekitatud või raisatult kasutatud objektid sunnivad Androidi süsteemi tihedamalt mälu koristama (*garbage collection*), mille tulemusena kannatab kiirus, kuid ka kasutuskogemus läbi väikeste „jõnksude". Lisaks, sarnaselt kiirusega on see kaudses seoses aku kasutusega [24].

Tähtis on märkida, et kõikide jõudlustestide läbiviimiseks kasutati teekides seadistusi, kus andmete puhverdamine mälus on deaktiveeritud. Kirjeldatud käitumine valiti esiteks võrdsete testitulemuste saamiseks, kuna kõik teegid ei toeta andmete puhverdamist. Teiseks, vastasel korral ei väljendaks testide tulemused enam teegi efektiivsust andmete pärimisel otse SQLite andmebaasist, mida üritataksegi mõõta.

Testimiseks kasutati tehasesätestatud ja lennurežiimile seatud Samsung S2 (GT-I9100) nutitelefoni järgnevate andmetega:

- CPU: Arm Cortex A9 Dual Core 1200MHz
- RAM: 1GB
- OS: Android 4.4.4 (API 19)
- CPUFreq Governor: performance
- I/O scheduler – cfq
- SQLite: 3.7.11

Testides kasutati teekide viimaseid stabiilseid versioone[25]:

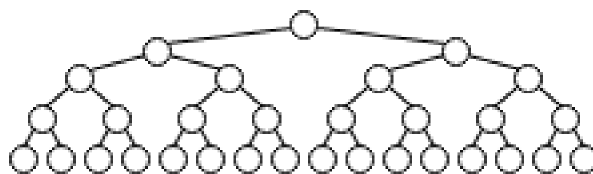
- Sugar ORM: 1.4
- greenDAO: 2.1.0
- DBFlow: 2.2.1
- SqliteMagic: 0.3.1-snapshot

Testitakse järgnevaid andmetüüpe:

- Märgijadadega – andmetüüp sisaldab 12 täidetud *String* välja.
- Arvudega – andmetüüp sisaldab 12 täidetud numbrilist välja, milleks on *short*, *int*, *long*, *float*, *double*, *boolean* ning nende objektide vastandid.

[25] 30. novembri 2015 seisuga.

- Segaandmetega – andmetüüp sisaldab 12 välja: neli *Stringi*, millest 2 on täitmata ning tüüpide *long* ja *double* kohta neli välja, millest kaks on täidetud primitiivset ning üks täidetud ja üks täitmata objekti vastandit.
- Suur segaandmetega – andmetüüp sisaldab 44 välja: 12 *Stringi*, millest 6 on täitmata ning iga *long*, *int*, *double*, *float* tüübi kohta 8 välja, millest neli on täidetud primitiivset ning kaks täidetud ja kaks täitmata objekti vastandit.
- Üheastmelise sõltuvusega – andmetüüp sisaldab kahte täidetud segaandmetega välja.
- Neljaastmelise sõltuvusega – andmetüüp kujutab endast täisbinaarpuud sügavusega 4 (vt Joonis 3), mille lehtedeks on segaandmetega andmetüübid.



Joonis 3: Neljaastmelise sõltuvusega andmetüüp.

4.1 Kiirus

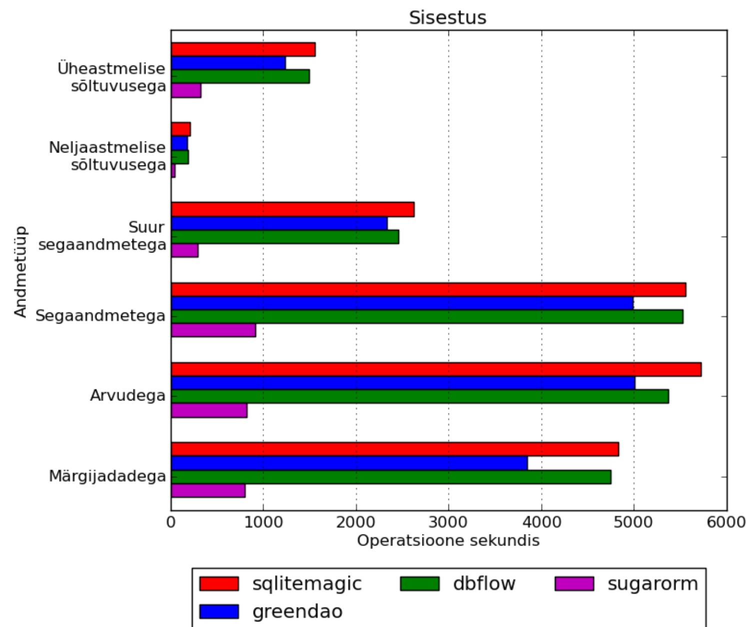
Järgnevalt on esitatud kiirustestide tulemused. Kuna mobiilseadmetes võib sama käsu käivitamiseks kuluv aeg üpriski palju varieeruda, siis võimalikult adekvaatsete tulemuste saamiseks jooksutati kõikide teekide testkomplekte meelevaldsed 24 korda.

4.1.1 Sisestus

Testi käigus sisestati ühe transaktsiooniga 400 kompleksandmetüüpide ja 25000 primitiivandmetüüpide kirjet tühja tabelisse. Tulemustest on näha (vt Joonis 4), et SqliteMagic töötab kõikide andmetüüpide puhul kiiremini, kui konkurendid. Ainult DBFlow sisestuste arv sekundis on peaaegu kõikide andmetüüpide puhul suhteliselt samaväärne. Lisaks on näha, et *reflectionit* kasutav Sugar ORM on umbes 6 korda aeglasem.

Uurides teekide lähtekoodi ilmneb olukord, kus teegid SqliteMagic, greenDAO ja DBFlow kasutavad andmete sisestamiseks eelkompileeritud SQLite lauseid, kuid Sugar ORM kompileerib igal sisestusel uue INSERT lause. Seega tuleneb viimase teegi kiiruselt teistele allajäämine ka vähesest optimeeritusest.

Teekide lähtekoodidest ilmneb ka, et kolme kiirema salvestusstrateegia on üpriski identne, kuid sellest hoolimata on salvestuskiirused rohkemal või vähemal määral erinevad. Kiiruserinevusest arusaamiseks kasutati meetodite jälgimise (*traceview*) tööriista ning mõõdeti 15000 segaandmetega objekti sisestamist. Joonis 17 (vt Lisa A) kuvab meetodite tööaja mõõtmistulemusi.



Joonis 4: Andmete sisestus.

Meetodite mõõtmistulemustes on näha, et igas teegis algab sisestusoperatsioon testi meetodiga *MixedDataHandler.bulkInsert* ning operatsiooni põhitöö tehakse ühes kindlas meetodis. SqliteMagic teegis kulub põhilist tööd tegeva meetodini *BulkInsertBuilder.execute* jõudmiseks 0.1%; DBFlow teegis meetodini *SqlUtils.insert* jõudmiseks 7.3% ning greenDAO teegis meetodini *AbstractDao.executeInsertInTx* jõudmiseks 0.9% protsessori ajast. DBFlows jõutakse meetodini aeglasemalt esiteks tänu mitmetele sisemistele ümbersuunamistele, kuid peamiselt tänu meetodile *BaseModel.getModelAdapter*, mis otsib andmeobjekti kohta genereeritud klassi *reflectioni* vahendusel või puhverdatuna *HashMap* objektist.

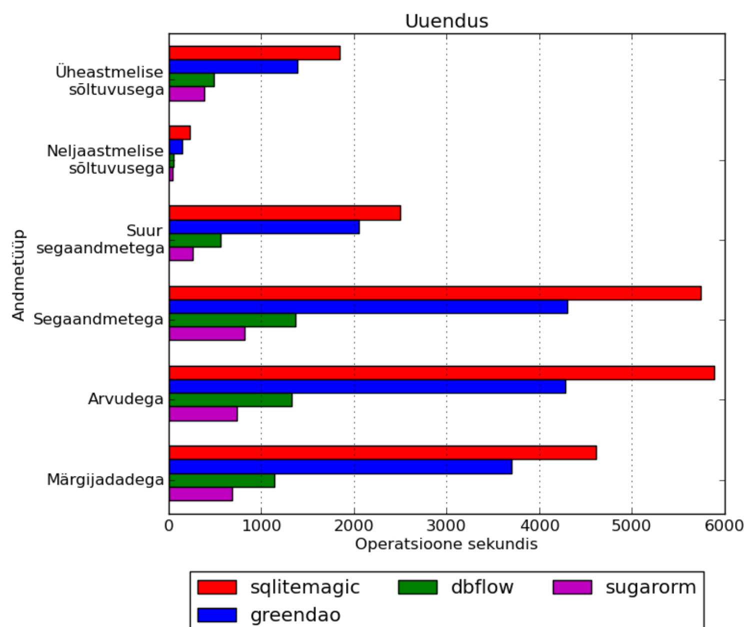
Peale salvestamislausele vastava objekti ettevalmistamist toimub reaalne andmete salvestamine kõikides teekides meetodis *SQLiteStatement.executeInsert*. Kuna kõik teegid kasutavad enamvähem sama SQLi andmete salvestamiseks sõltub operatsiooni täitmisaeg objekti ettevalmistamiskiirusest ning muude järjepidamismeetodite rohkusest ja nende täitmiskiirusest. Kõige vähem kulutatakse kirjeldatu peale aega SqliteMagicus – ettevalmistamiseks 19.6% ja järjepidamisele 5%. Optimaalsuselt järgmine on DBFlow, kus ettevalmistamine võtab 22.4% ja järjepidamine 4.3% protsessori ajast. Kõige rohkem aega raiskab greenDAO, kus ettevalmistamiseks kulub 22.3% ja järjepidamiseks 10.4%. Kirjeldatud järjekorda väljendavad ka kiirustestide tulemused.

SqliteMagicu salvestamislause ettevalmistuskiirus tuleneb selle puhastamisest ühe meetodiga *clearBindings* ning andmeväärtuste kättesaamisest juurdepääsumeetoditeta. DBFlow teostab kirjeldatud operatsiooni aeglasemalt, kuna puhastust ei teostata mitte korraga vaid meetodiga *bindNull* kohtades, kus seda vaja on. Tulemuseks on rohkem meetodite väljakutseid, mis nagu ka mõõtmistulemustest näha saab, annavad suure koguse peale märgatava kiiruskaotuse. Kirjeldatust tulenevalt töötab DBFlow seda kiiremini, mida vähem on nullitavaid välju. Viimast näitavad ka kiirustestide tulemused - näiteks üheastmelise sõltuvusega andmeobjektidel pole ühtegi nullitavat välja ning teekide SqliteMagic ja DBFlow kiirused on põhimõtteliselt võrdsed. greenDAO kasutab puhastamiseks sama

meetodit, mis SqliteMagic, kuid kaotab ajas tänu andmeväärtuste pärimisele läbi juurdepääsumeetodite.

4.1.2 Andmete uuendamine

Testi käigus uuendati ühe transaktsiooniga 400 kompleksandmetüüpide ja 25000 primitiivandmetüüpide kirjet. Tulemustest on näha (vt Joonis 5), et SqliteMagic töötab kõikide andmetüüpide puhul märgatavalt kiiremini, kui konkurendid. GreenDAO on küll teistest konkurentidest kordades kiirem, kuid jääb siiski SqliteMagicule märgatavalt alla.



Joonis 5: Andmete uuendus.

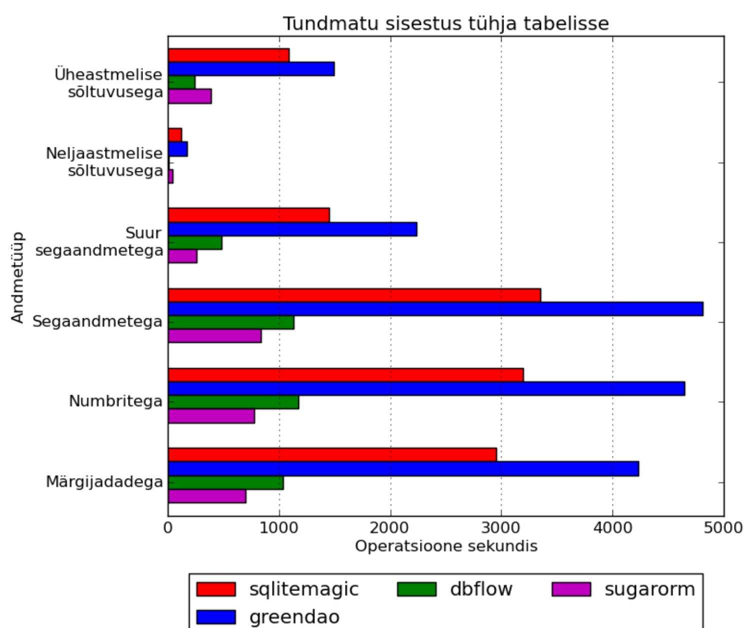
Teekide lähtekoodi uurides on näha, et kaks kõige kiiremat kasutavad, sarnaselt sisestamisele, uuendamiseks eelkompileeritud SQLite lauseid, mis tagab kordades kiirema teostusaja. Kaks aeglasemat teeki teostavad uuenduse väärtustades *ContentValues* objekti ning käivitades uuendamisoperatsiooni, mille tulemusena loodud lause kompileeritakse ja täidetakse. Testide tulemusest on aga näha, et DBFlow on Sugar ORM-st peaaegu kõikjal kaks korda kiirem. Seega enamuse *reflectioni* kõrvaldamine tagab ka reaalsuses märgatavaid kiirusvõite.

Kuna SqliteMagic ja greenDAO kasutavad umbes sama strateegiat andmete uuendamiseks kasutati kiiruserinevustest arusaamiseks meetoride jälgimise tööriista. Mõõdeti 15000 segaandmetega objekti uuendamist. Joonis 16 (vt Lisa A) kuvab meetodite tööaja mõõtmistulemusi, millest on näha, et kiiruserinevus pärineb täpselt samadest põhjustest, mis andmete sisestamiselgi (vt ptk 4.1.1).

