TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Informaatikainstituut
Tarkvaratehnika õppetool

WhiteDB C# API loomine ja jõudluse analüüs

bakalaureusetöö

Üliõpilane: Andrei Reinus
Üliõpilaskood: 111881
Juhendaja: Martin Rebane

Tallinn
2015

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

---------- ----------

(*kuupäev*) (*allkiri*)

Annotatsioon

Käesoleva töö eesmärk on uurida, kuidas oleks võimalik WhiteDB andmebaasi kasutada Microsoft.NET raamistikul.

Töös uuritakse, millised on kiirusekaod on .NETi või Pythoni kasutamisel võrreldes C APIga ning kuidas paigutub WhiteDB MongoDB kõrvale.

Töö tulemusel on WhiteDB andmebaasi võimalik kasutada .NETi arendustes omamata täpseid teadmisi C programmeerimiskeelest. Tarkvaraarendajatel ja arhitektidel on alusinformatsiooni WhiteDB jõudlusest ning saavad teha faktipõhiseid otsuseid antud tarkvara kasutamisel oma projektides.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 36 leheküljel, 5 peatükki, 10 joonist, 9 tabelit.

Abstract

Aim of this thesis is find ways to introduce WhiteDB database into .NET framework.

Author also intends to find out how much overhead is C# API using compared to C API and Python API and to ompare WhiteDB against MongoDB in few common use-cases.

WhiteDB has now C# API witch is comparable in speed with C API. MongoDB is slower compared to WhiteDB but has more features.

The thesis is in Estonian and contains 36 pages of text, 5 chapters, 10 figures, 9 tables.

Lühendite ja mõistete sõnastik

ACID	***Atomicity, Consistency, Isolation, Durability*** atomaarsus, konsistentsus, isoleeritus, püsivus Andmebaasihaldurite puhul tehingutöötluse põhikarakteristikud. Ilma nendeta ei saa tagada andmebaasihalduri töökindlust. [1]
API	***Application Programming Interface*** Rakendusprogrammiga määratud reeglistik, mille alusel rakendusprogramm kasutab operatsioonisüsteemi või teise rakendusprogrammi teenuseid. [1]
C#	Programmeerimiskeel, mida kasutatakse .NET raamistikule programmide kirjutamisel.
MongoDB	Enamlevinud NoSQL dokumendi andmebaas, mida arendab peamiselt MongoDB Inc.
Mono	Vabavaraline implementatsioon .NETi raamistikus tehtud programmide töötamiseks väljaspool Microsofti pool pakutud operatsioonisüsteeme, peamiselt Linuxi operatsioonisüsteemil
NewSQL	Kaasaegse tehnoloogia alusel arendatavad relatsioonilised andmebaasid, mis püüavad saavutada samaväärset jõudlust NoSQL lahendustega hülgamata ACID põhimõtteid.
.NET	***.NET Framework*** Microsofti poolt arendatav tarkvara raamistik, mis võimaldab mitmes erinevas programmeerimiskeeles rakendusi arendada samale virtuaalmasinale.
NoSQL	***Not only SQL*** Alternatiiv relatsioonilistele andmebaasidele, kus kiiruse ja jõudluse huvides hoitakse andmeid denormaliseeritud kujul.

P/Invoke	***Platform Invocation Service*** .NETi raamistiku tehnika, millega .NETi rakendused saavad kasutada C programmeerimiskeeles kirjutatud ning kompileeritud mooduleid.
Python	Interpreteeritav objektorienteeritud programmikeel, mille lõi Guido van Rossum 1990.a. [1]
POCO	***Plain old CLR object*** Klassi defineerimine ilma kindla raamistikuga sidumata.
SQL	***Structured Query Language*** Enimkasutatav päringukeel, mida toetavad kõik klient-server keskkonnale projekteeritud relatsioonandmebaasid. Päringukeeled kujutavad endast reeglite kogumit, mille alusel konstrueeritakse päringuid andmete otsimiseks andmebaasist [1]
WhiteDB	Eesti teadlaste poolt arendatav vabavaraline NoSQLi andmebaas

Jooniste nimekiri

Tabelite nimekiri

Sisukord

1. Sissejuhatus

Arvutite ajaloo jooksul on muutmälu alati olnud võrreldes kõvaketastega oluliselt kiirem, kuid tihtipeale jääb muutmälust väheseks kogu vajaliku informatsiooni talletamiseks. Tänasel päeval on mälu hind jõudnud piisavalt madalale, et hoida enamuse vajalikust informatsioonist muutmälus. Teisest küljest on muutunud nõuded internetirakendustele, kus kasutajad ei tekita enam sadu päringuid sekundis, vaid koormus võib küündida ka kümnete miljoniteni päringuteni sekundis.

Traditsioonilised andmebaasid (Oracle, Microsoft SQL Server, IBM DB2, PostgreSQL, MySQL) lähtuvad töötamisel ACID (Atomic, Consistent, Isolated, Durable) põhimõttest, kuid jõudlusnäitajad ei vasta rakenduses soovitule. Seetõttu on levimas trend, kus rakenduse tööks vajalikud andmed on hoiustatud serveri muutmälus.

1.1 Taust ja probleem

Minu huvialaks on leida võimalusi, kuidas kindlat tüüpi probleemi lahendamiseks kasutada võimalikult parimat lahendust. Selleks, et häid lahendusi välja pakkuda, peab teadma võimalustest, mida erinevad tarkvarad pakuvad. Eestis toimivates rakendustes on raske näha olukorda, kus kasutatavad andmed ei suudaks paikneda mälus.

Relatsiooniliste andmebaaside arendajatest osad on lisanud oma toodetesse mälupõhiseid andmete salvestamist võimaldavaid võimalusi kuid nende hind on aukartust äratavalt kõrge.

- Oracle-l on olemas TimesTen lisa oma põhitootele. Antud toode maksab arvestuslikult mitukümmend tuhat dollarit protsessori kohta. [1]

- Microsoft lisas SQL Server 2014 versiooni mälus paiknevate tabelite võimaluse, kuid see on saadaval ainult Enterprise litsentsiga ja maksab üle 14000 dollarit protsessori tuuma kohta. [2]

Teine grupp tooteid grupeeritakse NewSQL nimetuse alla [3]. Need on andmebaasid, mis kasutavad relatsioonilist andmemudelit ja ACID põhimõtteid, kuid kasutavad moodsaid

arhitektuurilisi ja tarkvaraarenduse võimalusi ning teadmisi. Antud uurimus ei uuri NewSQL andmebaase, kuid on väga huvitav uurimisobjekt järgmistele.

NoSQL on kolmas grupp tooteid, mis on hüljanud ACID põhimõtted andmete töötlemisel ning hoiustavad andmeid enamjaolt de-normaliseeritud kujul. Laias laastus jagunevad NoSQL andmebaasid:

- Document – Andmebaas teab iga kirje kohta ka temas esinevate väljade infot. Andmeid saab pärida iga andmevälja järgi.
- Key/Value – Kõik andmebaasis olevad väljad on defineeritud võti ja temale vastava välja järgi.
- Graph – Andmed on seotud andmebaasis lähtudes graafiteooria põhimõtetest.

WhiteDB [4] on TTÜ professori Tanel Tammeri algatatud projekt, mis liigitub NoSQL andmebaaside alla ja kasutab oma tööks jagatud mälu. Andmebaasi võib kasutada kui Document andmebaasina või siis Graph andmebaasina.

WhiteDB funktsionaalsust ei ole hetkel võimalik .NETi raamistikul kasutada, kuid vajadus mälupõhiste andmebaaside järgi on olemas. Samuti ei ole teada, kui efektiivne on loodav C# API võrreldes teiste APIdega. Tarkvara arhitektidel ja arendajatel puudub info, mille põhjal otsustada WhiteDB sobivuse üle projektis.

1.2 Ülesande püstitus

- Arendada avalik API, et .NETi rakendused saaksid kasutada WhiteDB funktsionaalsust.
- Uurida, kas ja kui palju kaotab WhiteDB oma kiiruses kasutades .NETi raamistiku võrreldes C ja Pythoniga.
- Uurida, kuidas WhiteDB on kasutatav MongoDB asemel.

1.3 Metoodika ja tööprotsess

API arendamisel lähtuti järgmistest reeglitest:

1. Kogu programmi kood peab olema avalik ja teiste poolt muudetav. WhiteDB on ise vabavaraline tarkvara ning kõikidele avatud arendusmudel aitab kaasa tarkvara arengule ning levimisele. C# API ei tohiks siinkohal olla erand.

2. Võimalikult palju koodist peab olema kaetud ühiktestidega. Kvaliteet on hea tarkvara alus ja ühiktestimine on parimaid viise selle tagamiseks.

3. API peab toetama Windowsi ja Linux operatsioonisüsteeme. Microsoft.NET ei ole ainult Windowsi keskne arendusplatvorm. Alates 2014 aastast on Microsoft järjest enam liikunud avatud lähtekoodiga tarkvara suunas ning suur osa .NET raamistiku komponentide programmikoodist on avalikult kättesaadavad [5].

Arenduse algfaasis võtsin ühendust ka kahe peamise WhiteDB arendajaga ning tutvustasin oma ideed. Tanel Tammet ja Priit Järv olid väga toetavad ning hilisemalt üles kerkinud küsimustele sain vastused.

Arenduse käigus kerkis üles probleem, et kuna WhiteDB kasutab oma tööks jagatud mälu siis Windowsi operatsioonisüsteemis peab selle jaoks kasutama mälus asuvaid faile. Failide poolt kasutusel olev mälu vabastatakse aga alles siis, kui kõik programmid, mis seda faili kasutasid on lõpetanud oma töö. Olukorras, kus programmi eesmärk on luua, täita ja kustutada andmebaas ning seda protsessi korrata korduvalt nagu tehakse punktis 3.3 siis Windowsi operatsioonisüsteemis kasutatakse protsessiga seotud mälu ning Linuxi operatsioonisüsteemis jagatud mälu.

1.4 Ülevaade tööst

Disainides C# API funktsionaalsust ja lahendust sai selgeks vajadus esmaselt lahendada suhtlus C APIga ning siis saab edasi teha objektorienteeritud kõrgema taseme funktsionaalsust.

Jõudlustestide läbiviimiseks oli vaja esmalt kirjutada liidestus C APIga, mis oli aluseks ülejäänud funktsionaalsuse loomisel. Seejärel sai programmeerida kolmes erinevas programmeerimiskeeles jõudlustestide programmid, et võrrelda C, C# ja Pythoni API omavahelist jõudluserinevust samade ülesannete täitmisel. Võrdluseks MongoDB-ga realiseeriti sama funktsionaalsus nii MongoDB, kui ka WhiteDB APIga ning võrreldi kiiruse erinevust.

2. WhiteDB C# API

Aluseks hilisemate jõudlustestide tegemiseks puudus .NETi raamistikus võimalus ühenduda WhiteDB andmebaasiga. Antud lõputöö raames sai selline funktsionaalsus arendatud ning avaldatud vabavaralise tarkvarana. Tarkvara lähekood on saadava internetis aadressil https://github.com/andreireinus/whitedb-csharp.

WhiteDB C# API jaguneb mitmeks osaks:

- C funktsioonide definitsioonid C# keeles (NativeAPI ja IndexAPI klassid)
- Kõrgema taseme kiht NativeAPI kasutamiseks (DataContext ja DataRecord klassid)
- *Generic* klassid POCO objektide käitlemisel (DataContext<T>, ModelBuilder<T>, ModelBinder<T> klassid)
- .h failidest C funktsioonide automaatne genereerimine C# klassideks (WhiteDb.Generator nimeruum)

.NETi raamistik kasutab C APIdega suhtlemiseks tehnikat nimetusega „Platform Invocation Service“ (lühendatult P/Invoke). P/Invoke kasutamiseks tuleb C funktsioon defineerida C# poolel kindlate parameetritega ja siis on võimalik neid .NETi raamistikus kasutada.

Näiteks on C keeles .h failis defineeritud funktsioon:

```
int wg_detach_database(void* dbase);
```

Siis C# peab defineerima vastava funktsiooni järgnevalt

```
class NativeAPI
{
  [DllImport("wgdb.dll",
     EntryPoint = "wg_detach_database",
     CallingConvention = CallingConvention.Cdecl)]
  public static extern int wg_detach_database(IntPtr dbase);
}
```

WhiteDB API sisaldab üle 300 funktsiooni ning käsitsi defineerimine võtaks liiga palju aega. Hilisemate muudatuste ja täienduste sisseviimine on samuti ajakulukas ning täpsust nõudev tegevus. Töö lihtsustamiseks sai kirjutatud programm, mis võtab sisendiks C .h faili ja genereerib sellest väljundina C# definitsioonid.

Antud API arenduse staadiumis sai programmeerida jõudlustestideks vajaliku funktsionaalsust, kuid ärirakendustes oleks selline kasutus kohmakas ning ei annaks olulisi eeliseid C programmeerimisekeele ees.

2.1 DataContext, DataRecord

Lihtsustamaks WhiteDB kasutuselevõttu rakendustes sai lisaks programmeeritud abstraktsioonikiht NativeAPI-le.

WhiteDB kasutab andmebaasile viitamise tavapärast mälule viitamist kasutades C andmeobjekti *void** ning C# vastab andmetüübile IntPtr. Klass DataContext peidab antud viite kasutamise klassisiseselt ära (viitele saab soovi korral ikkagi kätte) ning sisaldab endas funktsionaalsust andmebaasiga ühendamiseks, kirjete loomiseks ja kustutamiseks ning päringute teostamiseks.

```
public class DataContext : IDisposable
{
    // Kontruktor andmebaasi nime ja suurusega baitides
    public DataContext(string name, int size = 1000000000);

    // Tagastab uue loodud kirje
    public DataRecord CreateRecord(int length);

    // Kustutab kirje andmebaasist
    public void Delete(DataRecord record);

    // Loob andmebaasiga seotud uue päringuobjekti
    public QueryBuilder CreateQueryBuilder();

    // Tagastab andmebaasist esimese kirje
    public DataRecord GetFirstRecord();

    // Tagastab kirjele järgneva kirje
    public DataRecord GetNextRecord(DataRecord record);
}
```

Klass DataRecord on abstraktsioon WhiteDB kirjest ning sisaldab endal funktsionaalsust väljadele väärtuste kirjutamiseks (SetFieldValue meetodid) ning väljadelt lugemiseks (GetFieldValue* meetodid).

```
public class DataRecord
{
    public virtual char GetFieldValueChar(int index);
    public virtual DateTime GetFieldValueDate(int index);
    public virtual double GetFieldValueDouble(int index);
    public virtual int GetFieldValueInteger(int index);
    public virtual string GetFieldValueString(int index);
    public virtual DateTime GetFieldValueTime(int index);

    public void SetFieldValue(int index, char value);
    public void SetFieldValue(int index, double value);
    public void SetFieldValue(int index, int value);
    public void SetFieldValue(int index, string value);
    public void SetFieldValue(int index, DateTime value, DateSaveMode
mode);
}
```