4.1.3 Tundmatu sisestus

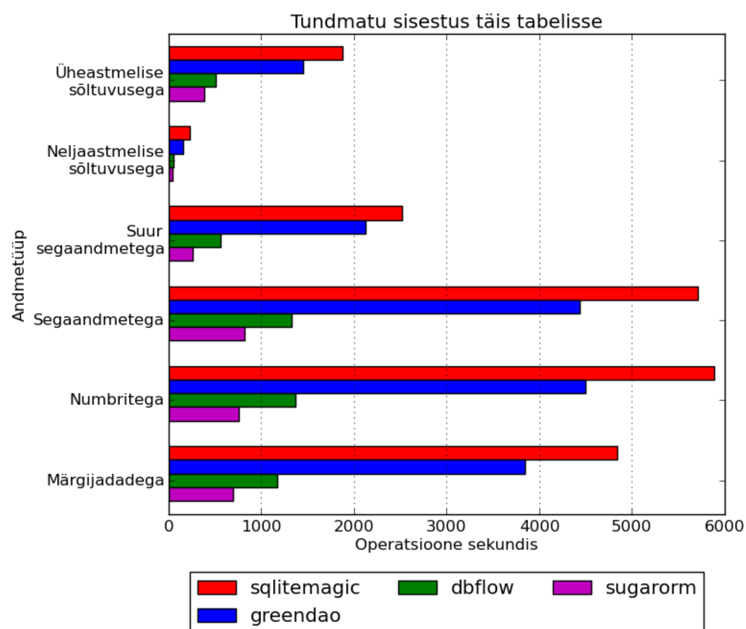
Tundmatuks sisestuseks nimetatakse olukorda, kus andmeid tahetakse salvestada, kuid pole teada, kas neid tuleb sisestada või uuendada. Enamasti on kõikides teekides kirjeldatu toetamiseks eraldi mugavusmeetod. Testi käigus mõõdeti mõlemat käitumist – sisestust, kui andmebaas on tühi (vt Joonis 6) ning uuendust, kui andmed on juba tabelis (vt Joonis 7). Mõlema puhul salvestati ühe transaktsiooniga 400 kompleksandmetüüpide ja 25000 primitiivandmetüüpide kirjet.

Kiirustesti tulemustest on näha, et teegid SqliteMagic ja greenDAO on mõlemal eeltingimusel teistest kordades kiiremad. Lähtekoodide uurimisel selgub, et kiirus saavutatakse eelkompileeritud SQLite lausetega. Lisaks võrreldes sarnase salvestusstrateegiatega teekide Sugar ORM ja DBFlow tulemusi, ilmneb jällegi *reflectioni* kasutamisest tingitud märkimisväärne kiirusekaotus.



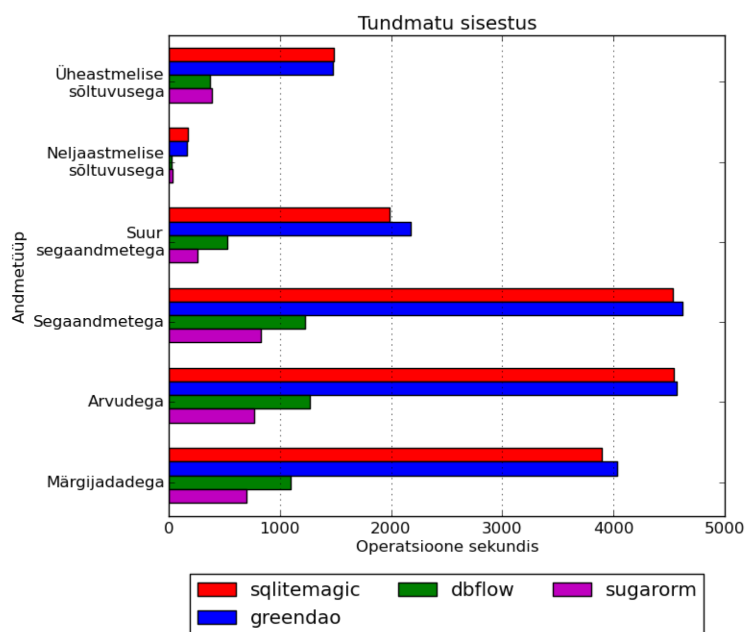
Joonis 6: Andmete tundmatu salvestamine tühja tabelisse.

Jälgides kahe kiirema teegi operatsiooni kiirust tühja ja täidetud tabeli korral ilmneb huvitav olukord, kus ühel korral on greenDAO märgatavalt kiirem ning teisel korral SqliteMagic peaaegu samaväärselt kiirem. Lähtekoodis on näha ka salvestusstrateegiate erinevus – greenDAO kasutab salvestamiseks INSERT OR REPLACE eelkompileeritud lauset samas, kui SqliteMagic üritab kõigepealt eelkompileeritud UPDATE lausega andmeid uuendada ja selle ebaõnnestumisel need sisestada eelkompileeritud INSERT lausega. Kirjeldatust tulenevalt jääb vastavalt tabeli täidetusele mõlemas teegis alati üks operatsioon täitmata, mida väljendavad ka testide tulemused.



Joonis 7: Andmete tundmatu salvestamine täis tabelisse.

Tundmatu sisestamise operatsiooni kokkuvõtliku ülevaate saamiseks on kahe eeltingimuse tulemused koondatud ühte graafikusse (vt Joonis 8). Tulemustest on näha, et kuigi greenDAO on enamuste andmetüüpide puhul kiirem, jääb SqliteMagic vaid natuke maha, mistõttu võib need üldjoontes võrdväärseteks lugeda.



Joonis 8: Andmete salvestamine, kui pole teda, kas tuleb uuendada või lisada.

Kui on ette teada, et teatud andmed pole andmebaasis, kasutatakse optimeerituse mõttes *insert* operatsiooni. Seega leiab tundmatu sisestuse operatsioon peamiselt kasutust olukordades, kus tahetakse andmeid uuendada, kuid ei saa täiesti kindel olla, kas eelnevalt

on sama primaarvõtmega andmeid sisestatud. Üheks näiteolukorraks võib olla rakenduses andmete uuendamise nupu vajutamine. Seetõttu, võib kogemuse põhjal väita, et andmete uuendamist tuleb nimetatud operatsiooni kasutamise puhul palju rohkem ette, kui sisestamist. Kirjeldatust tulenevalt eelistatigi SqliteMagic teegi arendamisel tundmatu salvestuse operatsiooni puhul uuendamise kiirust salvestamise kiirusele, mida näitavad ka kiirustestide tulemused.

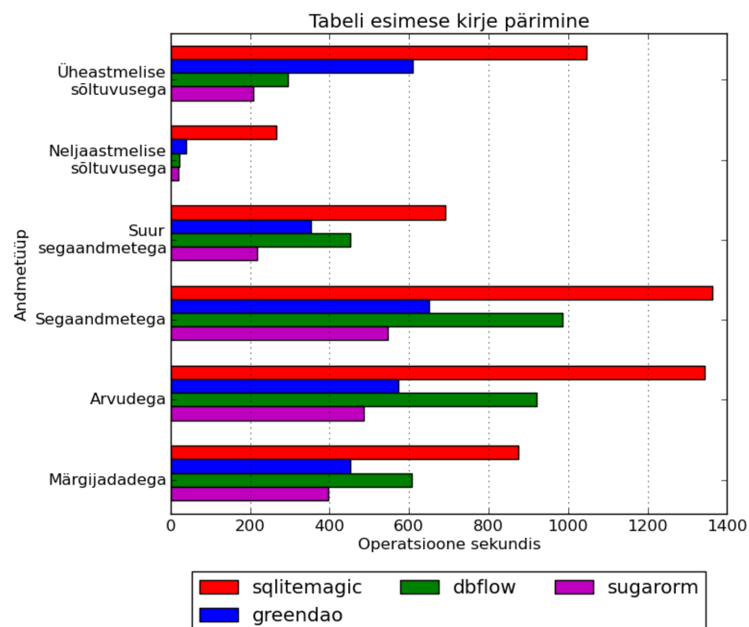
4.1.4 Pärimine

Teekide pärimiskiirust mõõdeti kahes testis. Esiteks päriti 400 kirjega kompleksandmetüüpide ja 25000 kirjega primitiivandmetüüpide tabeli esimest kirjet 1000 korda (vt Joonis 9). Teiseks päriti 400 kirjega kompleksandmetüüpide ja 25000 kirjega primitiivandmetüüpide tabeli kõiki andmeid 10 korda (vt Joonis 10).

Tähtis on märkida, et kõiki andmeid pärivas testis, operatsiooni all mõistetakse ühe kirje kättesaamist andmebaasist ja selle muutmist Java andmeobjektiks. Viimases arvutatakse operatsioonide arv sekundis järgneva valemiga:

$$T = \left(\frac{\sum_{i=1}^{n} t_i}{10^9 \sum_{i=1}^{n} r_i d_i} \right)^{-1}$$

Kus T on operatsioonide arv sekundis; n on testi läbiviimiste arv; t on kõiki andmeid päriva operatsiooni teostusaeg nanosekundites; r on ühes testis kõiki andmeid päriva operatsiooni teostamiste arv; d on päritud kirjete arv.



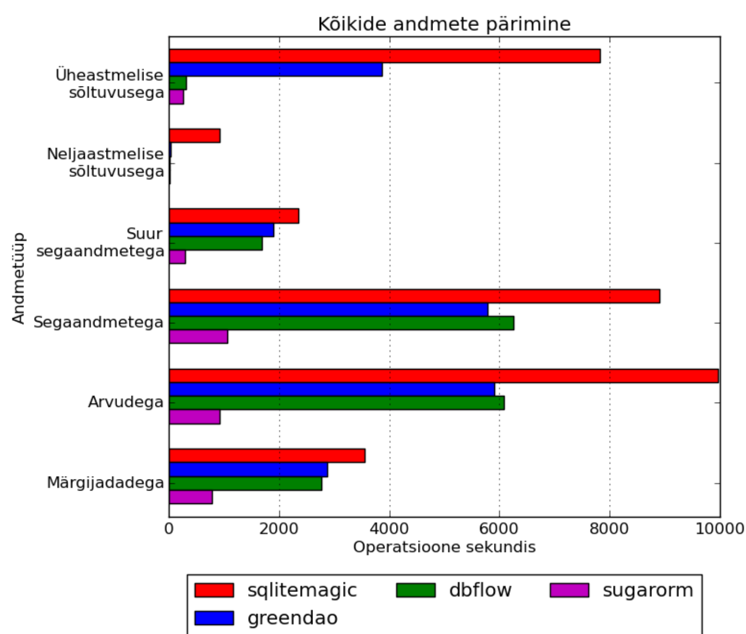
Joonis 9: Tabeli esimese kirje pärimine.

Esimese kirje pärimise kiirus väljendab eelkõige teekide suutlikkust väikese operatsiooni

puhul keskenduda läbiviidavale operatsioonile ja vähem aega kulutada kõikvõimalikele muudele sisemistele arvepidamistele ning koodi orkestreerimistele.

Tabeli esimest kirjet päriva testi tulemustest (vt Joonis 9) on näha, et SqliteMagic on teistest teekidest märgatavalt kiirem. Lisaks on näha, et ükski teine teek ei tööta konstantselt hästi – DBFlow on küll primitiivseid andmetüüpe sisaldavate testide korral paremuselt teine, kuid kompleksandmete puhul jääb greenDAO-st maha. Lisaks on kompleksandmete puhul märgata trendi – kui üheastmelise sõltuvusega andmete korral töötab greenDAO peaaegu poole aeglasemalt ja teised umbes neli korda aeglasemalt, siis neljaastmelise sõltuvusega andmete korral töötab greenDAO 7 korda ja teised umbes 12 korda aeglasemalt, kui SqliteMagic.

Kõiki andmeid päriva operatsiooni testi tulemused (vt Joonis 10) on üpriski sarnased esimese kirje päringu tulemustega. Väga selgelt ilmutab *reflectioni* kasutamisest tulenev kiiruskaotus Sugar ORM teegi operatsioonikiiruses, mis on võrreldes teiste lahendustega enamasti vähemalt kümme korda aeglasem.



Joonis 10: Kõikide andmete pärimine.

Veelgi enam ilmutab kompleksandmete pärimiskiiruse erinevus SqliteMagic ja ülejäänud teekide vahel. Üheastmelise sõltuvusega andmete korral töötab greenDAO kaks korda ja ülejäänud umbes 26 korda aeglasemalt ning neljaastmelise sõltuvusega andmete korral greenDAO 23 korda ja teised umbes 43 korda aeglasemalt, kui SqliteMagic. Uurides teekide lähtekoodi on ilmne, et SqliteMagicu kiiruseelis kompleksandmete puhul tuleneb optimeeritud SQL-st – odavam on teostada üks andmebaasi päring ja sealt parsida kogu graafi struktuur objektideks, kui teostada iga kompleksvälja kohta eraldi päring ja parsida saadud objekt.

Ülejäänud andmetüüpide jaoks kiiruserinevustest arusaamiseks kasutati meetodite jälgimise tööriista. Mõõdeti 15000 kirjega segaandmete tabeli kõikide andmete pärimist. Joonis 18 (vt Lisa A) kuvab meetodite tööaja mõõtmistulemusi.

Meetodite mõõtmistulemustes on näha, et igas teegis algab päringuoperatsioon testi meetodiga *MixedDataHandler.queryAll* ning põhitöö on jaotunud loogiliselt kaheks – päringu täitmine, mille tulemusena saadakse reaalseid päringutulemustele vastavaid andmeid sisaldav kursor ning päringutulemuste parsimine Java objektideks. Seetõttu sõltub päringu kiirus, kas SQLi teostuskiirusest või sellest, kui optimeeritult kursorist andmeid kätte saadakse. Kuna testides täidetakse primitiivsete andmetüüpide korral kõikides teekides sama SQLi, peab kiiruserinevus tulenema kursori parsimisest.

SqliteMagic kasutab iserealiseeritud lahendust Androidi *Cursor* liidesest nimega *FastCursor*, mis teades kuidas andmeid päritakse, optimeerib palju ebavajalikust järjepidamisest. Vaadates meetodite mõõtmistulemusi, on selge, et viimasest tuleneb ka kiirusvõit. Näiteks DBFlow teegis kulutatakse 21.5% kogu operatsiooni ajast meetodile *SQLiteCursor.getColumnIndex* ning 10.5% ajast meetodile *AbstractWindowedCursor.checkPosition*, mis on osa kõikidest andmete lugemise meetoditest *AbstractWindowedCursor* objektis. GreenDAOs on esimese meetodi kasutamine välja optimeeritud, kuid endiselt kulutatakse 17% meetodile *AbstractWindowedCursor.checkPosition* ning omajagu aega läheb kaduma muu järjepidamise peale.