2.2 Päringute teostamine

Päringute teostamiseks sai programmeeritud kaks eraldiseisvat klassi QueryBuilder ja Query. QueryBuilder on klass, mis tegeleb päringu koostamisega ja annab võimaluse anda päringule ette tingimusi, mille järgi otsida.

```
public class QueryBuilder
{
    public QueryBuilder AddCondition(uint column, ConditionOperator
condition, int value);
    public QueryBuilder AddCondition(uint column, ConditionOperator
condition, string value);
    public Query Execute();
}
```

Kasutusnäitena teostab järgnev programmikood otsingu teiselt väljalt ning tagastab otsingutulemuse, kus välja väärtus on „Tallinn“.

```
using (var database = new DataContext("1"))
{
    QueryBuilder queryBuilder = database.CreateQueryBuilder();
    queryBuilder.AddCondition(1, ConditionOperator.Equal, "Tallinn");
    Query query = queryBuilder.Execute();
}
```

Otsingutingimuste kuhjumisel muutub selline kasutusviis kohmakaks ning seetõttu eksponeerib QueryBuilder endas Fluent interface [6] põhimõttel funktsionaalsust. Põhimõtte seisneb selles, et iga meetod tagastab iseennast väljaarvatud resultaati tagastav meetud. Varasema näite saame niimoodi ümber kirjutada järgnevalt.

```
using (var database = new DataContext("1"))
{
    var query = database.CreateQueryBuilder()
        .AddCondition(1, ConditionOperator.Equal, "Tallinn").Execute();
}
```

Query klass sisaldab endas meetodit järgneva otsingutulemuse kirje lugemiseks ning tagastab tühiväärtuse kui kõik otsingutulemused on läbi käidud.

```
public class Query
{
    public DataRecord Fetch();
}
```

2.2.1 Mäluhaldus .NETis

C programmeerimiskeel kasutab tihtipeale andmete edastamiseks viitasid mälule, mis on pärast kasutamist vaja vabastada. Seetõttu realiseerivad DataContext ja Query klassid C# IDisposable interfacet, mis nõuab Dispose meetodit klassis, kus tegeletakse .NETi raamistikust väljaspool olevate ressursside vabastamisega. Lisaks sisaldab C# keel endas võtmesõna *using* mille abil on võimalik ressursside vabastamist lihtsustada. Olukorda on lihtsam selgitada näite varal.

```
StreamReader reader = new StreamReader("filename.txt");
// Failist lugemine vms
reader.Dispose();
// Fail on suletud ning
// objekt on vabastanud enda käsutuses olevad ressurssid
```

Kuigi siin toimub korrektne objekti väljakutsumine ja töö lõpetamine on sellisel lähenemisel puudused. Nimelt on reader muutuja kasutamiskõlbmatu, kuna fail millega ta töötas on juba suletud ning sisemised parameetrid võivad olla vales olekus. Parem on kasutada järgnevat näidet.

```
using (StreamReader reader = new StreamReader("filename.txt"))
{
    // Failist lugemine vms
}
```

Koodiplokkist *using* väljudes kutsutakse automaatselt välja meetod Dispose ning kõik ressursside vabastamiseks vajalikud tegevused tehakse ära. Lisaks ei tea enam plokile järgnevad read midagi muutujast *reader*, kui mingil põhjusel peaks keegi üritama muutujat kasutada siis saab vea juba lähtekoodi kompileerimisel.

2.3 POCO objektid ja generic DataContext

DataContext klass võimaldab WhiteDB-d kasutusele võtta lihtsamalt kui läbi NativeAPI klassi siis järgmine samm edasi on mudelklasside salvestamine teadmata, kuidas DataContext või NativeAPI töötavad. Lähteülesanne on järgnev. Meil on olemas klass Person, mis hoiab endas isiku nime ja vanust.

```
public class Person
{
    public int Age { get; set; }
    public string Name { get; set; }
}
```

Me soovime salvestada Person klassi instantsi WhiteDB andmebaasi võimalikult lihtsalt. Tavalise DataContext klassiga käiks see järgnevalt.

```
Person person = new Person { Name = "Kati", Age = 35 };
using (var database = new DataContext("1"))
{
    DataRecord record = database.CreateRecord(2);
    record.SetFieldValue(0, person.Age);
    record.SetFieldValue(1, person.Name);
}
```

Lihtsa näite puhul pole see kood väga keeruline, kuid suuremate andmeklasside puhul oleks selline lähenemine kohmakas. Seetõttu sai arendatud WhiteDB C# APIsse *generic* DataContext, mille peamiseks ülesandeks on andmete lihtsustatud salvestamine ja pärimine. Päringute teostamisest on seletatud järgmises osas ning siin keskendume salvestamisele.

```
public class DataContext<T>
{
    public IQueryable<T> Query();
    public T Create(T entity);
}
```

.NETi raamistik võimaldab andmetüüpe programmi töötamise ajal lähemalt uurida. Näiteks on võimalik iga klassi käest küsida, mis väljad tal küljes on või mis klassist ta pärineb. Samas saab ka uusi klasse juurde teha ning saab lahendada mõningaid probleeme, mis muidu oleks keeruline või isegi võimatu.
Kui rakendus on võtnud andmebaasist välja kirje, mida hilisemalt soovib uuendada, siis API-l on väga keeruline selgeks teha, mis kirjega oli tegemist. Kui vaadelda varasemalt kirjeldatud klassi Person siis seal pole kuskil kirjas viidet mäluaadressile WhiteDB andmebaasis. Üks lahendus oleks nõuda, et kõik WhiteDB-sse salvestatavad objektid pärineksid samast klassist või realiseeriksid kindlat liidest, kuid seda ma ei pea heaks praktikaks, sest see seob rakenduse mudeli kokku konkreetse andmebaasiga.
Parim lahendus probleemile on programmi töö käigus luua uus klass, mis pärineks Person klassist ja realiseeriks kindlat liidest. Vajadus iga kirje juures meeles pidada viidet andmebaasile ning viidet kirjele endale. Selleks defineerime liidese IRecord.

```
public interface IRecord
{
    IntPtr Database { get; set; }
    IntPtr Record { get; set; }
}
```

Kombineerides Person klassi IRecord liidesega saab järgneva klassi, kus on kõik Person klassi omadused ning realiseeritud IRecord liidesega sätestatud leping.

```
public class GeneratedPerson : Person, IRecord
{
    IntPtr Database { get; set; }
    IntPtr Record { get; set; }
}
```

Realiseeritud klass oleks kasutatav edasi Person klassi asemel, kuid kui arenduse käigus otsustatakse WhiteDB välja vahetada näiteks testimise ajaks siis pole vaja lähtekoodi muuta.

Genereeritud klassi loomiseks on vaja läbi teha järgnevad sammud:

1. Genereerida dünaamiline moodul
2. Defineerida uus klass (GeneratedPerson), mis pärineks Person klassist

3. Lisada GeneratedPerson klassile IRecord liides.
4. Realiseerida IRecord liides GeneratedPerson klassi sees.
5. Võtta kasutusele GeneratedPerson klass

Antud funktsionaalsus on realiseeritud ModelBuilder klassis ning saab edasi minna järgmise sammu juurde, kus GeneratedPerson klassi objektist on vaja andmed salvestada DataRecord klassi abil WhiteDB andmebaasi ning vastupidises suunas ka.

ModelBinder klass realiseerib meetodit, mille abil andmeid salvestatakse või loetakse.

```
public class ModelBinder<T>
{
    public T FromRecord(DataRecord record);
    public DataRecord ToRecord(T entity);
}
```

API laiendatavuse ja taaskasutatavuse huvides sai lisatud juurde abiklassid ja interface-d.

```
public interface IValueBinder
{
    object GetValue(DataRecord record, int index);
    void SetValue(DataRecord record, int index, object value);
}

public interface IValueBinder<T> : IValueBinder
{
}
```

IValueBinder interfacet realiseerivad klassid oskavad WhiteDB andmebaasist lugeda või salvestada andmeid kindlalt väljalt.

```
public class ValueBinderFactory
{
    public IValueBinder Get(Type type);
}
```

ValueBinderFactory klass oskab ette antud välja tüübi järgi leida klassi implementatsiooni, mis seda tüüpi välja oskab salvestada või lugeda WhiteDB-st.
Tulemusena käib andmete salvestamine algoritm lihtsate sammudena.

1. Leia kõik Person klassi väljad
2. Loo uus kirje andmebaasi
3. Itereeri üle kõikide Person klassi väljade
    a. Anna igale väljale indeks, mis vastab välja indeksile andmebaasis
    b. Leia välja andmetüüp
    c. Leia andmetüübile vastav IValueBinder
    d. Loe väljalt väärtus
    e. Salvesta väärtus andmebaasi kasutades IValueBinder-it.

Andmete lugemine andmebaasist on samaväärne protseduur

1. Leia kõik Person klassi väljad
2. Loo uus Person klassi instants,. siit tuleb juba tagasi GeneratedPerson, kasutades ModelBuilderit
3. Itereeri üle kõikide Person klassi väljade
    a. Loe andmebaasist väärtus

   b. Salvesta väärtus instantsi väljale
4. Täida IRecord liidese väljad viidetega andmebaasile ning kirjele

Nüüd on varasemalt näitena kasutatud programmi asendada järgnevaga

```
Person person = new Person { Name = "Kati", Age = 35 };

using (var database = new DataContext<Person>("1"))
{
    person = database.Create(person);
}
```

2.4 LINQ Provider WhiteDB andmebaasile

2007 aasta lõpus tuli välja .NET Framework 3.5, mis sisaldas endas uut viisi andmehulkadest päringute tegemisel [7]. Tehnoloogia peamisteks mõjustusteks olid SQLi ja Haskelli programmeerimiskeel ning muutus paradigma, kuidas andmehulki töödeldi. LINQ-t kasutades muutusid programmid vastama küsimusele „mida?“ mitte „kuidas?“.
Kui on programmis täisarvude massiiv väärtustega 1 kuni 10 siis paarisarvude massiivi leidmiseks oli vaja kirjutada järgnev programm.

```
int[] numbers = new int[] {1,2,3,4,5,6,7,8,9,10};
List<int> result = new List<int>();
for (int i = 0; i < numbers.Length; i++)
{
    if (numbers[i] % 2 == 0)
    {
          result.Add(numbers[i]);
    }
}
return result.ToArray();
```

Sama programm kasutades LINQd näeb välja järgnevalt.

```
int[] numbers = new int[] {1,2,3,4,5,6,7,8,9,10};
return numbers.Where(i => i % 2 == 0).ToArray();
```

LINQl on ka teistsugune ilmutusviis, mis sarnaneb oluliselt SQLile ning antud programmi saab kirjutata ka nii.

```
int[] numbers = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
return (from i in numbers where i % 2 == 0 select i).ToArray();
```

LINQ aluseks on IQueryable<T> interface, mille implementatsioonid ei piirdu ainult C# hulgaklassidega vaid on olemas võimalus teostada päringuid XMList või SQLi andmebaasidest. Samuti on olemas spetsiifilisemaid realisatsioone nagu Linq to Wikipedia [8] või LINQ to Active Directory [9].
Ühtne liides päringute teostamisel annab võimaluse vahetada aluseks olevaid päringu teostajaid. Programm võib arenduse algusfaasis hoida andmeid ainult enda mälus ning realiseerida äriloogilisi päringuid sealt. Hilisemas faasis tekib vajadus säilitada andmeid XML failidena kõvakettal ning siis pole vaja muuta kogu programmi vaid asendatakse allikas. Kui programmeerimisel kasutatakse ühiktestimist siis on tõenäoline vajadus säilitada mõlemad võimalused ning siis on oht keerukuse kasvul või koodi dubleerimisel.

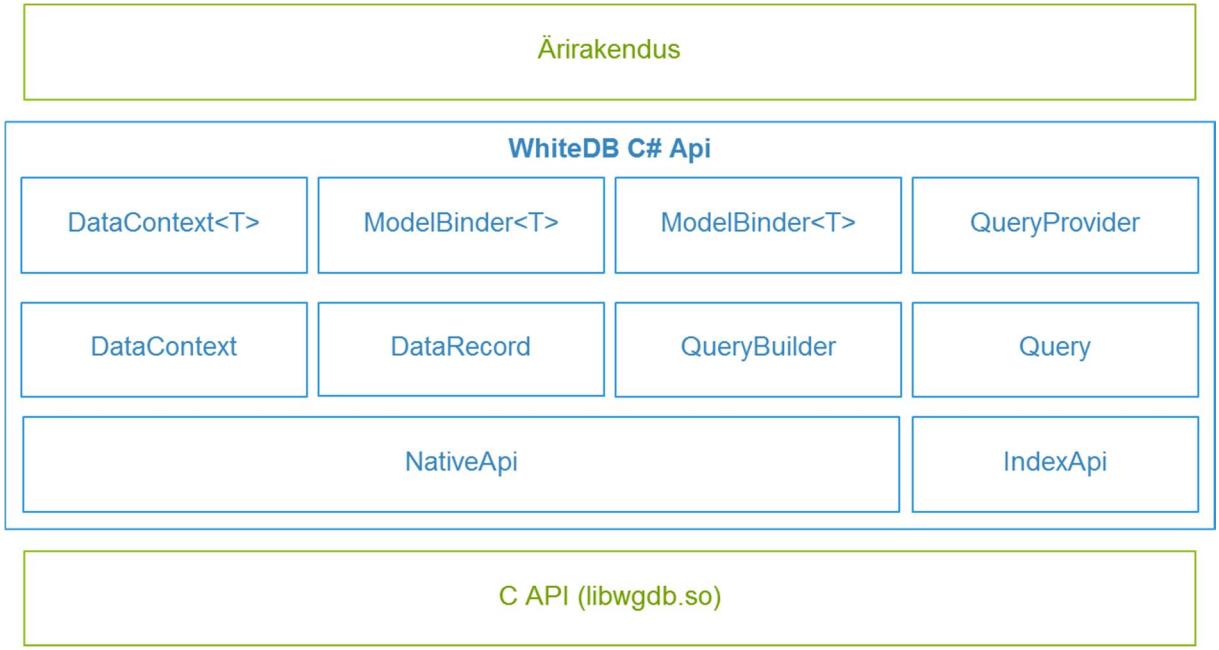
Iga andmehulgast päringu teostamise aluseks on, et sellele andmehulgale on olemas LINQ Provider, mis realiseerib ette antud päringuid.
IQueryProvider realiseerimiseks loodi QueryProvider klass, mis saab parameetriks päringu parameetri(d) ning ehitab üles QueryBuilderi abil üles päringu, mida saata WhiteDB-le täitmiseks. Päringu vastused konverteeritakse ümber DataContexti parameetri klassi objektideks kasutades ModelBinderi funktsionaalsust.
Tulemuseks saame kirjutada programmi, mis lisab andmebaasi 3 isikut ja teostab päringu kasutades LINQt.

```
using (var data = new DataContext<Person>("1"))
{
    data.Create(new Person { Age = 11, FirstName = "Kati", LastName =
"Karu" });
    data.Create(new Person { Age = 12, FirstName = "Mati", LastName =
"Mesi" });
    data.Create(new Person { Age = 14, FirstName = "Mati", LastName =
"Mustikas" });
    IQueryable<Person> query = data.Query().Where(a => a.FirstName ==
"Mati" && a.Age > 12);
    foreach (Person person in query)
    {
          Console.WriteLine("{0} {1}, {2}", person.FirstName,
person.LastName, person.Age);
    }
}
```

Oluline on märkida, et antud LINQ provideri implementatsioon on üsnagi piiratud võrreldes teistega ja seda järgnevatel põhjusel.