4.2 Mälukasutus

Kuna teegid jooksevad vägagi piiratud mäluga seadmetel, on kiirusega vähemalt sama oluline või isegi olulisem aspekt efektiivne mälukasutus. Iga operatsiooni mälukasutuse analüüsimiseks jooksutati üks test, mille käigus mõõdeti mälukasutust erinevate tööriistadega. Iga operatsiooni käigus kasutati testandmetena segaandmetega objekte, kuna antud andmetüüp sarnaneb kõige enam reaalelulistes olukordades kasutavatele andmetüüpidele. Iga test koosnes järgnevatest faasidest:

- Eelseadistusfaas – teostati rida operatsioone, et tagada vajalik andmebaasi seisund vaadeldava operatsiooni läbiviimiseks. Näiteks andmete sisestamiseks kustutati tabelist kõik eelnevad andmed ning allokeeriti kõik sisestatavad andmed. Lisaks käivitatakse selle faasi lõpus programmaatiliselt väljakutsutud mälukoristus meetodiga *System.gc*, mis tagab et põhitöö käigus toimuvad operatsioonisüsteemi initsialiseeritud mälukoristused poleks põhjustatud eelseadistusfaasi jääknähtudest.
- Puhkefaas – eristamaks eelseadistus- ja põhitööfaase peatati testi töö *Thread.sleep* meetodiga märgatavaks perioodiks. Mälu allokatsioonide graafilisel kujutulusel ilmnevad puhkefaasid sirgete platoodena, mistõttu on inimsilmal kerge eristada erinevaid faase.
- Põhitööfaas – käivitatakse testitav operatsioon.
- Puhkefaas – teostatakse teine testi töö peatamine, mis märgib põhitöö lõppu.
- Prahikoristusfaas – teostatakse programmaatiline mälukoristus *System.gc* meetodiga, mis koristab kõik üleliigsed testi käigus allokeeritud, kuid selle lõpuks vabastatud objektid.

Mälukasutust mõõdeti kahe tööriistaga:

- Mälu monitor (*memory monitor*) [22, ptk *Memory Monitor*] – näitab reaalajas rakenduse mälukasutust graafikuna. X-teljel kujutatakse aega ning y-teljel rakenduse

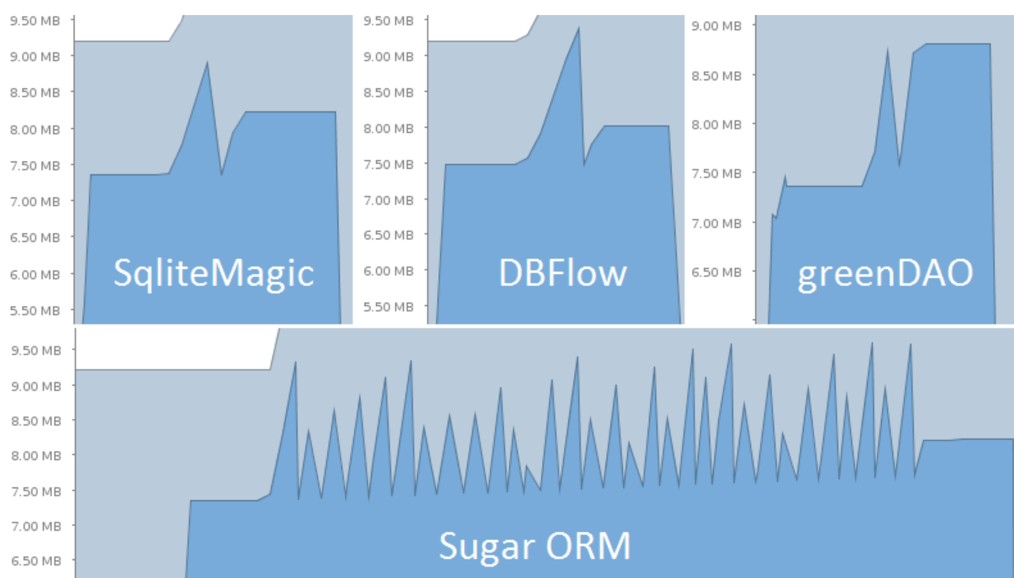
vaba (helesinine) ja allokeeritud (tumesinine) mälu mahtusid megabaitides. Iga tumesinise ala järsk vähenemine väljendab mälukoristussündmust, mistõttu mida vähem „sakke” graafikul on, seda optimaalsemalt kasutatakse mälu ning seda vähem „jõnkse” kasutaja näeb rakenduse kasutamisel. Antud tööriista graafikuid vaadeldes tuleb silmas pidada, et töö ajal allokeeritud mälu hulka ei väljenda mitte tumesiniste alade pindala suurus, vaid sakitippude summa. Näiteks, kui ühel graafikul tumesinine ala suureneb ühtlaselt ning jääb pidama mingil suurusel (graafikul tekib horisontaalne platoo) ja teisel graafikul suureneb tumesinine ala ühtlaselt sama väärtuseni, kuid kukub seejärel märgatavalt alla ja jätkab natuke suurenemist jäädes lõpuks pidama mingil suurusel, siis optimaalsemalt ja vähem mälu kasutas esimesel graafikul kujutatud rakendus.

- Allokatsioonide jälgija (*allocation tracker*) [22, ptk *Allocation Tracker*] – salvestab rakenduse mälu allokatsioone soovitud perioodi jooksul. Tööriista vahendusel näeb kõiki perioodi jooksul allokeeritud objekte koos nende tüübi, suuruse, allokeeriva lõime ja väljakutse hierarhiaga.

Järgnevalt on esitatud mälukasutuse analüüsi tulemused erinevate operatsioonide puhul.

4.2.1 Sisestus

Testi käigus sisestati ühe transaktsiooniga 15000 segaandmetega objekti tühja tabelisse. Joonis 11 väljendab graafiliselt teekide mälukasutust operatsiooni vältel. Nagu graafikutelt näha, siis teegid SqliteMagic ja DBFlow on mälukasutuselt kõige optimaalsemad – allokeeritud maht on võrreldes teistega väiksem ning kogu operatsiooni jooksul toimub mälukoristus ühel korral. Neile järgneb greenDAO samuti ühe mälukoristusega, kuid mahult allokeeritakse rohkem, millest võiks järeldada, et kui operatsioon oleks mõne millisekundi kauem kestnud oleks ka teine koristus toimunud. Kõige rohkem mälu raiskab *reflectionit* kasutav ning iga sisestuse jaoks uut INSERT lauset kompileeriv Sugar ORM (vt ptk 4.1.1).



Joonis 11: Andmete sisestuse mälukasutus.

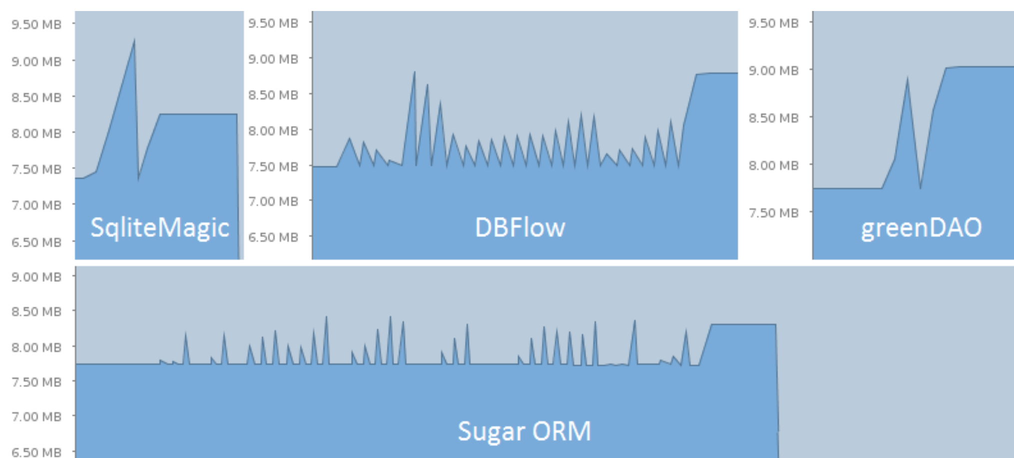
Mälukasutusest parema ülevaate saamiseks kasutati allokatsioonide jälgimistööriista (vt

Lisa B, Joonis 19). Tulemustest on näha, et teegid SqliteMagic, DBFlow ja greenDAO kulutavad mälu peamiselt primitiivsete andmetüüpide *autoboxingule*, mis tuleneb vältimatust *SQLiteStatement* sisemisest koodist, kus primitiivsed päringu argumendid sisestatakse *Object* massiivi. GreenDAO suurem mälueraldus võrreldes SqliteMagicuga tuleneb ilmselt vähesest optimeeritusest andmeobjektide väärtuste pärimisel, kus võivad toimuda mõned üleliigsed *autoboxingud* (vt ptk 4.1.1).

Vaadates Sugar ORM mälukasutust selgub, et *reflectioni* kasutamisele kulub üle 60% operatsiooni ajal eraldatud mälust ning ülejäänud kulub veerunimedele vastavate *String*-ide ehitamisele. Kuna ühegi objekti puhul puhverdamist ei kasutata, tuleb iga kirje salvestamiseks kõik objektid uuesti allokeerida. Kirjeldatust tulenevalt toimuvadki väga sagedased mälukoristussündmused, mistõttu näeb operatsiooni mälukasutuse graafik väga sakiline välja.

4.2.2 Andmete uuendamine

Testi käigus uuendati ühe transaktsiooniga 15000 segaandmetega objekti. Joonis 12 väljendab graafiliselt teekide mälukasutust operatsiooni vältel. Nagu graafikutelt näha, siis kõige efektiivsemalt kasutavad mälu SqliteMagic ja greenDAO – nii eraldatud mälu maht, kui ka mälukoristuste arv on teistest teekidest väiksem. Kõige rohkem raiskavad mälu DBFlow ja Sugar ORM, mis on ka mälukasutuselt suhteliselt sarnased.



Joonis 12: Andmete uuenduse mälukasutus.

Mälukasutusest parema ülevaate saamiseks kasutati allokatsioonide jälgimistööriista (vt Lisa B, Joonis 20). Tulemustest on näha, et SqliteMagic, greenDAO ja Sugar ORM allokeerivad mälu identselt sisestusoperatsioonile, tänu millele on ka mälukasutuste graafikud väga sarnased.

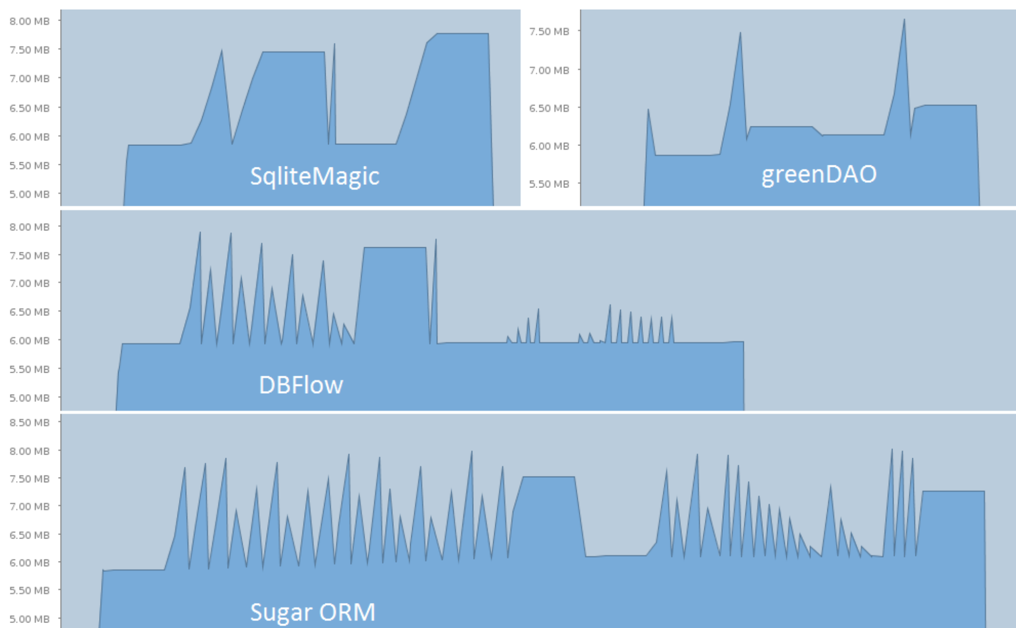
DBFlow erinev mälu allokeerimine tuleneb salvestusstrateegia muutumisest võrreldes sisestusoperatsiooniga. Kui sisestusel kasutati eelkompileeritud SQL lauseid, siis uuendamisel kompileeritakse uuendamislause iga kirje kohta uuesti. Kirjeldatut väljendavad ka mälus eraldatud objektid. Kuna ühegi objekti puhul puhverdamist ei kasutata, raisatakse väga palju mälu eraldades suurtes kogustes lühikeselt elavaid objekte, mistõttu ongi graafikul näha sagedased mälukoristused. Suures plaanis kulub mälu kahele asjale – uuendamislausete ehitamisele ning uuendatavate andmete järjepidamisele. Lisaks on näha,

kuidas Java standardteeki kuuluvad kollektsioonid, eesotsas *HashMap*-ga, võtavad üpriski palju mälu. Viimase kasutus tuleneb *ContentValues* objektist, mis on põhimõtteliselt *HashMap*-i pakend (*wrapper*). Kahjuks käib *ContentValues* objekti vahendusel enamus Androidi API andmebaasi operatsioonidest, mistõttu on selle kasutamisest tulenev ebaefektiivne mälukasutus vältimatu.

4.2.3 Tundmatu sisestus

Testi käigus kasutati tundmatu sisestuse operatsiooni ning salvestati ühe transaktsiooniga 10000 segaandmetega objekti tühja tabelisse peale mida objekte uuendati ning käivitati operatsioon uuesti. Joonis 13 väljendab graafiliselt teekide mälukasutust operatsiooni vältel. Testi tulemuste graafikuid vaadeldes tuleb seetõttu tähele panna, et graafiku keskel on platoo, millele järgneb enamasti üks sakk, peale mida on järjekordne platoo. Kirjeldatud piirkonda tuleks ignoreerida, kuna tegemist on andmete mälus uuendamisega, et neid ettevalmistada järgnevaks operatsiooni taaskäivitamiseks.

Võrreldes testitulemuse graafikuid andmete sisestamise ja uuendamise graafikutega on märgata ilmseid sarnasusi. Nagu peatükis 4.1.3 mainitud kasutavad SqliteMagic ja greenDAO tundmatu sisestuse korral eelkompileeritud SQL lauseid, mis tagab üpriski mälusäästliku teostuse. Viimast on näha ka graafikutelt, kus mõlema teegi puhul teostatakse mälukoristusi süsteemi poolt maksimaalselt ühe korra. Kuna DBFlow ja Sugar ORM antud operatsiooni puhul uuendamisega võrreldes (vt ptk 4.2.2) salvestusstrateegiat ei muuda, saab graafikute sarnasuse põhjal järeldada, et ka mälu eraldatakse sarnaselt.

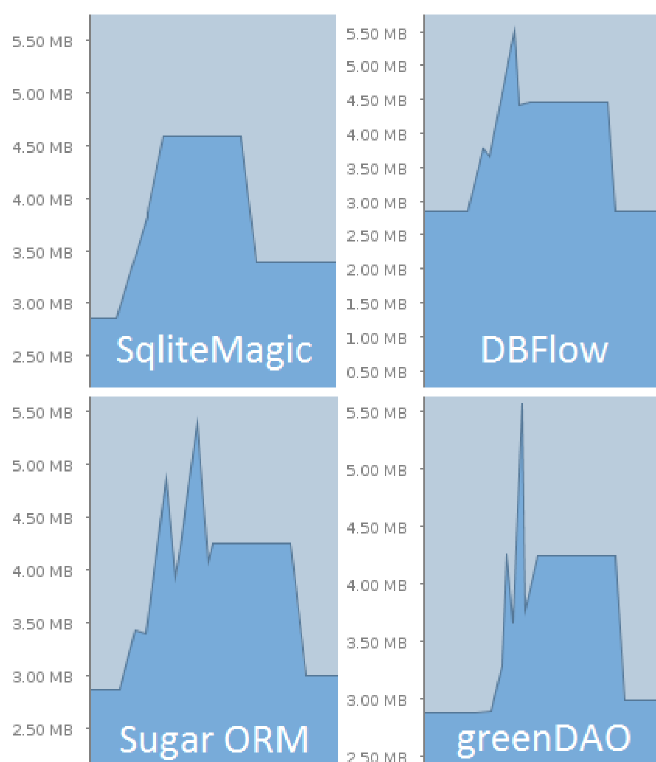


Joonis 13: Andmete salvestamise mälukasutus, kui pole teada kas tuleb uuendada või sisestada.

4.2.4 Pärimine

Teekide mälukasutust pärimisel mõõdeti kahes testis. Esiteks päriti 5000 kirjega tabelist esimest kirjet 1000 korda (vt Joonis 14). Teiseks päriti 15000 kirjega tabelist kõiki andmeid ühe korra (vt Joonis 15).

Esimest kirjet päriva testi tulemustest on näha, et kõige ühtlasemalt ja kontrollitumalt eraldab mälu SqliteMagic. Analüüsides operatsiooni objektide allokatsioone (vt Lisa B, Joonis 21) on näha, et 69.89% kulub andmete reaalseks pärimiseks Android API kaudu; ∼17% möödapääsmatule *autoboxingule*, andmebaasiühenduse sulgemisele ja igasugustele sisemistele järjepidamistele; ∼12% iserealiseeritud kursori allokeerimisele ja järjepidamisele ning reaalse päritud objekti allokeerimisele. Seetõttu võib öelda, et raisatud allokatsioonide arv on viidud miinimumini, mis väljendub ka graafikul.



Joonis 14: Tabeli esimese kirje pärimise mälukasutus.

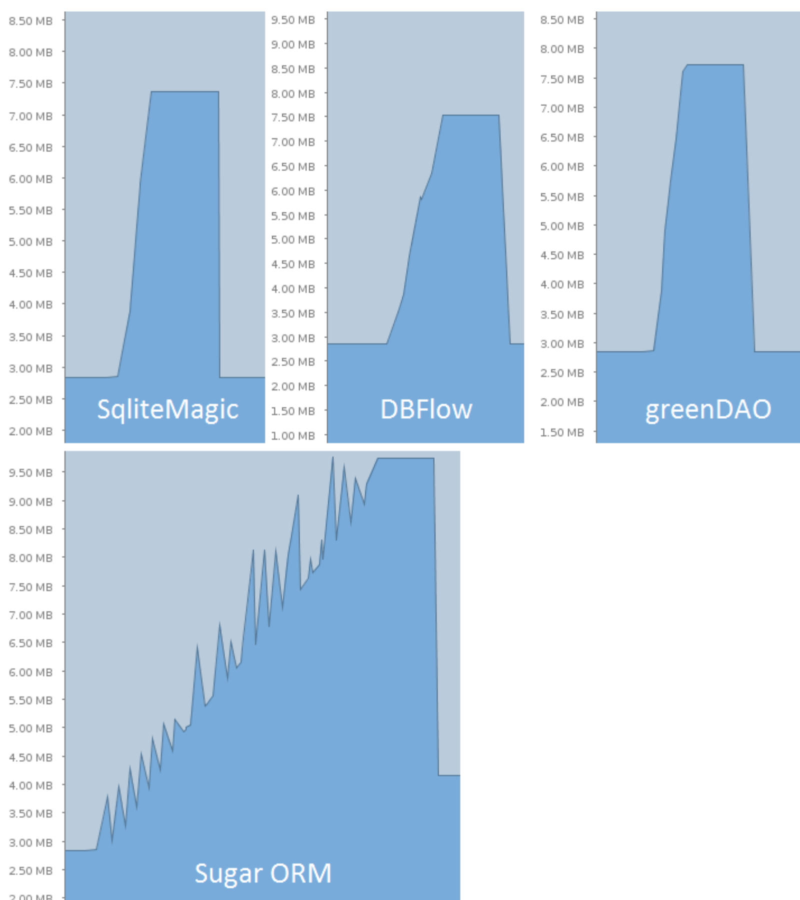
DBFlow puhul on graafikul näha peaaegu ühtlast mälu eraldamist. Samas on ilmne, et mälu maht ja selle eraldamise kiirus on märgatavalt suuremad, kui SqliteMagicul. Detailseid objektide eraldusi vaadatest tuleb välja, et lisaks andmetele, mida allokeerib SqliteMagic, on palju mälu eraldatud *AbstractStringBuilder*, *HashMap* ja paketis *com.raizlabs.android.dbflow.sql* asuvatele objektidele. Uurides teegi lähtekoodi ning allokatsioonide väljakutsepuid selgub, et lisamälu eraldamine tuleneb kahest kohast. Esiteks, kuna DBFlows pole päringu eelkompileerimise võimalust, nagu SqliteMagicus, siis iga päringu teostamiseks tuleb SQL uuesti kokku ehitada. Viimasest tulenevalt kulubki objektile *AbstractStringBuilder* 17.31% ja paketis *com.raizlabs.android.dbflow.sql* asuvatele objektidele 6.92% kogu operatsiooni ajal eraldatud mälust. Teiseks, nagu peatükis 4.1.4 mainitud, kasutab teek Android API poolt realiseeritud kursorit, milles väga palju aega kulub meetodi *SQLiteCursor.getColumnIndex* väljakutsumisele. Uurides nimetatud

klassi lähtekoodi tuleb välja, et sisemiselt kasutatakse veeru nimede ja nende indeksite kokkuviimiseks *HashMap* objekti. Kirjeldatul on lisaks kiiruskaotusele ka väga suur mõju mälukasutusele hõivates 15.34% kogu operatsiooni teostamiseks kulunud mälust.

Nagu kiiruse puhul, on ka mälu eraldamisel greenDAO ja Sugar ORM käitumine sarnane. Mõlemad eraldavad lühikese aja jooksul palju väikese eluajaga objekte, millest tulenevalt võibki graafikutel näha sakke. Sarnaselt eelnevalt vaadeldud operatsioonidele tuleneb Sugar ORM ebaefektiivne mälukasutus puhverdamata *reflection*-i kasutamisest, mida kinnitavad ka allokatsioonide jälgija mõõtetulemused.

Kuigi greenDAOs on päringu eelkompileerimise võimalus nõutakse, et päringut loov lõim oleks sama, mis seda läbiviiv lõim, mistõttu ei saa seda luua põhilõimes ja kasutada suvalises taustatöölõimes. Üks võimalus oleks päring esimesel kasutamisel puhverdada, kuid sel juhul ei annaks testid reaalelulisi tulemusi. Näiteks on väga tavaline, et reaalses rakenduses antakse päringu töö täita suvalisele lõimele tasutatöö lõimede kogumist (*thread pool*) ning päringu tulemused saadetakse põhilõimesse, mis teostab nende kuvamist. Seetõttu tuleb igal päringul defineerida ka uus päringu objekt. Kõigest tulenevalt ongi operatsiooni allokatsioone analüüsides näha, et *AbstractStringBuilder*-le kulutatakse 37.79% ning sisemistele päringuobjektidele 11.23% kogu eraldatud mälust.

Kõiki andmeid päriva testi tulemused (vt Joonis 15) on SqliteMagic ja Sugar ORM teekide puhul vägagi sarnased esimese kirje päringu tulemustega, kuid greenDAO ja DBFlow mälukasutused on paranenud. Viimast tingib asjaolu, kus päringut teostatakse vaid ühe korra, mistõttu on selle teostamiseks tarvilikke objekte vaja eraldada kõigest üks kord.



Joonis 15: Kõikide andmete pärimise mälukasutus.

Vaadates operatsiooni objektide allokatsioone (vt Lisa B, Joonis 22) on näha, et teegid SqliteMagic, DBFlow ja greenDAO eraldavad mälu peaaegu identselt, mistõttu on ka nende graafikud väga sarnased. Kuna Sugar ORM endiselt kasutab päringuks optimeerimata *reflectionit*, mis toodab palju lühikese eluajaga objekte, siis tulemuseks on tihedad mälukoristused. Kirjeldatut on näha nii graafikult, mis on üpriski sakiline, kui ka objektide allokatsioonides, millest enamus on tingitud *reflectioni* kasutamisest.

4.3 Analüüsi tulemused

Jõudlustestide tulemuste analüüsi käigus selgus, et antud töö tulemusena valminud teek töötab peaaegu kõikide operatsioonide puhul teistest teekidest kiiremini. Ainsaks erinevuseks on tundmatu sisestuse operatsioon tühja tabelisse, mille puhul jäädi märgatavalt alla greenDAO-le. Teisalt, sama operatsiooni kiirus täis tabeli puhul oli testest teekidest märgatavalt kiirem. Seega keskmiselt on SqliteMagic ja greenDAO tundmatu sisestuse puhul kiiruselt samaväärsed. Kuna reaalsetes rakendustes esineb nimetatud operatsiooni puhul andmete olemasolust tingitud uuendamist rohkem, saaks järeldada, et SqliteMagic on reaalelulisteks olukordadeks paremini optimeeritud.

Mälukasutustestide tulemuste analüüsi käigus selgus, et SqliteMagicu mälukasutus on kõikide operatsioonide puhul optimaalsem või samaväärne võrreldud teekidega.

Kokkuvõtlikult võib järeldada, et antud töö tulemusena valminud teegis rakendatud teoreetilised optimeeringud ja lahendused tagavad edu ka reaalsetes rakendustes. Lisaks ollakse operatsioonide teostuses konkurentidest märgatavalt stabiilsem ja efektiivsem.

5 Teegi tulevik ja edasised arendused

Antud töö valmimise hetkel on loodud teek avalikustatud Bitbucket keskkonnas avatud lähtekoodiga [27] ning kirjeldatud funktsionaalsuste ulatuses täielikult tootmises kasutatav. Teeki kasutab aktiivselt töö autor ning tema tuttavatest koosnev testgrupp, kelle ülesanne on anda esmast tagasisidet, leida vigu ning soovitada puuduvaid funktsionaalsusi. Tuleviku plaan on peale funktsionaalsuste pagasi ja API stabiliseerumist kogu kood migreerida n-ö avatud lähtekoodiga projektide mekasse, GitHubi. Tulevikulootuseks on teegi kasutamine viia massidesse ning muutuda kogukonna standardiks SQLite andmebaasiga suhtluse korraldamises.

Kuigi kõik valminud funktsionaalsused on põhjalikult testitud ja täielikult tootmises kasutatavad, leiab töö autor, et enne versiooni 1.0.0 väljalaskmist tuleb realiseerida veel mõned olulised funktsionaalsused. Järgnevalt on neist välja toodud kaks kõige suuremat.

5.1 Tüübiohutu päringute API

Siiani valminud päringute API on funktsionaalsuselt üpriski täielik ja voolav. Ainsaks puuduseks on päringut modifitseerivate meetodite parameetrid, mis aktsepteerivad kõiki *Object* tüüpi laiendavaid objekte. Esiteks on kirjeldatu halb tüübiteisendajaid kasutavate andmetüüpide jaoks, kuna teisendus tuleb teostada käsitsi. Siinkohal on mõistlik kasutada tüübiteisendajate staatilisi meetodeid, kuid sellest hoolimata on tegevus üpriski tüütu ja ebaintuitiivne. Teiseks puudub tüübikindlus – ükski mehhanism ei takista päringu parameetrina kasutada andmetüüpi, mis ei vasta viidatava veeru andmetüübile.

Kirjeldatud puuduste adresseerimiseks on planeeritud jOOQ teegist inspireeritud tüübiohutu päringute lahendus [18]. Andmemudelitele genereeritud veergudele vastavate tavaliste sõnede asemel genereeritakse veeru olemust väljendavad muutumatud objektid. Viimaseid saab otse päringutes kasutada ning Java geneerika vahendusel tagatakse kõik õiged andmetüübid. Lisaks, kuna veeruobjektid on teegi genereeritud saab neisse lisada kõik vajalikud tüübiteisendused.

5.2 Automaatne andmebaasi migratsioon

Kuna kogu andmebaasi struktuur on defineeritud antud teeki kasutades ja andmebaasi migratsioone viiakse läbi SQL faile käivitades (vt ptk 2.9), siis teoreetiliselt on võimalik migratsiooni protsessi automatiseerida. Esialgseks lahenduseks on planeeritud järgnev:

- Enne iga rakenduse andmebaasi versiooni väljastamist tuleb käivitada Gradle *task*, mille tulemusena salvestatakse andmebaasi struktuuri tõmmis JSON formaadis projekti kausta, mis asub väljaspool rakenduse faile.
- Kui leidub eelneva versiooni tõmmis, siis võrreldakse seda uue versiooni tõmmisega. Võrdluse tulemusena genereeritakse rakenduse migratsioonifailide kausta migratsiooni läbiviivad SQL laused faili, mis on uue versiooni nimega.

Kokkuvõte

Töö põhieesmärgiks oli luua Android SQLite ORM teek, mida oleks kerge kasutada, kuid mille operatsioonide kiirus oleks kiirem või vähemalt sama kiire, kui olemasolevastes lahendustes. Seejuures sooviti, et loodav teek ei kasutaks üldse *reflectionit*, oleks mälukasutuselt optimeeritud ning millel oleks lisaks tavapärastele andmebaasioperatsioonidele funktsionaalsused, millega oleks võimalik tihedas konkurentsis teistest eristuda.

Põhjalike taustauuringute, kogukonna trendide ja konkurentide analüüsi tulemusena valmis kõiki nõudmisi ja probleeme lahendav avatud lähtekoodiga teek nimega SqliteMagic. Teek ei kasuta üldse *reflectionit* ning seda on võrreldes alternatiividega lihtne uude projekti lisada või olemasolevasse kaasata. Loodud API on disainitud olema kergesti kasutatav, paindlik, kasutajat abistav ja suunav. Mitmelõimelise töö jaoks on tugi RxJava vahendusel, mille abil realiseeriti ka tabeli andmete muudatuste teavitamise süsteem. Konkurentidest eristumiseks on teegis esmaklassiline tugi muutumatute ja kompleksväljadega andmemudelitele.