1. WhiteDBs puudub ühishulga mõiste, otse C API-t kasutades väljendada päringust, kus tingimustes esineb „või“ tingimus. Ideeliselt saaks selle probleemi lahendada kahe eraldiseisva päringu tegemisega ning nende tulemuste liitmisel, kuid antud töö skoobist jäi see välja
2. Pole võimalik otsida osaliselt teksti ja ei saa teha päringut, kus tingimus oleks näiteks kõik isikud, kelle eesnimi algab M tähega.
3. Samuti ei ole võimalik teha tehteid kirje väljade väärtustega ning ei saa esitada päringut, kus tulemuseks on kõik isikud, kelle vanus on paarisarv.
4. WhiteDB ei võimalda otsingutulemusi sorteerida.

2.5 Arhitektuuri joonis



Joonis 1 C# API

3. WhiteDB erinevate klient-APIde jõudluse võrdlus

WhiteDB on kirjutatud programmeerimiskeeles C, mis annab maksimaalselt ära kasutada arvuti võimalusi, kuid tarkvara loomine võib olla keerulisem, mistõttu eelistatakse ärirakenduste kirjutamisel kasutada kõrgema taseme keeli.

3.1 Jõudlustestide metoodika

Jõudlustestides võrreldakse 6 erinevat kasutusjuhtu. Iga kasutusjuht realiseeriti kolmes erinevas programmeerimiskeeles. Programmeerimiskeeles C loodud lahendus on aluseks teiste tulemuste võrdlemisel. Testimiseks kasutati Linuxi operatsioonisüsteemiga (Ubuntu 14.10) virtuaalarvutit, kuhu oli riistvaraliselt eraldatud 1 protsessori tuum ning 4500 megabaiti muutmälu. Tarkvarana on paigaldatud Mono (versioon: 3.2.8) .NET/C# rakenduse töötamiseks ning Python (versioon 2.7.8).

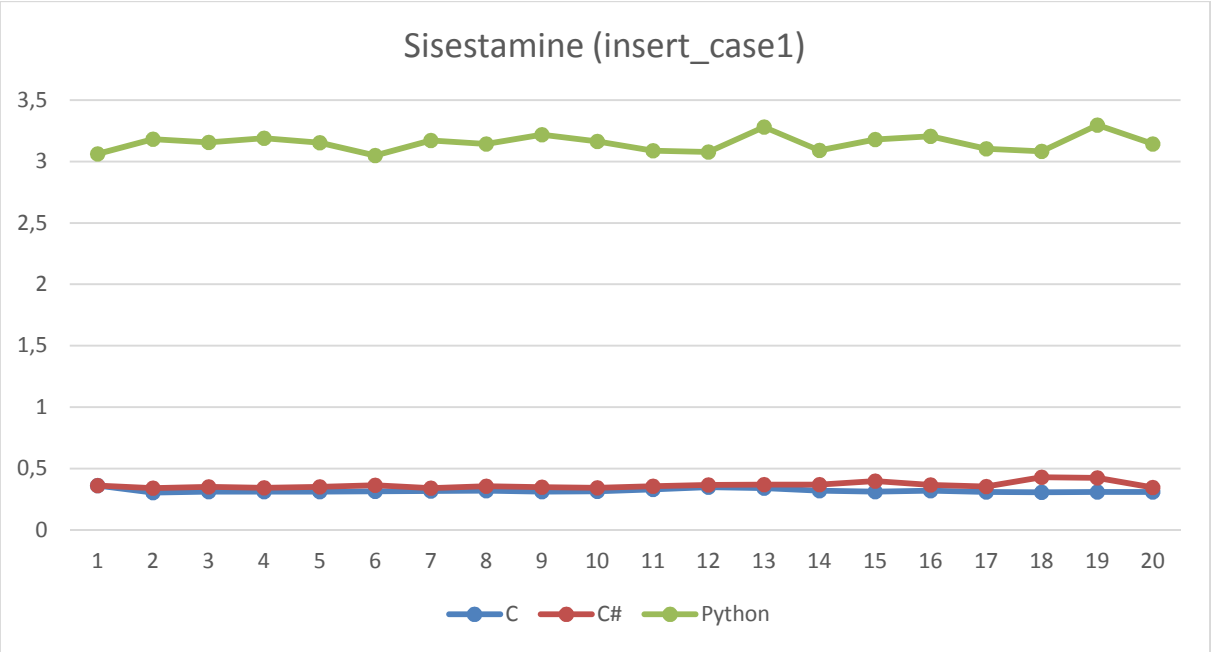
Jõudlustestideks kasutusel olev programmikood on internetis avalikult väljas, aadressil: https://github.com/andreireinus/whitedb-bench

3.2 Andmete sisestamine väikeses mahus

Jõudlustestis loodi tühi andmebaas suurusega 55 megabaiti, kuhu lisati 68000 kirjet, kus iga kirje sisaldas 100 numbrilist välja ning iga väli sai väärtuseks rea numbri.

Kui C ja C# erinevus kiiruses oli keskmiselt 13,96%, mis on tõenäoline kiirusekadu C#st C funktsioonide väljakutsumisel. MSDNi artikkel väidab, et keskmiselt on 10 kuni 30 x86 instruktsiooni iga väljakutse kohta. [10]

Python oli keskmiselt 9,8 korda aeglasem võrreldes C-ga ja näitab, kui ebaefektiivne on Pythoni enda andmestruktuuride muutmisel C andmestruktuurideks.



Joonis 2 API võrdlus - andmete sisestamine väikeses mahus

Tabel 1 API võrdlus – andmete sisestamine väikeses mahus

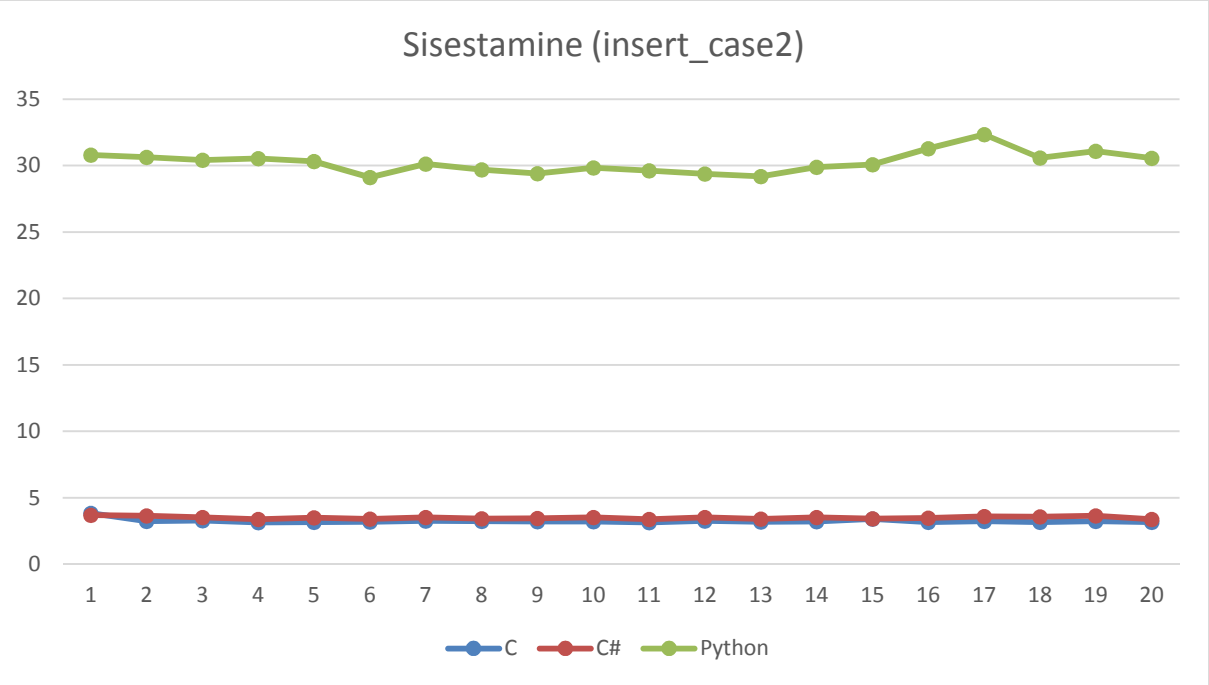
	C	**C#**	**Python**
Minimaalne	0,306s	0,340s	3,151s
Maksimaalne	0,362s	0,430s	3,048s
Keskmine	0,319s	0,364s	3,298s