Tehtud töö valideerimiseks võrreldi seda erinevate lahenduste silmapaistvamate teekidega. Analüüsi tulemusena selgus, et tänu rakendatud optimeeringutele on SqliteMagic konkurentidest enamasti kiirem ning optimeerituma või samaväärse mälukasutusega. Lisaks jõuti järeldusele, et teek on nimetatud aspektides ka konkurentidest stabiilsem ning paremini optimeeritud reaalelulisteks olukordadeks.

Võttes kokku töö tulemused, võib öelda, et kuigi kõik püstitatud eesmärgid täideti, vajas selleni jõudmine mitmete esmapilgul võimatute takistuste ületamist ning märkimisväärselt raamidest väljaspool mõtlemist. Siiski õnnestus takistused ületada ning tulemuseks on teek, millel on potentsiaali muutuda Android kogukonna SQLite andmebaasisuhtluse standardlahenduseks.

Viited

[1] Android Studio overview, 2015. https://developer.android.com/tools/studio/index.html, 4. mai 2015.

[2] Apache Harmony, 2011. http://harmony.apache.org/, 8. detsember 2015.

[3] ART and Dalvik, 2015. https://source.android.com/devices/tech/dalvik/index.html, 8. detsember 2015.

[4] AutoParcel, 2015. https://github.com/frankiesardo/auto-parcel, 8. detsember 2015.

[5] AutoValue, 2015. https://github.com/google/auto/tree/master/value, 8. detsember 2015.

[6] Joshua Bloch. *Effective Java (2nd Edition) (The Java Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2 edition, 2008.

[7] Building apps with over 65k methods, 2015. https://developer.android.com/tools/building/multidex.html#about, 4. mai 2015.

[8] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[9] DBFlow, 2015. https://github.com/Raizlabs/DBFlow, 8. detsember 2015.

[10] Gradle plugin user guide, 2015. http://tools.android.com/tech-docs/new-build-system/user-guide, 8. detsember 2015.

[11] greenDAO, 2015. http://greendao-orm.com/, 8. detsember 2015.

[12] Groovy issue ticket, 2015. https://github.com/groovy/groovy-android-gradle-plugin/issues/65, 8. detsember 2015.

[13] Groovy runtime and compile-time metaprogramming, 2015. http://www.groovy-lang.org/metaprogramming.html, 8. detsember 2015.

[14] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[15] Java Beans, 2015. https://en.wikibooks.org/wiki/Java_Programming/JavaBeans, 8. detsember 2015.

[16] Java compilation overview, 2015. http://openjdk.java.net/groups/compiler/doc/compilation-overview/, 8. detsember 2015.

[17] Java reflection API, 2015. http://docs.oracle.com/javase/tutorial/reflect/index.html, 8. detsember 2015.

[18] jOOQ teegi koduleht, 2015. http://www.jooq.org/, 8. detsember 2015.

[19] JSR 269: Pluggable annotation processing API, 2006. https://www.jcp.org/en/jsr/detail?id=269, 8. detsember 2015.

[20] Lombok, 2015. https://projectlombok.org/, 8. detsember 2015.

[21] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edition, 2008.

[22] Memory profilers, 2015. `http://developer.android.com/tools/performance/comparison.html`, 8. detsember 2015.

[23] Morpheus, 2015. `https://github.com/stephanenicolas/morpheus`, 8. detsember 2015.

[24] Performance tips, 2015. `http://developer.android.com/training/articles/perf-tips.html`, 30. november 2015.

[25] RxJava wiki, 2015. `https://github.com/ReactiveX/RxJava/wiki`, 4. mai 2015.

[26] SQLBrite, 2015. `https://github.com/square/sqlbrite`, 8. detsember 2015.

[27] SqliteMagic projekti lähtekoodi koduleht, 2015. `https://bitbucket.org/siimkinks/sqlitemagic/`, 8. detsember 2015.

[28] Sugar ORM, 2015. `http://satyan.github.io/sugar/`, 8. detsember 2015.

[29] Support annotations, 2015. `http://tools.android.com/tech-docs/support-annotations`, 8. detsember 2015.

[30] Jake Wharton and Jesse Wilson. Droidcon montreal keynote - an open source advantage, 2015. `https://www.youtube.com/watch?v=PCxz2LEmuL4`, 2. mai 2015.

A Lisa: meetodite jälgimisvaated

Name — SqliteMagic	Incl Cp	Incl Cpu T
0 (toplevel)	100.0%	2121.708
1 com/siimkinks/sqlite/speedtests/handlers/MixedDataHandler.bulkUpdate (Ljava/util/List;)V	96.6%	2048.577
2 com/siimkinks/sqlitemagic/SqliteMagic_MixedData_Handler$BulkUpdateBuilder.execute ()Z	96.5%	2047.840
Parents		
1 com/siimkinks/sqlite/speedtests/handlers/MixedDataHandler.bulkUpdate (Ljava/util/List;)V	100.0%	2047.840
Children		
self	1.1%	22.018
3 android/database/sqlite/SQLiteStatement.executeUpdateDelete ()I	76.1%	1557.980
6 com/siimkinks/sqlite/speedtests/data/SqliteMagic_MixedData_Dao.bindToUpdateStatement (	21.9%	448.248
118 java/util/ArrayList$ArrayListIterator.next ()Ljava/lang/Object;	0.3%	5.548
129 java/util/ArrayList$ArrayListIterator.hasNext ()Z	0.3%	5.129
147 com/siimkinks/sqlitemagic/SqliteMagic_MixedData_Handler.getUpdateStatement ()Landrc	0.2%	4.609
153 com/siimkinks/sqlitemagic/SqlManager.beginTransaction ()V	0.2%	4.201
512 java/util/ArrayList.iterator ()Ljava/util/Iterator;	0.0%	0.075
687 com/siimkinks/sqlitemagic/SqliteMagic_MixedData_Handler.access$200 ()Ljava/util/concu	0.0%	0.027
751 com/siimkinks/sqlitemagic/SqliteMagic.getSqlManager ()Lcom/siimkinks/sqlitemagic/SqlM	0.0%	0.005
(context switch)	0.0%	0.000
3 android/database/sqlite/SQLiteStatement.executeUpdateDelete ()I	73.4%	1557.980
4 android/database/sqlite/SQLiteSession.executeForChangedRowCount (Ljava/lang/String;[Ljava/lan	62.6%	1327.655
5 android/database/sqlite/SQLiteConnection.executeForChangedRowCount (Ljava/lang/String;[Ljava,	48.0%	1018.996
6 com/siimkinks/sqlite/speedtests/data/SqliteMagic_MixedData_Dao.bindToUpdateStatement (Landr	21.1%	448.248
Parents		
Children		
self	14.8%	66.514
11 android/database/sqlite/SQLiteProgram.bindLong (IJ)V	41.8%	187.247
15 android/database/sqlite/SQLiteProgram.bindDouble (ID)V	30.5%	136.912
62 android/database/sqlite/SQLiteProgram.bindString (ILjava/lang/String;)V	6.9%	31.102
73 android/database/sqlite/SQLiteProgram.clearBindings ()V	3.8%	16.844
68 java/lang/Long.longValue ()J	1.1%	4.853
70 java/lang/Double.doubleValue ()D	1.1%	4.776
(context switch)	0.0%	0.000

Name — greenDAO	Incl Cpu	Incl Cpu T
0 (toplevel)	100.0%	2184.818
1 com/siimkinks/sqlite/speedtests/handlers/MixedDataHandler.bulkUpdate (Ljava/u	95.8%	2093.315
2 de/greenrobot/dao/AbstractDao.updateInTx (Ljava/lang/Iterable;)V	95.5%	2087.155
Parents		
1 com/siimkinks/sqlite/speedtests/handlers/MixedDataHandler.bulkUpdate (l	100.0%	2087.155
Children		
self	0.6%	13.223
3 de/greenrobot/dao/AbstractDao.updateInsideSynchronized (Ljava/lang/Ob	98.1%	2046.901
87 de/greenrobot/dao/internal/TableStatements.getUpdateStatement ()Lan	0.6%	13.452
146 android/database/sqlite/SQLiteDatabase.beginTransaction ()V	0.2%	4.647
153 java/util/ArrayList$ArrayListIterator.next ()Ljava/lang/Object;	0.2%	4.542
178 java/util/ArrayList$ArrayListIterator.hasNext ()Z	0.2%	4.106
447 de/greenrobot/dao/identityscope/IdentityScopeLong.lock ()V	0.0%	0.204
598 java/util/ArrayList.iterator ()Ljava/util/Iterator;	0.0%	0.080
3 de/greenrobot/dao/AbstractDao.updateInsideSynchronized (Ljava/lang/Object;L	93.7%	2046.901
Parents		
Children		
self	1.4%	29.098
4 android/database/sqlite/SQLiteStatement.execute ()V	67.4%	1380.630
7 com/siimkinks/sqlite/speedtests/data/MixedDataDao.bindValues (Landroid/	23.9%	489.271
21 de/greenrobot/dao/AbstractDao.attachEntity (Ljava/lang/Object;Ljava/lan	4.2%	86.606
12 android/database/sqlite/SQLiteProgram.bindLong (IJ)V	1.8%	37.728
76 com/siimkinks/sqlite/speedtests/data/MixedDataDao.getKey (Ljava/lang/C	1.0%	19.668
60 java/lang/Long.longValue ()J	0.2%	3.900
(context switch)	0.0%	0.000
4 android/database/sqlite/SQLiteStatement.execute ()V	63.2%	1380.630
5 android/database/sqlite/SQLiteSession.execute (Ljava/lang/String;[Ljava/lang/Ob	54.6%	1191.857
6 android/database/sqlite/SQLiteConnection.execute (Ljava/lang/String;[Ljava/lang	43.1%	941.655
7 com/siimkinks/sqlite/speedtests/data/MixedDataDao.bindValues (Landroid/datab	22.4%	489.271
8 com/siimkinks/sqlite/speedtests/data/MixedDataDao.bindValues (Landroid/datab	22.0%	481.105
Parents		
Children		
self	23.5%	113.209
12 android/database/sqlite/SQLiteProgram.bindLong (IJ)V	31.7%	152.496
18 android/database/sqlite/SQLiteProgram.bindDouble (ID)V	23.3%	112.174
73 android/database/sqlite/SQLiteProgram.bindString (ILjava/lang/String;)V	5.3%	25.449
86 android/database/sqlite/SQLiteProgram.clearBindings ()V	2.8%	13.549
60 java/lang/Long.longValue ()J	1.7%	8.038
169 com/siimkinks/sqlite/speedtests/data/MixedData.getTwoL ()J	0.9%	4.245
79 java/lang/Double.doubleValue ()D	0.9%	4.112
177 com/siimkinks/sqlite/speedtests/data/MixedData.getOne ()Ljava/lang/St	0.9%	4.107
181 com/siimkinks/sqlite/speedtests/data/MixedData.getOneL ()J	0.8%	4.082
116 com/siimkinks/sqlite/speedtests/data/MixedData.getId ()Ljava/lang/Long	0.8%	4.048
182 com/siimkinks/sqlite/speedtests/data/MixedData.getThree ()Ljava/lang/S	0.8%	4.032
185 com/siimkinks/sqlite/speedtests/data/MixedData.getTwo ()Ljava/lang/St	0.8%	3.986
186 com/siimkinks/sqlite/speedtests/data/MixedData.getThreeL ()Ljava/lang,	0.8%	3.980
188 com/siimkinks/sqlite/speedtests/data/MixedData.getFourD ()Ljava/lang/l	0.8%	3.977
189 com/siimkinks/sqlite/speedtests/data/MixedData.getFourL ()Ljava/lang/L	0.8%	3.977
194 com/siimkinks/sqlite/speedtests/data/MixedData.getThreeD ()Ljava/lang	0.8%	3.931
196 com/siimkinks/sqlite/speedtests/data/MixedData.getFour ()Ljava/lang/St	0.8%	3.916
197 com/siimkinks/sqlite/speedtests/data/MixedData.getTwoD ()D	0.8%	3.903
198 com/siimkinks/sqlite/speedtests/data/MixedData.getOneD ()D	0.8%	3.894
(context switch)	0.0%	0.000

Joonis 16: SqliteMagic, DBFlow ja greenDAO uuendamisoperatsiooni meetodite jälgimisvaade.

Name — SqliteMagic	Incl Cpu	Incl Cpu T
0 (toplevel)	100.0%	2216.893
1 com/siimkinks/sqlite/speedtests/handlers/MixedDataHandler.bulkInsert (Ljava/util/List;)V	95.1%	2108.530
2 com/siimkinks/sqlitemagic/SqliteMagic_MixedData_Handler$BulkInsertBuilder.execute ()Z	95.0%	2105.926
Parents		
1 com/siimkinks/sqlite/speedtests/handlers/MixedDataHandler.bulkInsert (Ljava/util/List;)V	100.0%	2105.926
Children		
self	1.6%	33.943
3 android/database/sqlite/SQLiteStatement.executeInsert ()J	75.4%	1588.272
6 com/siimkinks/sqlite/speedtests/data/SqliteMagic_MixedData_Dao.bindToInsertStatement (	19.6%	413.603
17 java/lang/Long.valueOf (J)Ljava/lang/Long;	1.3%	28.169
90 com/siimkinks/sqlite/speedtests/data/SqliteMagic_MixedData_Dao.setId (Lcom/siimkinks/s	0.7%	15.705
105 com/siimkinks/sqlitemagic/SqlManager.beginTransaction ()V	0.4%	8.778
131 java/util/ArrayList$ArrayListIterator.next ()Ljava/lang/Object;	0.3%	5.800
144 java/util/ArrayList$ArrayListIterator.hasNext ()Z	0.3%	5.271
165 com/siimkinks/sqlitemagic/SqliteMagic_MixedData_Handler.getInsertStatement ()Landro	0.2%	4.124
118 dalvik/system/VMDebug.startClassPrep ()V	0.1%	2.153
610 java/util/ArrayList.iterator ()Ljava/util/Iterator;	0.0%	0.070
774 com/siimkinks/sqlitemagic/SqliteMagic_MixedData_Handler.access$000 ()Ljava/util/conc	0.0%	0.025
768 com/siimkinks/sqlitemagic/SqliteMagic.getSqlManager ()Lcom/siimkinks/sqlitemagic/Sql	0.0%	0.013
(context switch)	0.0%	0.000
3 android/database/sqlite/SQLiteStatement.executeInsert ()J	71.6%	1588.272
4 android/database/sqlite/SQLiteSession.executeForLastInsertedRowId (Ljava/lang/String;[Ljava/lar	60.7%	1346.532
5 android/database/sqlite/SQLiteConnection.executeForLastInsertedRowId (Ljava/lang/String;[Ljava	46.7%	1034.194
6 com/siimkinks/sqlite/speedtests/data/SqliteMagic_MixedData_Dao.bindToInsertStatement (Landr	18.7%	413.603
Parents		
Children		
self	15.4%	63.588
14 android/database/sqlite/SQLiteProgram.bindLong (IJ)V	35.2%	145.770
15 android/database/sqlite/SQLiteProgram.bindDouble (ID)V	34.8%	143.833
60 android/database/sqlite/SQLiteProgram.bindString (ILjava/lang/String;)V	8.0%	32.970
80 android/database/sqlite/SQLiteProgram.clearBindings ()V	4.2%	17.293
69 java/lang/Double.doubleValue ()D	1.2%	5.080
66 java/lang/Long.longValue ()J	1.2%	5.069

Name — DBFlow	Incl Cpu Ti	Incl Cpu T
0 (toplevel)	100.0%	2011.793
1 com/siimkinks/sqlite/speedtests/handlers/MixedDataHandler.bulkInsert (Ljava/util/List;)V	95.7%	1925.774
2 com/raizlabs/android/dbflow/runtime/TransactionManager.transact (Ljava/lang/String;Lja	95.6%	1922.879
3 com/raizlabs/android/dbflow/runtime/TransactionManager.transact (Landroid/database/s	95.6%	1922.528
4 com/siimkinks/sqlite/speedtests/handlers/MixedDataHandler$$Lambda$1.run ()V	95.3%	1917.599
5 com/siimkinks/sqlite/speedtests/handlers/MixedDataHandler.access$lambda$0 (Ljava/uti	95.3%	1917.573
6 com/siimkinks/sqlite/speedtests/handlers/MixedDataHandler.lambda$bulkInsert$12 (Ljav	95.3%	1917.553
7 com/raizlabs/android/dbflow/structure/BaseModel.insert ()V	94.2%	1894.830
Parents		
Children		
self	0.7%	12.777
8 com/raizlabs/android/dbflow/structure/ModelAdapter.insert (Lcom/raizlabs/android,	94.4%	1789.019
24 com/raizlabs/android/dbflow/structure/BaseModel.getModelAdapter ()Lcom/raizlal	4.9%	93.034
(context switch)	0.0%	0.000
8 com/raizlabs/android/dbflow/structure/ModelAdapter.insert (Lcom/raizlabs/android/dbflo	88.9%	1789.019
9 com/raizlabs/android/dbflow/sql/SqlUtils.insert (Lcom/raizlabs/android/dbflow/structure/M	88.4%	1779.243
Parents		
Children		
self	1.6%	28.383
10 android/database/sqlite/SQLiteStatement.executeInsert ()J	73.3%	1304.120
13 com/siimkinks/sqlite/speedtests/data/MixedData$Adapter.bindToStatement (Land	22.4%	399.359
77 com/raizlabs/android/dbflow/sql/SqlUtils.notifyModelChanged (Lcom/raizlabs/andi	1.4%	24.133
99 com/siimkinks/sqlite/speedtests/data/MixedData$Adapter.updateAutoIncrement (	0.7%	12.774
108 com/raizlabs/android/dbflow/structure/ModelAdapter.getInsertStatement ()Lanc	0.5%	8.786
124 dalvik/system/VMDebug.startClassPrep ()V	0.1%	1.108
304 com/raizlabs/android/dbflow/structure/BaseModel$Action.<clinit> ()V	0.0%	0.580
10 android/database/sqlite/SQLiteStatement.executeInsert ()J	64.8%	1304.120
11 android/database/sqlite/SQLiteSession.executeForLastInsertedRowId (Ljava/lang/String;	55.0%	1107.019
12 android/database/sqlite/SQLiteConnection.executeForLastInsertedRowId (Ljava/lang/Str	42.4%	852.223
13 com/siimkinks/sqlite/speedtests/data/MixedData$Adapter.bindToStatement (Landroid/d	19.9%	399.359
14 com/siimkinks/sqlite/speedtests/data/MixedData$Adapter.bindToStatement (Landroid/d	19.4%	390.848
Parents		
Children		
self	16.9%	66.130
22 android/database/sqlite/SQLiteProgram.bindLong (IJ)V	31.1%	121.690
23 android/database/sqlite/SQLiteProgram.bindDouble (ID)V	30.1%	117.654
52 android/database/sqlite/SQLiteProgram.bindNull (I)V	12.8%	50.117
74 android/database/sqlite/SQLiteProgram.bindString (ILjava/lang/String;)V	6.9%	26.922
85 java/lang/Long.longValue ()J	1.1%	4.211
84 java/lang/Double.doubleValue ()D	1.1%	4.124
(context switch)	0.0%	0.000

Name — greenDAO	Incl Cpu	Incl Cpu T
0 (toplevel)	100.0%	2245.014
1 com/siimkinks/sqlite/speedtests/handlers/MixedDataHandler.bulkInsert (Ljava/u	94.8%	2127.220
2 de/greenrobot/dao/AbstractDao.insertInTx (Ljava/lang/Iterable;Z)V	94.5%	2121.497
Parents		
1 com/siimkinks/sqlite/speedtests/handlers/MixedDataHandler.bulkInsert (l	100.0%	2121.497
Children		
self	0.0%	0.046
3 de/greenrobot/dao/AbstractDao.executeInsertInTx (Landroid/database/sc	99.4%	2108.280
110 de/greenrobot/dao/internal/TableStatements.getInsertStatement ()Lar	0.6%	13.171
3 de/greenrobot/dao/AbstractDao.executeInsertInTx (Landroid/database/sqlite/S	93.9%	2108.280
Parents		
Children		
self	1.1%	23.908
4 android/database/sqlite/SQLiteStatement.executeInsert ()J	67.3%	1419.077
7 com/siimkinks/sqlite/speedtests/data/MixedDataDao.bindValues (Landroi	22.3%	470.378
14 de/greenrobot/dao/AbstractDao.updateKeyAfterInsertAndAttach (Ljava/	8.6%	180.823
149 java/util/ArrayList$ArrayListIterator.next ()Ljava/lang/Object;	0.2%	4.903
174 java/util/ArrayList$ArrayListIterator.hasNext ()Z	0.2%	4.391
189 android/database/sqlite/SQLiteDatabase.beginTransaction ()V	0.2%	4.275
272 dalvik/system/VMDebug.startClassPrep ()V	0.0%	0.267
498 de/greenrobot/dao/identityscope/IdentityScopeLong.lock ()V	0.0%	0.188
684 java/util/ArrayList.iterator ()Ljava/util/Iterator;	0.0%	0.070
(context switch)	0.0%	0.000
4 android/database/sqlite/SQLiteStatement.executeInsert ()J	63.2%	1419.077
5 android/database/sqlite/SQLiteSession.executeForLastInsertedRowId (Ljava/lan	54.1%	1214.914
6 android/database/sqlite/SQLiteConnection.executeForLastInsertedRowId (Ljava	42.3%	949.672
7 com/siimkinks/sqlite/speedtests/data/MixedDataDao.bindValues (Landroid/data	21.0%	470.378
8 com/siimkinks/sqlite/speedtests/data/MixedDataDao.bindValues (Landroid/data	20.6%	461.615
Parents		
Children		
self	23.9%	110.310
18 android/database/sqlite/SQLiteProgram.bindLong (IJ)V	26.7%	123.093
19 android/database/sqlite/SQLiteProgram.bindDouble (ID)V	26.3%	121.433
72 android/database/sqlite/SQLiteProgram.bindString (ILjava/lang/String;)'	6.1%	27.964
96 android/database/sqlite/SQLiteProgram.clearBindings ()V	3.2%	14.848
178 com/siimkinks/sqlite/speedtests/data/MixedData.getId ()Ljava/lang/Lor	0.9%	4.342
183 com/siimkinks/sqlite/speedtests/data/MixedData.getThreeL ()Ljava/lan	0.9%	4.287
182 com/siimkinks/sqlite/speedtests/data/MixedData.getFourL ()Ljava/lang	0.9%	4.287
184 com/siimkinks/sqlite/speedtests/data/MixedData.getTwoD ()D	0.9%	4.283
185 com/siimkinks/sqlite/speedtests/data/MixedData.getOneD ()D	0.9%	4.280
187 com/siimkinks/sqlite/speedtests/data/MixedData.getThreeD ()Ljava/lan	0.9%	4.278
91 java/lang/Double.doubleValue ()D	0.9%	4.275
190 com/siimkinks/sqlite/speedtests/data/MixedData.getOneL ()J	0.9%	4.259
191 com/siimkinks/sqlite/speedtests/data/MixedData.getTwo ()Ljava/lang/S	0.9%	4.258
193 com/siimkinks/sqlite/speedtests/data/MixedData.getTwoL ()J	0.9%	4.252
194 com/siimkinks/sqlite/speedtests/data/MixedData.getThree ()Ljava/lang	0.9%	4.248
197 com/siimkinks/sqlite/speedtests/data/MixedData.getFour ()Ljava/lang/!	0.9%	4.232
198 com/siimkinks/sqlite/speedtests/data/MixedData.getFourD ()Ljava/lang	0.9%	4.231
199 com/siimkinks/sqlite/speedtests/data/MixedData.getOne ()Ljava/lang/S	0.9%	4.228
76 java/lang/Long.longValue ()J	0.9%	4.227
(context switch)	0.0%	0.000

Joonis 17: SqliteMagic, DBFlow ja greenDAO sisestamisoperatsiooni meetodite jälgimisvaade.

SqliteMagic

Name	Incl Cpu %	Incl Cpu Time
0 (toplevel)	100.0%	2337.712
1 com/siimkinks/sqlite/speedtests/handlers/MixedDataHandler.queryAll ()Ljava/util/List;	96.5%	2256.819
2 com/siimkinks/sqlitemagic/CompiledSelect.execute ()Ljava/util/List;	96.5%	2256.805
3 com/siimkinks/sqlitemagic/CompiledSelect.executeInternal (Lrx/Subscription;)Ljava/util/List;	96.5%	2256.797
Parents		
Children		
self	0.0%	0.049
4 com/siimkinks/sqlitemagic/SqlUtil.getAllFromCursor (Ljava/lang/String;Lcom/siimkinks/sqlitemagic/	87.9%	1982.655
14 com/siimkinks/sqlitemagic/SqliteMagicCursor.getFastCursor ()Lcom/siimkinks/sqlitemagic/FastCur	12.0%	271.104
100 android/database/sqlite/SQLiteDatabase.rawQueryWithFactory (Landroid/database/sqlite/SQLiteI	0.1%	2.946
632 com/siimkinks/sqlitemagic/SqlManager.getReadableDatabase ()Landroid/database/sqlite/SQLiteD	0.0%	0.038
907 com/siimkinks/sqlitemagic/SqliteMagic.getSqlManager ()Lcom/siimkinks/sqlitemagic/SqlManage	0.0%	0.005
4 com/siimkinks/sqlitemagic/SqlUtil.getAllFromCursor (Ljava/lang/String;Lcom/siimkinks/sqlitemagic/FastCu	84.8%	1982.655
5 com/siimkinks/sqlitemagic/GeneratedClassesManager.allFromCursor (Ljava/lang/String;Lcom/siimkinks/sq	84.8%	1982.644
6 com/siimkinks/sqlitemagic/SqliteMagic_MixedData_Handler.allFromCursor (Lcom/siimkinks/sqlitemagic/Fa	84.8%	1982.614
Parents		
Children		
self	2.5%	49.593
7 com/siimkinks/sqlite/speedtests/data/SqliteMagic_MixedData_Dao.shallowObjectFromCursorPosition	94.4%	1870.742
51 com/siimkinks/sqlitemagic/FastCursor.moveToNext ()Z	1.8%	35.800
58 java/util/ArrayList.add (Ljava/lang/Object;)Z	0.7%	14.100
63 com/siimkinks/sqlitemagic/CompiledSelect$1.isUnsubscribed ()Z	0.6%	11.799
117 dalvik/system/VMDebug.startClassPrep ()V	0.0%	0.484
491 java/util/ArrayList.<init> (I)V	0.0%	0.089
940 com/siimkinks/sqlitemagic/util/MutableInt.<init> ()V	0.0%	0.004
954 com/siimkinks/sqlitemagic/FastCursor.getCount ()I	0.0%	0.003
(context switch)	0.0%	0.000
7 com/siimkinks/sqlite/speedtests/data/SqliteMagic_MixedData_Dao.shallowObjectFromCursorPosition (Lcor	80.0%	1870.742
Parents		
Children		
self	11.3%	211.212
8 com/siimkinks/sqlitemagic/FastCursor.isNull (I)Z	23.9%	446.990
9 com/siimkinks/sqlitemagic/FastCursor.getLong (I)J	23.7%	442.670
12 com/siimkinks/sqlitemagic/FastCursor.getDouble (I)D	17.8%	333.881
13 com/siimkinks/sqlitemagic/FastCursor.getString (I)Ljava/lang/String;	15.6%	291.431
28 java/lang/Long.valueOf (J)Ljava/lang/Long;	3.6%	67.017
29 java/lang/Double.valueOf (D)Ljava/lang/Double;	3.5%	65.417
62 com/siimkinks/sqlite/speedtests/data/MixedData.<init> ()V	0.6%	12.124
(context switch)	0.0%	0.000
8 com/siimkinks/sqlitemagic/FastCursor.isNull (I)Z	19.1%	446.990
9 com/siimkinks/sqlitemagic/FastCursor.getLong (I)J	18.9%	442.670
Parents		
Children		
self	21.3%	94.254
11 android/database/CursorWindow.getLong (II)J	78.7%	348.416
(context switch)	0.0%	0.000
10 android/database/CursorWindow.getType (II)I	15.0%	351.705
11 android/database/CursorWindow.getLong (II)J	14.9%	348.416
12 com/siimkinks/sqlitemagic/FastCursor.getDouble (I)D	14.3%	333.881
13 com/siimkinks/sqlitemagic/FastCursor.getString (I)Ljava/lang/String;	12.5%	291.431
14 com/siimkinks/sqlitemagic/SqliteMagicCursor.getFastCursor ()Lcom/siimkinks/sqlitemagic/FastCursor;	11.6%	271.104
15 com/siimkinks/sqlitemagic/FastCursor.from (Lcom/siimkinks/sqlitemagic/SqliteMagicCursor;)Lcom/siimk	11.5%	269.880
16 com/siimkinks/sqlitemagic/FastCursor.<init> (Lcom/siimkinks/sqlitemagic/SqliteMagicCursor;)V	11.5%	269.868
17 android/database/sqlite/SQLiteCursor.getCount ()I	11.5%	269.813
18 android/database/sqlite/SQLiteCursor.fillWindow (I)V	11.5%	269.804
19 android/database/sqlite/SQLiteQuery.fillWindow (Landroid/database/CursorWindow;IIZ)I	11.5%	269.194
20 android/database/sqlite/SQLiteSession.executeForCursorWindow (Ljava/lang/String;[Ljava/lang/Object;L	11.5%	268.999
21 android/database/sqlite/SQLiteConnection.executeForCursorWindow (Ljava/lang/String;[Ljava/lang/Obje	11.5%	267.856
22 android/database/sqlite/SQLiteConnection.nativeExecuteForCursorWindow (IIIIIZ)J	11.4%	266.856