3.3 Andmete sisestamine suures mahus

Jõudlustestis loodi tühi andmebaas suurusega 1 gigabait, kuhu lisati 680000 kirjet, kus iga kirje sisaldas 100 numbrilist välja ning iga väli sai väärtuseks rea numbri.

C# oli keskmiselt 7,6% aeglasem C-st, mis annab lootust pikalt kestva protsessi puhul on C#-i *overhead* väiksema osakaaluga.

Kuigi C# ja Pythoni vahe langes eelmise testiga, siis erinevus oli ikkagi keskmiselt 8,6 korda. Olukorras, kus WhiteDB-d kasutav rakendus vajab pidevalt sisestada suuremas koguses andmeid, siis peab arvestama Pythoni kasutamisest tingitud jõudluskaoga.



Joonis 3 API võrdlus - andmete sisestamine suures mahus

Tabel 2 API võrdlus – andmete sisestamine suures mahus

	C	**C#**	**Python**
Minimaalne	3,139	3,379	29,106
Maksimaalne	3,831	3,689	30,241
Keskmine	3,249	3,499	30,241

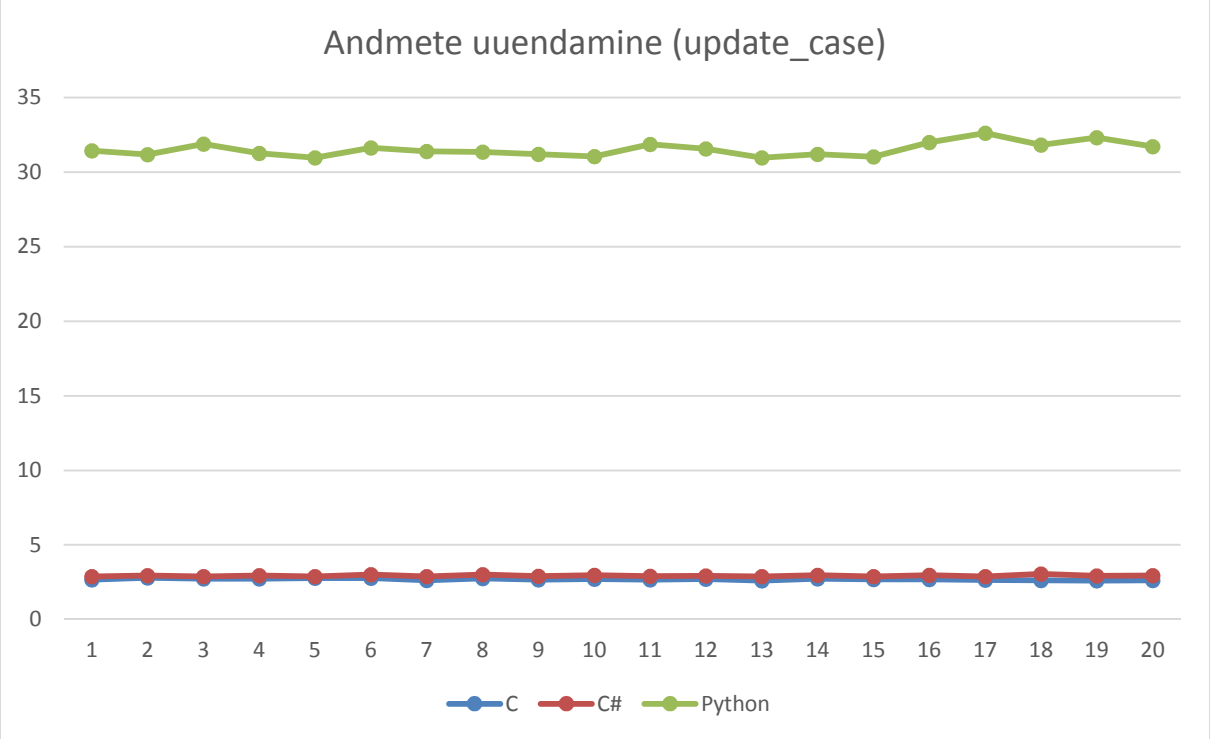
3.4 Andmete uuendamine

Enne testimist loodi tühi andmebaas ning lisati 680000 kirjet, kus iga kirje sisaldas 100 numbrilist välja ning iga väli sai väärtuseks rea numbri. Jõudlustesti käigus käidi läbi kõik andmebaasis olnud kirjed ning muudeti iga välja väärtust ühevõrra suuremaks. Ajakuluna mõõdeti ainult andmete muutmisele kulunud aega.

C programmeerimiskeeles on realiseeritult näeb välja järgnevalt.

```
record = wg_get_first_record(db);
while (record != NULL) {
   for (j = 0; j < field_count; j++) {
      wg_set_int_field(db, record, j, j + 1);
   }
   record = wg_get_next_record(db, record);
}
```

Ka selles testülesandes on C ning C# erinevus marginaalne, kuid Python on üle 10 korra aeglasem.



Joonis 4 API võrdlus - andmete uuendamine

Tabel 3 API võrdlus – andmete uuendamine

	C	C#	Python

Minimaalne	2,604	2,875	30,963
Maksimaalne	2,791	3,058	32,630
Keskmine	2,690	2,932	31,528