DBFlow

Name	Incl Cpu %	Incl Cpu Time
0 (toplevel)	100.0%	2319.363
1 com/siimkinks/sqlite/speedtests/handlers/MixedDataHandler.queryAll ()Ljava/util/	96.1%	2229.030
2 com/raizlabs/android/dbflow/sql/language/Where.queryList ()Ljava/util/List;	96.1%	2228.993
3 com/raizlabs/android/dbflow/sql/language/BaseModelQueriable.queryList ()Ljava/	96.1%	2228.890
4 com/raizlabs/android/dbflow/sql/SqlUtils.queryList (Ljava/lang/Class;Ljava/lang/St	95.9%	2224.118
Parents		
Children		
self	0.0%	0.153
5 com/raizlabs/android/dbflow/sql/SqlUtils.convertToList (Ljava/lang/Class;Lar	99.5%	2213.771
78 android/database/sqlite/SQLiteDatabase.rawQuery (Ljava/lang/String;[Lja	0.3%	6.745
143 dalvik/system/VMDebug.startClassPrep ()V	0.1%	2.886
320 com/raizlabs/android/dbflow/config/BaseDatabaseDefinition.getWritablel	0.0%	0.422
334 com/raizlabs/android/dbflow/config/FlowManager.getDatabaseForTable (l	0.0%	0.121
894 java/lang/Class.isAssignableFrom (Ljava/lang/Class;)Z	0.0%	0.020
5 com/raizlabs/android/dbflow/sql/SqlUtils.convertToList (Ljava/lang/Class;Landroid/	95.4%	2213.771
Parents		
Children		
self	0.8%	17.779
6 com/siimkinks/sqlite/speedtests/data/MixedData$Adapter.loadFromCursor (	82.6%	1829.561
14 android/database/AbstractCursor.moveToFirst ()Z	11.5%	254.109
28 android/database/AbstractCursor.moveToNext ()Z	3.5%	76.432
56 com/siimkinks/sqlite/speedtests/data/MixedData$Adapter.newInstance ()L	1.3%	29.880
102 java/util/ArrayList.add (Ljava/lang/Object;)Z	0.2%	5.403
143 dalvik/system/VMDebug.startClassPrep ()V	0.0%	0.283
398 com/raizlabs/android/dbflow/config/FlowManager.getInstanceAdapter (L	0.0%	0.261
336 java/util/ArrayList.<init> ()V	0.0%	0.063
(context switch)	0.0%	0.000
6 com/siimkinks/sqlite/speedtests/data/MixedData$Adapter.loadFromCursor (Landr	78.9%	1829.561
7 com/siimkinks/sqlite/speedtests/data/MixedData$Adapter.loadFromCursor (Landr	78.5%	1821.602
Parents		
Children		
self	7.8%	141.486
8 android/database/sqlite/SQLiteCursor.getColumnIndex (Ljava/lang/String;)I	32.4%	589.872
9 android/database/AbstractWindowedCursor.isNull (I)Z	26.6%	484.474
20 android/database/AbstractWindowedCursor.getLong (I)J	13.3%	242.751
23 android/database/AbstractWindowedCursor.getDouble (I)D	10.0%	181.684
24 android/database/AbstractWindowedCursor.getString (I)Ljava/lang/String	7.6%	137.641
60 java/lang/Long.valueOf (J)Ljava/lang/Long;	1.2%	22.227
62 java/lang/Double.valueOf (D)Ljava/lang/Double;	1.2%	21.467
(context switch)	0.0%	0.000
8 android/database/sqlite/SQLiteCursor.getColumnIndex (Ljava/lang/String;)I	25.4%	589.872
9 android/database/AbstractWindowedCursor.isNull (I)Z	20.9%	484.474
Parents		
Children		
self	19.6%	95.075
21 android/database/CursorWindow.getType (II)I	47.9%	232.275
10 android/database/AbstractWindowedCursor.checkPosition ()V	32.4%	157.124
(context switch)	0.0%	0.000
10 android/database/AbstractWindowedCursor.checkPosition ()V	14.4%	333.726
11 android/database/sqlite/SQLiteCursor.getCount ()I	14.0%	324.808
12 android/database/AbstractCursor.moveToPosition (I)Z	13.9%	322.623
13 java/util/HashMap.get (Ljava/lang/Object;)Ljava/lang/Object;	11.6%	268.517
14 android/database/AbstractCursor.moveToFirst ()Z	11.0%	254.109
15 android/database/sqlite/SQLiteCursor.fillWindow (I)V	11.0%	253.993
16 android/database/sqlite/SQLiteQuery.fillWindow (Landroid/database/CursorWind	10.9%	252.257
17 android/database/sqlite/SQLiteSession.executeForCursorWindow (Ljava/lang/Str	10.8%	251.560
18 android/database/sqlite/SQLiteConnection.executeForCursorWindow (Ljava/lang	10.8%	250.079
19 android/database/sqlite/SQLiteConnection.nativeExecuteForCursorWindow (IIIII:	10.7%	249.051
20 android/database/AbstractWindowedCursor.getLong (I)J	10.5%	242.751

greenDAO

Name	Incl Cpu %	Incl Cpu Time
0 (toplevel)	100.0%	2292.230
1 com/siimkinks/sqlite/speedtests/handlers/MixedDataHandler.queryAll ()Ljav	96.6%	2213.730
2 de/greenrobot/dao/AbstractDao.loadAll ()Ljava/util/List;	96.3%	2207.899
3 de/greenrobot/dao/AbstractDao.loadAllAndCloseCursor (Landroid/database/	95.6%	2192.370
4 de/greenrobot/dao/AbstractDao.loadAllFromCursor (Landroid/database/Cur:	95.6%	2192.357
Parents		
Children		
self	0.0%	0.311
5 de/greenrobot/dao/AbstractDao.loadAllUnlockOnWindowBounds (Lan	88.9%	1948.655
12 android/database/sqlite/SQLiteCursor.getCount ()I	11.0%	241.798
231 dalvik/system/VMDebug.startClassPrep ()V	0.0%	0.746
385 java/lang/StringBuilder.append (I)Ljava/lang/StringBuilder;	0.0%	0.244
470 de/greenrobot/dao/DaoLog.d (Ljava/lang/String;)I	0.0%	0.139
471 android/database/AbstractCursor.moveToFirst ()Z	0.0%	0.138
107 java/lang/StringBuilder.append (Ljava/lang/String;)Ljava/lang/Strir	0.0%	0.071
619 de/greenrobot/dao/identityscope/IdentityScopeLong.reserveRoom	0.0%	0.068
649 java/util/ArrayList.<init> (I)V	0.0%	0.061
683 de/greenrobot/dao/identityscope/IdentityScopeLong.lock ()V	0.0%	0.050
43 android/database/CursorWindow.getNumRows ()I	0.0%	0.043
534 java/lang/StringBuilder.toString ()Ljava/lang/String;	0.0%	0.017
940 java/lang/StringBuilder.<init> ()V	0.0%	0.012
991 android/database/AbstractWindowedCursor.getWindow ()Landroid	0.0%	0.004
5 de/greenrobot/dao/AbstractDao.loadAllUnlockOnWindowBounds (Landroid/	85.0%	1948.655
Parents		
Children		
self	0.9%	17.450
6 de/greenrobot/dao/AbstractDao.loadCurrent (Landroid/database/Cur:	87.4%	1704.071
22 android/database/AbstractCursor.moveToNext ()Z	11.3%	220.704
93 java/util/ArrayList.add (Ljava/lang/Object;)Z	0.3%	6.406
43 android/database/CursorWindow.getNumRows ()I	0.0%	0.021
67 android/database/CursorWindow.getStartPosition ()I	0.0%	0.003
(context switch)	0.0%	0.000
6 de/greenrobot/dao/AbstractDao.loadCurrent (Landroid/database/Cursor;IZ)	74.3%	1704.071
Parents		
Children		
self	1.9%	33.017
7 com/siimkinks/sqlite/speedtests/data/MixedDataDao.readEntity (Lanc	89.0%	1517.343
10 android/database/AbstractWindowedCursor.getLong (I)J	4.7%	80.139
36 de/greenrobot/dao/identityscope/IdentityScopeLong.put2NoLock (J	3.1%	52.279
61 de/greenrobot/dao/identityscope/IdentityScopeLong.get2NoLock (J	1.0%	16.281
113 de/greenrobot/dao/AbstractDao.attachEntity (Ljava/lang/Object;)V	0.3%	5.012
(context switch)	0.0%	0.000
7 com/siimkinks/sqlite/speedtests/data/MixedDataDao.readEntity (Landroid/d	66.2%	1517.343
8 com/siimkinks/sqlite/speedtests/data/MixedDataDao.readEntity (Landroid/d	65.7%	1506.854
Parents		
Children		
self	7.8%	117.961
9 android/database/AbstractWindowedCursor.isNull (I)Z	37.2%	560.541
10 android/database/AbstractWindowedCursor.getLong (I)J	21.2%	319.174
16 android/database/AbstractWindowedCursor.getDouble (I)D	15.9%	240.175
24 android/database/AbstractWindowedCursor.getString (I)Ljava/lang/	11.9%	179.517
29 java/lang/Long.valueOf (J)Ljava/lang/Long;	3.7%	55.873
55 java/lang/Double.valueOf (D)Ljava/lang/Double;	1.8%	27.760
99 com/siimkinks/sqlite/speedtests/data/MixedData.<init> (Ljava/lang/	0.4%	5.853
(context switch)	0.0%	0.000
9 android/database/AbstractWindowedCursor.isNull (I)Z	24.5%	560.541
10 android/database/AbstractWindowedCursor.getLong (I)J	20.9%	479.907
Parents		
Children		
self	19.5%	93.540
21 android/database/CursorWindow.getLong (II)J	48.0%	230.205
11 android/database/AbstractWindowedCursor.checkPosition ()V	32.5%	156.162
(context switch)	0.0%	0.000
11 android/database/AbstractWindowedCursor.checkPosition ()V	20.4%	468.076
12 android/database/sqlite/SQLiteCursor.getCount ()I	14.9%	340.819
13 android/database/AbstractCursor.checkPosition ()V	12.3%	281.653

Joonis 18: SqliteMagic, DBFlow ja greenDAO kõikide andmete pärimise operatsiooni meetodite jälgimisvaade.

B Lisa: mälueraldusvaated

Joonis 19: Sisestamisoperatsiooni allokeeritud objektide vaade.

Joonis 20: Uuendamisoperatsiooni allokeeritud objektide vaade.

Method (SqliteMagic)	Count	Size
android	41006 (74.35%)	1360176 (69.90%)
database	41000 (74.34%)	1360000 (69.89%)
sqlite	30000 (54.39%)	1008000 (51.80%)
SQLiteConnection	27000 (48.95%)	896000 (46.05%)
SQLiteDirectCursorDriver	1000 (1.81%)	48000 (2.47%)
SQLiteDatabase	1000 (1.81%)	32000 (1.64%)
SQLiteProgram	1000 (1.81%)	32000 (1.64%)
CursorWindow	4000 (7.25%)	160000 (8.22%)
AbstractCursor	4000 (7.25%)	96000 (4.93%)
Observable	2000 (3.63%)	64000 (3.29%)
AbstractWindowedCursor	1000 (1.81%)	32000 (1.64%)
text	2 (0.00%)	64 (0.00%)
graphics	2 (0.00%)	64 (0.00%)
ddm	2 (0.00%)	48 (0.00%)
java	10029 (18.18%)	337056 (17.32%)
lang	8010 (14.52%)	240176 (12.34%)
String	3001 (5.44%)	96032 (4.94%)
ref	2000 (3.63%)	96000 (4.93%)
FinalizerReference	2000 (3.63%)	96000 (4.93%)
Integer	1009 (1.83%)	16144 (0.83%)
Long	1000 (1.81%)	16000 (0.82%)
Double	1000 (1.81%)	16000 (0.82%)
util	2010 (3.64%)	96336 (4.95%)
WeakHashMap	2000 (3.63%)	96000 (4.93%)
LinkedList	6 (0.01%)	192 (0.01%)
HashMap	3 (0.01%)	112 (0.01%)
LinkedHashMap	1 (0.00%)	32 (0.00%)
nio	9 (0.02%)	544 (0.03%)
com	4087 (7.41%)	247800 (12.74%)
siimkinks	4087 (7.41%)	247800 (12.74%)
sqlitemagic	3085 (5.59%)	167720 (8.62%)
DbHelper	1017 (1.84%)	114400 (5.88%)
FastCursor	1000 (1.81%)	32000 (1.64%)
SqliteMagic_MixedData_Handler	1008 (1.83%)	16784 (0.86%)
SqliteMagicCursor	60 (0.11%)	4536 (0.23%)
sqlite	1002 (1.82%)	80080 (4.12%)
speedtests	1002 (1.82%)	80080 (4.12%)
data	1000 (1.81%)	80000 (4.11%)
SqliteMagic_MixedData_Dao	1000 (1.81%)	80000 (4.11%)
activity	2 (0.00%)	80 (0.00%)
rx	14 (0.03%)	320 (0.02%)
org	9 (0.02%)	288 (0.01%)
< Unknown >	9 (0.02%)	144 (0.01%)

Method (DBFlow)	Count	Size
android	31650 (48.29%)	1095232 (46.03%)
database	31648 (48.29%)	1095168 (46.02%)
sqlite	23552 (35.94%)	836096 (35.14%)
SQLiteConnection	19872 (30.32%)	647680 (27.22%)
SQLiteDirectCursorDriver	1472 (2.25%)	105984 (4.45%)
SQLiteCursor	736 (1.12%)	35328 (1.48%)
SQLiteDatabase	736 (1.12%)	23552 (0.99%)
SQLiteProgram	736 (1.12%)	23552 (0.99%)
CursorWindow	2944 (4.49%)	117760 (4.95%)
AbstractCursor	2944 (4.49%)	70656 (2.97%)
Observable	1472 (2.25%)	47104 (1.98%)
AbstractWindowedCursor	736 (1.12%)	23552 (0.99%)
text	1 (0.00%)	32 (0.00%)
graphics	1 (0.00%)	32 (0.00%)
java	26509 (40.45%)	1048480 (44.06%)
lang	13985 (21.34%)	588592 (24.73%)
AbstractStringBuilder	8089 (12.34%)	411824 (17.31%)
String	2208 (3.37%)	70656 (2.97%)
ref	1472 (2.25%)	70656 (2.97%)
FinalizerReference	1472 (2.25%)	70656 (2.97%)
Integer	744 (1.14%)	11904 (0.50%)
Long	736 (1.12%)	11776 (0.49%)
Double	736 (1.12%)	11776 (0.49%)
util	12516 (19.10%)	459392 (19.31%)
HashMap	10305 (15.72%)	365088 (15.34%)
WeakHashMap	1472 (2.25%)	70656 (2.97%)
ArrayList	735 (1.12%)	23520 (0.99%)
LinkedList	4 (0.01%)	128 (0.01%)
nio	8 (0.01%)	496 (0.02%)
com	7354 (11.22%)	235376 (9.89%)
raizlabs	6618 (10.10%)	164720 (6.92%)
android	6618 (10.10%)	164720 (6.92%)
dbflow	6618 (10.10%)	164720 (6.92%)
sql	6618 (10.10%)	164720 (6.92%)
QueryBuilder	2205 (3.36%)	70560 (2.97%)
builder	1472 (2.25%)	47104 (1.98%)
language	2941 (4.49%)	47056 (1.98%)
siimkinks	736 (1.12%)	70656 (2.97%)
sqlite	736 (1.12%)	70656 (2.97%)
speedtests	736 (1.12%)	70656 (2.97%)
data	736 (1.12%)	70656 (2.97%)
MixedData	736 (1.12%)	70656 (2.97%)
org	7 (0.01%)	224 (0.01%)
rx	7 (0.01%)	160 (0.01%)
< Unknown >	8 (0.01%)	128 (0.01%)