3.5 Andmete pärimine

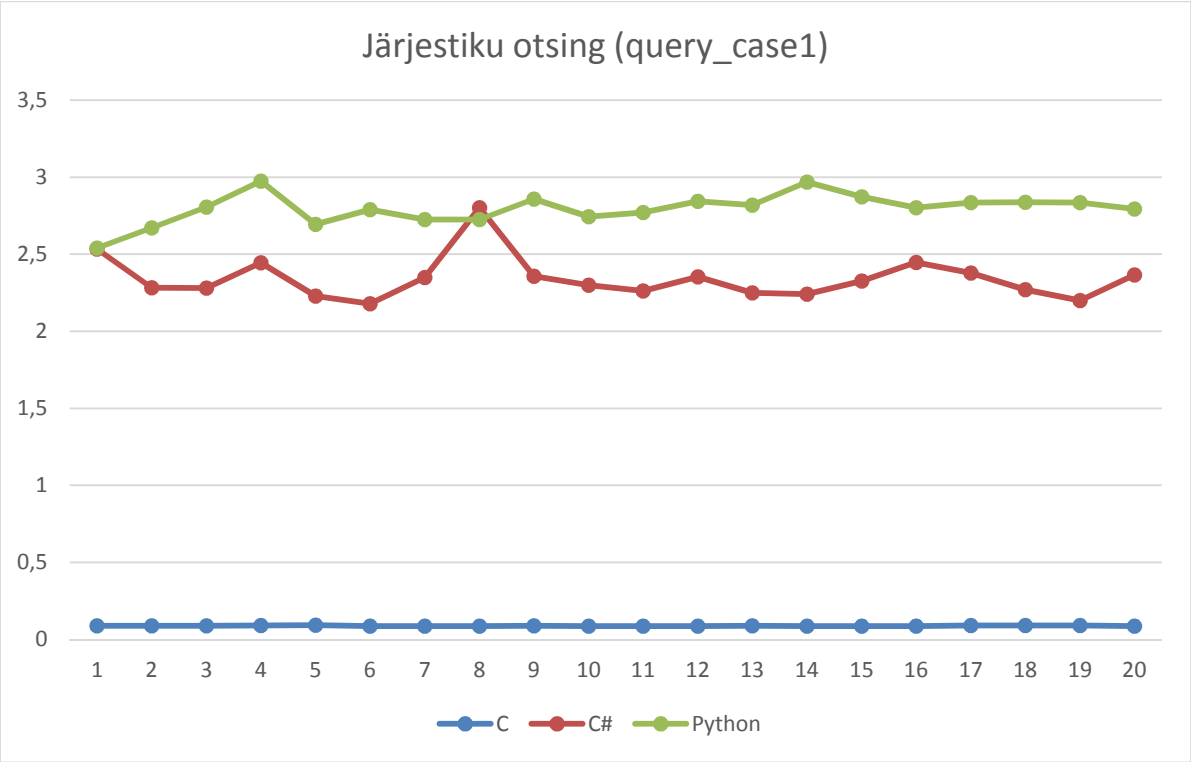
Andmete pärimisel kasutati maailma asulate andmebaasi [11], mis sisaldab endas 3.1 miljonit kirjet. Iga rida sisaldab andmeid asukohariigi, asula nime, elanikkonna kohta. Testi ülesanne on leida asulate seast kõik Eesti asulad.

3.5.1 Järjestiku otsing

Andmetest otsimine sooritatakse kasutamata WhiteDB sisemist otsingu funktsionaalsust vaid käiakse kõik kirjed ükshaaval läbi. C programmeerimiskeeles on testülesanne realiseeritud nii:

```
record = wg_get_first_record(db);
while (record != NULL) {
  str = wg_decode_str(db, wg_get_field(db, record, 0));
  if (strcmp("ee", str) == 0)
    count++;
  record = wg_get_next_record(db, record);
}
```

Tulemustena oli üllatav, et antud olukorras on C# ja Pythoni kiiruseerinevus üsna lähedane üksteisele.



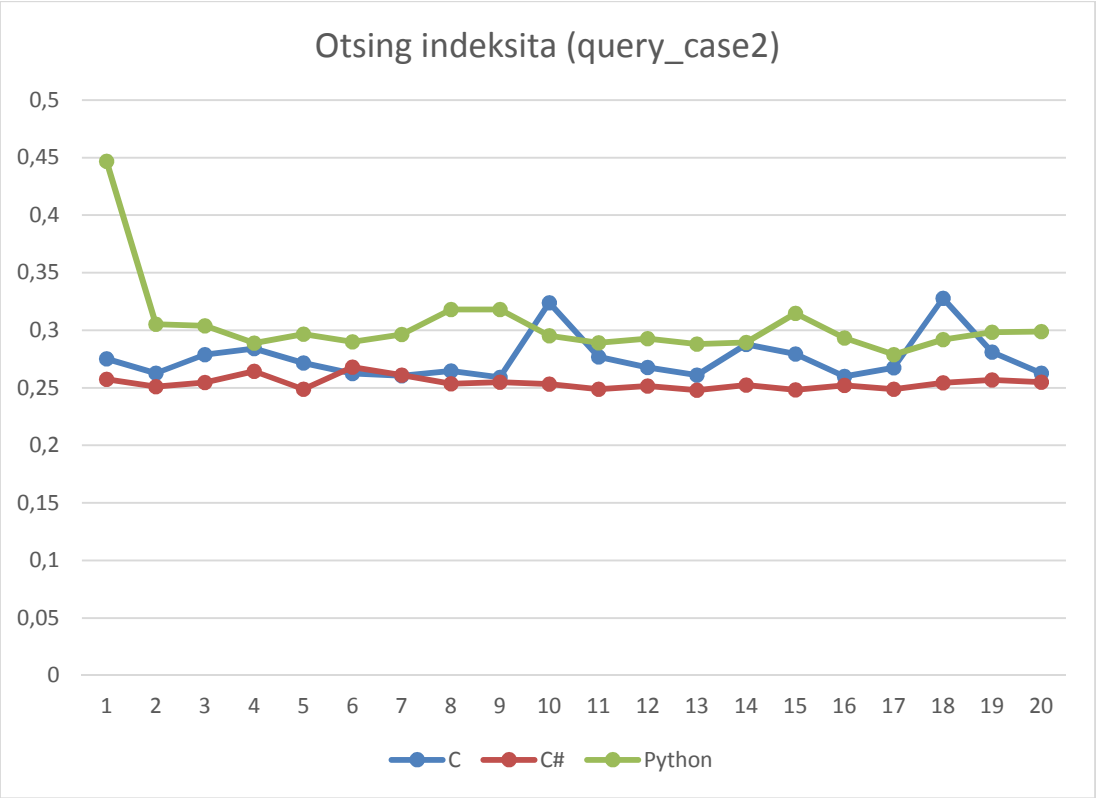
Joonis 5 API võrdlus - kirjete otsing järjestiku

Tabel 4 API võrdlus – kirjete otsing järjestiku

	C	**C#**	**Python**
Minimaalne	0,086	2,178	2,541
Maksimaalne	0,092	2,801	2,975
Keskmine	0,088	2,342	2,795

3.5.2 Indekseerimata otsing

WhiteDB C API kood uurides leidsin, et indeksita otsingu puhul käiakse kõik kirjed ükshaaval läbi nagu eelmises jõudlustestis aga nüüd teeb põhitöö ära C API sisemiselt ning andmete liikumist väljapoole C teeki on oluliselt vähem. Seda oodatud tulemust näitasid ka jõudlustestid, kuid ainukeseks üllatavaks osaks oli tulemus, kus C# oli keskmiselt kiirem kui C. Seda tulemust ma ei oska seletada, ning eeldan mingit programmeerimise viga jõudlustesti kirjutamisel C programmeerimiskeeles, mida ma ei suutnud tuvastada.