Method (greenDAO)	Count	Size
java	19055 (29.08%)	1608272 (49.43%)
lang	13713 (20.92%)	1412736 (43.42%)
AbstractStringBuilder	6849 (10.45%)	1229776 (37.79%)
ref	1524 (2.33%)	73152 (2.25%)
FinalizerReference	1524 (2.33%)	73152 (2.25%)
String	1523 (2.32%)	48736 (1.50%)
Long	1524 (2.33%)	24384 (0.75%)
Integer	770 (1.17%)	12320 (0.38%)
Double	762 (1.16%)	12192 (0.37%)
Object	761 (1.16%)	12176 (0.37%)
util	5334 (8.14%)	195040 (5.99%)
ArrayList	3806 (5.81%)	121776 (3.74%)
WeakHashMap	1523 (2.32%)	73104 (2.25%)
LinkedList	4 (0.01%)	128 (0.00%)
HashMap	1 (0.00%)	32 (0.00%)
nio	8 (0.01%)	496 (0.02%)
android	33518 (51.15%)	1170032 (35.96%)
database	32755 (49.98%)	1121264 (34.46%)
sqlite	24373 (37.19%)	853040 (26.22%)
SQLiteConnection	20566 (31.38%)	682464 (20.97%)
SQLiteDirectCursorDriver	1523 (2.32%)	109680 (3.37%)
SQLiteProgram	1523 (2.32%)	36544 (1.12%)
SQLiteDatabase	761 (1.16%)	24352 (0.75%)
CursorWindow	3048 (4.65%)	121920 (3.75%)
AbstractCursor	3048 (4.65%)	73152 (2.25%)
Observable	1524 (2.33%)	48768 (1.50%)
AbstractWindowedCursor	762 (1.16%)	24384 (0.75%)
util	761 (1.16%)	48704 (1.50%)
text	1 (0.00%)	32 (0.00%)
graphics	1 (0.00%)	32 (0.00%)
de	11417 (17.42%)	365344 (11.23%)
greenrobot	11417 (17.42%)	365344 (11.23%)
dao	11417 (17.42%)	365344 (11.23%)
query	9132 (13.93%)	292224 (8.98%)
QueryBuilder	3805 (5.81%)	133936 (4.12%)
Query	1522 (2.32%)	60880 (1.87%)
AbstractQueryData	1522 (2.32%)	48704 (1.50%)
WhereCollector	761 (1.16%)	24352 (0.75%)
AbstractQuery	1522 (2.32%)	24352 (0.75%)
AbstractDao	1524 (2.33%)	48768 (1.50%)
internal	761 (1.16%)	24352 (0.75%)
SqlUtils	761 (1.16%)	24352 (0.75%)
com	762 (1.16%)	60960 (1.87%)
siimkinks	762 (1.16%)	60960 (1.87%)
sqlite	762 (1.16%)	60960 (1.87%)
speedtests	762 (1.16%)	60960 (1.87%)
data	762 (1.16%)	60960 (1.87%)
MixedDataDao	762 (1.16%)	60960 (1.87%)
dalvik	761 (1.16%)	48704 (1.50%)
org	7 (0.01%)	224 (0.01%)
rx	7 (0.01%)	160 (0.00%)
< Unknown >	8 (0.01%)	128 (0.00%)

Method (Sugar ORM)	Count	Size
java	29109 (44.42%)	1604640 (56.45%)
lang	18585 (28.36%)	961344 (33.82%)
AbstractStringBuilder	7145 (10.90%)	405168 (14.25%)
String	7658 (11.69%)	486512 (17.12%)
reflect	1544 (2.36%)	28352 (1.00%)
AccessibleObject	1487 (2.27%)	23792 (0.84%)
Constructor	57 (0.09%)	4560 (0.16%)
Class	1544 (2.36%)	24704 (0.87%)
Double	172 (0.26%)	2752 (0.10%)
Long	287 (0.44%)	4592 (0.16%)
ref	172 (0.26%)	8256 (0.29%)
Integer	63 (0.10%)	1008 (0.04%)
nio	9322 (14.22%)	596592 (20.99%)
DirectByteBuffer	9316 (14.22%)	596224 (20.98%)
ByteBuffer	5 (0.01%)	320 (0.01%)
CharBuffer	1 (0.00%)	48 (0.00%)
util	1202 (1.83%)	46704 (1.64%)
com	32413 (49.46%)	1106720 (38.94%)
android	31270 (47.71%)	1068320 (37.59%)
dex	31268 (47.71%)	1068272 (37.58%)
Dex	23036 (35.15%)	848752 (29.86%)
Annotation	2744 (4.19%)	87808 (3.09%)
EncodedValue	2744 (4.19%)	43904 (1.54%)
Mutf8	2744 (4.19%)	87808 (3.09%)
internal	2 (0.00%)	48 (0.00%)
orm	1029 (1.57%)	34752 (1.22%)
util	858 (1.31%)	31104 (1.09%)
NamingHelper	801 (1.22%)	25632 (0.90%)
SugarCursorFactory	57 (0.09%)	5472 (0.19%)
SugarRecord	171 (0.26%)	3648 (0.13%)
google	114 (0.17%)	3648 (0.13%)
libcore	1542 (2.35%)	49344 (1.74%)
android	2453 (3.74%)	81232 (2.86%)
database	2451 (3.74%)	81168 (2.86%)
sqlite	1824 (2.78%)	61104 (2.15%)
SQLiteQueryBuilder	57 (0.09%)	1824 (0.06%)
SQLiteDatabase	57 (0.09%)	1824 (0.06%)
SQLiteDirectCursorDriver	57 (0.09%)	2736 (0.10%)
SQLiteProgram	57 (0.09%)	1824 (0.06%)
SQLiteConnection	1539 (2.35%)	50160 (1.76%)
SQLiteCursor	57 (0.09%)	2736 (0.10%)
AbstractCursor	228 (0.35%)	5472 (0.19%)
Observable	114 (0.17%)	3648 (0.13%)
AbstractWindowedCursor	57 (0.09%)	1824 (0.06%)
CursorWindow	228 (0.35%)	9120 (0.32%)
text	1 (0.00%)	32 (0.00%)
graphics	1 (0.00%)	32 (0.00%)

Joonis 21: Esimese kirje pärimise allokeeritud objektide vaade.

Method (SqliteMagic)	Count	Size
android	37428 (57.11%)	1497104 (58.80%)
database	37426 (57.11%)	1497040 (58.80%)
CursorWindow	37426 (57.11%)	1497040 (58.80%)
text	1 (0.00%)	32 (0.00%)
graphics	1 (0.00%)	32 (0.00%)
com	9356 (14.28%)	748480 (29.40%)
siimkinks	9356 (14.28%)	748480 (29.40%)
sqlite	9356 (14.28%)	748480 (29.40%)
speedtests	9356 (14.28%)	748480 (29.40%)
data	9356 (14.28%)	748480 (29.40%)
SqliteMagic_MixedData_Dao	9356 (14.28%)	748480 (29.40%)
java	18733 (28.58%)	300128 (11.79%)
lang	18721 (28.57%)	299552 (11.77%)
Long	9357 (14.28%)	149712 (5.88%)
Double	9357 (14.28%)	149712 (5.88%)
Integer	6 (0.01%)	96 (0.00%)
String	1 (0.00%)	32 (0.00%)
nio	6 (0.01%)	368 (0.01%)
ByteBuffer	5 (0.01%)	320 (0.01%)
CharBuffer	1 (0.00%)	48 (0.00%)
util	6 (0.01%)	208 (0.01%)
LinkedList	4 (0.01%)	128 (0.01%)
WeakHashMap	1 (0.00%)	48 (0.00%)
HashMap	1 (0.00%)	32 (0.00%)
org	5 (0.01%)	160 (0.01%)
rx	7 (0.01%)	160 (0.01%)
< Unknown >	6 (0.01%)	96 (0.00%)

Method (greenDAO)	Count	Size
android	29110 (44.42%)	1164384 (52.60%)
database	29108 (44.42%)	1164320 (52.60%)
CursorWindow	29108 (44.42%)	1164320 (52.60%)
text	1 (0.00%)	32 (0.00%)
graphics	1 (0.00%)	32 (0.00%)
com	7277 (11.10%)	582160 (26.30%)
siimkinks	7277 (11.10%)	582160 (26.30%)
sqlite	7277 (11.10%)	582160 (26.30%)
speedtests	7277 (11.10%)	582160 (26.30%)
data	7277 (11.10%)	582160 (26.30%)
MixedDataDao	7277 (11.10%)	582160 (26.30%)
java	29128 (44.45%)	466496 (21.08%)
lang	29115 (44.43%)	465856 (21.05%)
Long	21830 (33.31%)	349280 (15.78%)
Double	7277 (11.10%)	116432 (5.26%)
Integer	7 (0.01%)	112 (0.01%)
String	1 (0.00%)	32 (0.00%)
nio	7 (0.01%)	432 (0.02%)
ByteBuffer	6 (0.01%)	384 (0.02%)
CharBuffer	1 (0.00%)	48 (0.00%)
util	6 (0.01%)	208 (0.01%)
LinkedList	4 (0.01%)	128 (0.01%)
WeakHashMap	1 (0.00%)	48 (0.00%)
HashMap	1 (0.00%)	32 (0.00%)
org	6 (0.01%)	192 (0.01%)
rx	7 (0.01%)	160 (0.01%)
< Unknown >	7 (0.01%)	112 (0.01%)

Method (DBFlow)	Count	Size
android	37425 (57.11%)	1496976 (53.11%)
database	37423 (57.10%)	1496912 (53.11%)
CursorWindow	37423 (57.10%)	1496912 (53.11%)
text	1 (0.00%)	32 (0.00%)
graphics	1 (0.00%)	32 (0.00%)
com	9355 (14.27%)	898080 (31.86%)
siimkinks	9355 (14.27%)	898080 (31.86%)
sqlite	9355 (14.27%)	898080 (31.86%)
speedtests	9355 (14.27%)	898080 (31.86%)
data	9355 (14.27%)	898080 (31.86%)
MixedData	9355 (14.27%)	898080 (31.86%)
java	18735 (28.59%)	423056 (15.01%)
lang	18720 (28.56%)	299536 (10.63%)
Long	9356 (14.28%)	149696 (5.31%)
Double	9356 (14.28%)	149696 (5.31%)
Integer	7 (0.01%)	112 (0.00%)
String	1 (0.00%)	32 (0.00%)
util	8 (0.01%)	123088 (4.37%)
ArrayList	2 (0.00%)	122880 (4.36%)
java.lang.Object[]	1 (0.00%)	73728 (2.62%)
java.lang.Object[]	1 (0.00%)	49152 (1.74%)
LinkedList	4 (0.01%)	128 (0.00%)
WeakHashMap	1 (0.00%)	48 (0.00%)
HashMap	1 (0.00%)	32 (0.00%)
nio	7 (0.01%)	432 (0.02%)
ByteBuffer	6 (0.01%)	384 (0.01%)
CharBuffer	1 (0.00%)	48 (0.00%)
org	6 (0.01%)	192 (0.01%)
rx	7 (0.01%)	160 (0.01%)
< Unknown >	7 (0.01%)	112 (0.00%)

Method (Sugar ORM)	Count	Size
java	29252 (44.64%)	1614640 (56.69%)
lang	19331 (29.50%)	984704 (34.57%)
String	7907 (12.07%)	506048 (17.77%)
AbstractStringBuilder	7461 (11.38%)	411184 (14.44%)
reflect	1720 (2.62%)	31552 (1.11%)
AccessibleObject	1657 (2.53%)	26512 (0.93%)
Constructor	63 (0.10%)	5040 (0.18%)
Class	1720 (2.62%)	27520 (0.97%)
Long	319 (0.49%)	5104 (0.18%)
Double	192 (0.29%)	3072 (0.11%)
Integer	11 (0.02%)	176 (0.01%)
ref	1 (0.00%)	48 (0.00%)
nio	9766 (14.90%)	624992 (21.94%)
DirectByteBuffer	9754 (14.88%)	624256 (21.92%)
ByteBuffer	10 (0.02%)	640 (0.02%)
CharBuffer	2 (0.00%)	96 (0.00%)
util	155 (0.24%)	4944 (0.17%)
Collections	63 (0.10%)	2016 (0.07%)
ArrayList	63 (0.10%)	2016 (0.07%)
LinkedList	22 (0.03%)	704 (0.02%)
HashMap	6 (0.01%)	176 (0.01%)
concurrent	1 (0.00%)	32 (0.00%)
com	34304 (52.34%)	1168112 (41.01%)
android	33221 (50.69%)	1135472 (39.87%)
dex	33219 (50.69%)	1135424 (39.86%)
Dex	24420 (37.26%)	900784 (31.63%)
Annotation	2933 (4.48%)	93856 (3.30%)
Mutf8	2933 (4.48%)	93856 (3.30%)
EncodedValue	2933 (4.48%)	46928 (1.65%)
internal	2 (0.00%)	48 (0.00%)
orm	955 (1.46%)	28544 (1.00%)
util	829 (1.26%)	26528 (0.93%)
NamingHelper	829 (1.26%)	26528 (0.93%)
SugarRecord	126 (0.19%)	2016 (0.07%)
google	126 (0.19%)	4032 (0.14%)
siimkinks	2 (0.00%)	64 (0.00%)
libcore	1658 (2.53%)	53056 (1.86%)
reflect	1658 (2.53%)	53056 (1.86%)
InternalNames	1658 (2.53%)	53056 (1.86%)
android	260 (0.40%)	10368 (0.36%)
rx	40 (0.06%)	1504 (0.05%)
org	10 (0.02%)	320 (0.01%)
< Unknown >	11 (0.02%)	176 (0.01%)

Joonis 22: Kõikide kirjete pärimise allokeeritud objektide vaade.