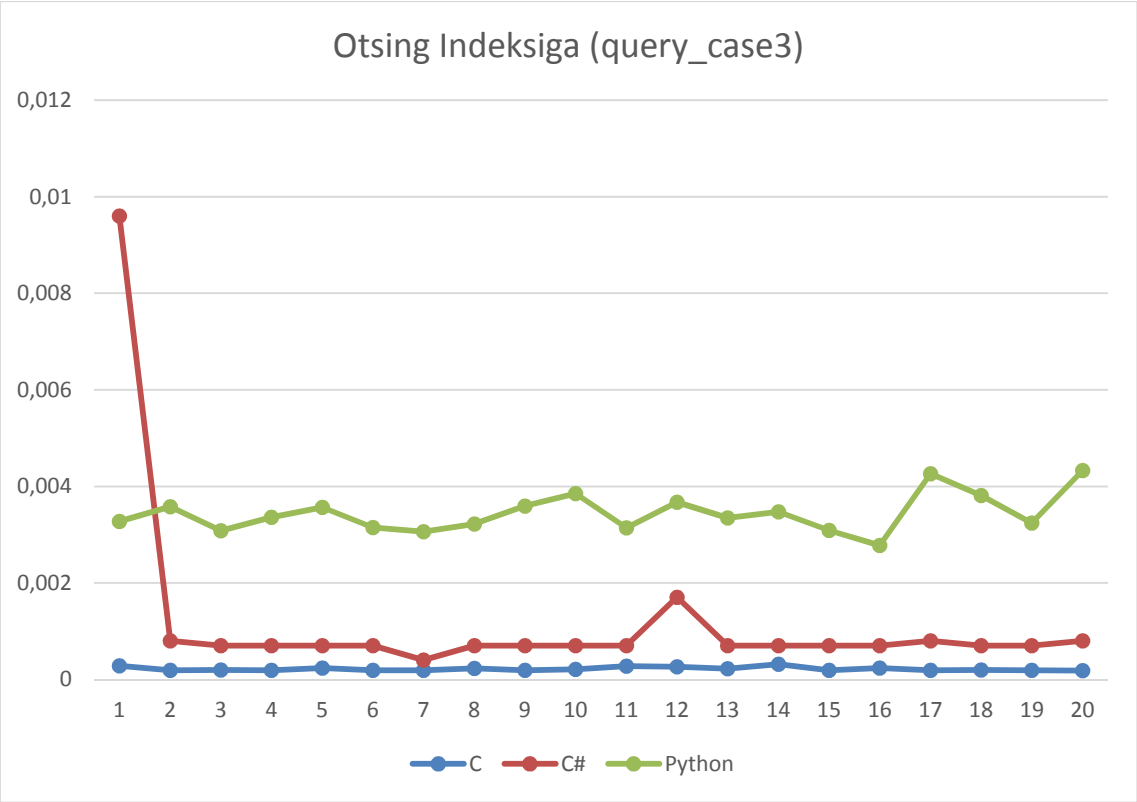
Joonis 6 API võrdlus - kirjete otsing indeksita

Tabel 5 API võrdlus – kirjete otsing indeksita

	C	**C#**	**Python**
Minimaalne	0,259	0,248	0,279
Maksimaalne	0,328	0,268	0,447
Keskmine	0,276	0,254	0,305

3.5.3 Indekseeritud välja järgi otsing

Indekseeritud väljalt otsing on üldiselt väga kiire kõikide API-dega ning kuna sisemiselt teeb põhitöö ära C API siis oli ka tulemus oodatult ühtlane.



Joonis 7 API võrdlus - kirjete otsing indeksiga

Tabel 6 API võrdlus – kirjete otsing indeksiga

	C	**C#**	**Python**
Minimaalne	0,00019s	0,00040s	0,00278s
Maksimaalne	0,00031s	0,00960s	0,00433s
Keskmine	0,00022s	0,00120s	0,00344s

3.6 Järeldused

Jõudlustestide tulemusena on näha, et kõik WhiteDB API-d sobivad kasutuseks reaalsete rakenduste loomisel. Ainukene nõrk koht oli Pythoni andmete sisestamise ja muutmine, mis viitab dünaamilise programmeerimiskeelte nõrkusele sisemiste andmetüüpide muutmisel C programmeerimiskeele andmetüüpideks. Loodud C# API on piisavalt kiire ja pakub konkurentsi ka C API-le ning võimaldab kirjutada rakendusi kõrgema taseme programmeerimiskeeles.

4. WhiteDB ja MongoDB jõudlustest

Uurimise all on WhiteDB kiirus võrreldes enimlevinud NoSQL andmebaasiga. DB-Engines Ranking veebilehe andmetel [12] on MongoDB Mai 2015 seisuga kõige populaarsem NoSQLi andmebaas ja üldises tabelis tõusnud 4ndale kohale. Eespool on relatsioonilased andmebaasid: Oracle, MySQL ning Microsoft SQL Server. Antud pingereas on WhiteDB auväärsel viimasel kohal. Soov on teada saada kuidas WhiteDB peab vastu võrdluses populaarseima andmebaasiga.

WhiteDB-l on võimalus salvestada andmeid kõvakettale, kuid antud uurimus käsitleb mälupõhist opereerimist ning selleks, et MongoDB ei salvestaks andmeid maha kõvakettale muudeti andmebaasi konfiguratsiooni. Linuxi keskkonnas loodi 2 gigabaidine mälupõhine kõvaketas kasutades järgnevaid käske.

```
# mkdir /ramdisk
# mount -t tmpfs -o size=2000M tmpfs /ramdisk/
```

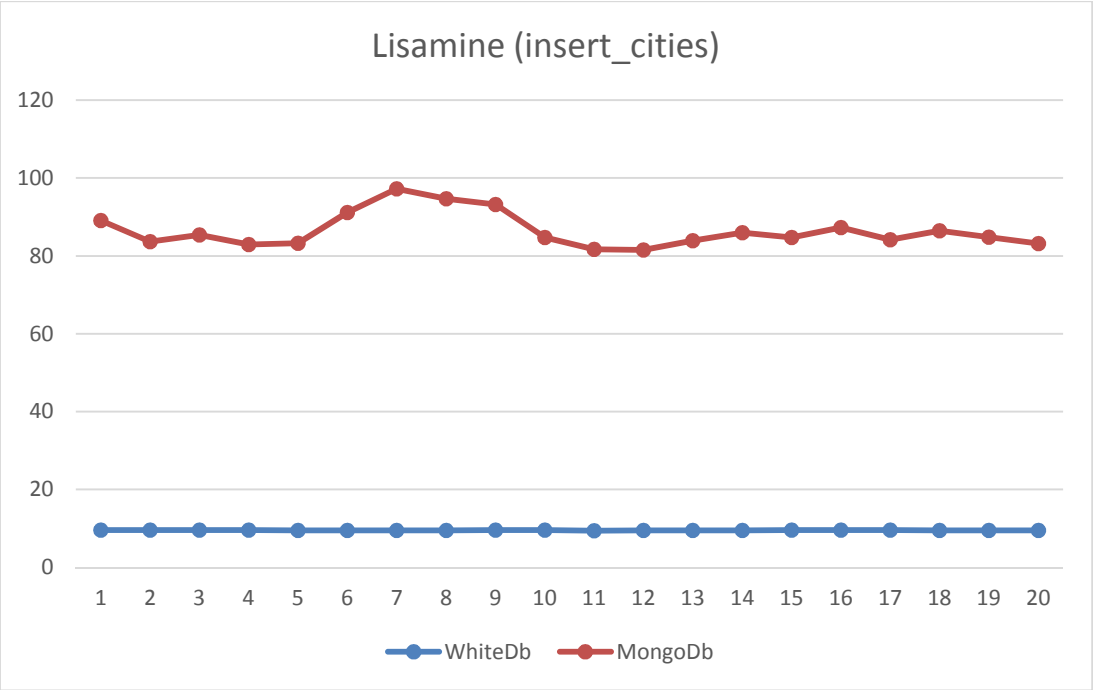
MongoDB käivitamisel kasutati loodud kõvaketast ja muudeti konfiguratsiooni seadeid, et vältida üleliigset tööd kõvakettal.

```
mongod --dbpath /ramdisk --smallfiles --noprealloc --nojournal
```

4.1 Lisamine

Uuringu aluseks võeti maailma asukohtade andmebaas, kus on 3173958 asulat [11]. Iga asula kohta on teada riigi kood, nimi, administratiivne üksus, elanike arv ja paiknemine kaardil. Andmed loeti kõvakettal asuvast failist enne sisestamist mällu, et vältida kõvaketta aeglusest tulevaid mõjutusi mõõtmisel.

Andmete importimisel oli MongoDB keskmisel 8 korda aeglasem, kuid palju murettekitavam on tööks kulunud aja suur kõikumine. Minimaalse ja maksimaalse vahe moodustas 20% keskmisest kulunud ajast. Samal ajal oli WhiteDB puhul kõikumine 1.5%.



Joonis 8 Andmebaaside võrdlus – lisamine

Tabel 7 Andmebaaside võrdlus - lisamine

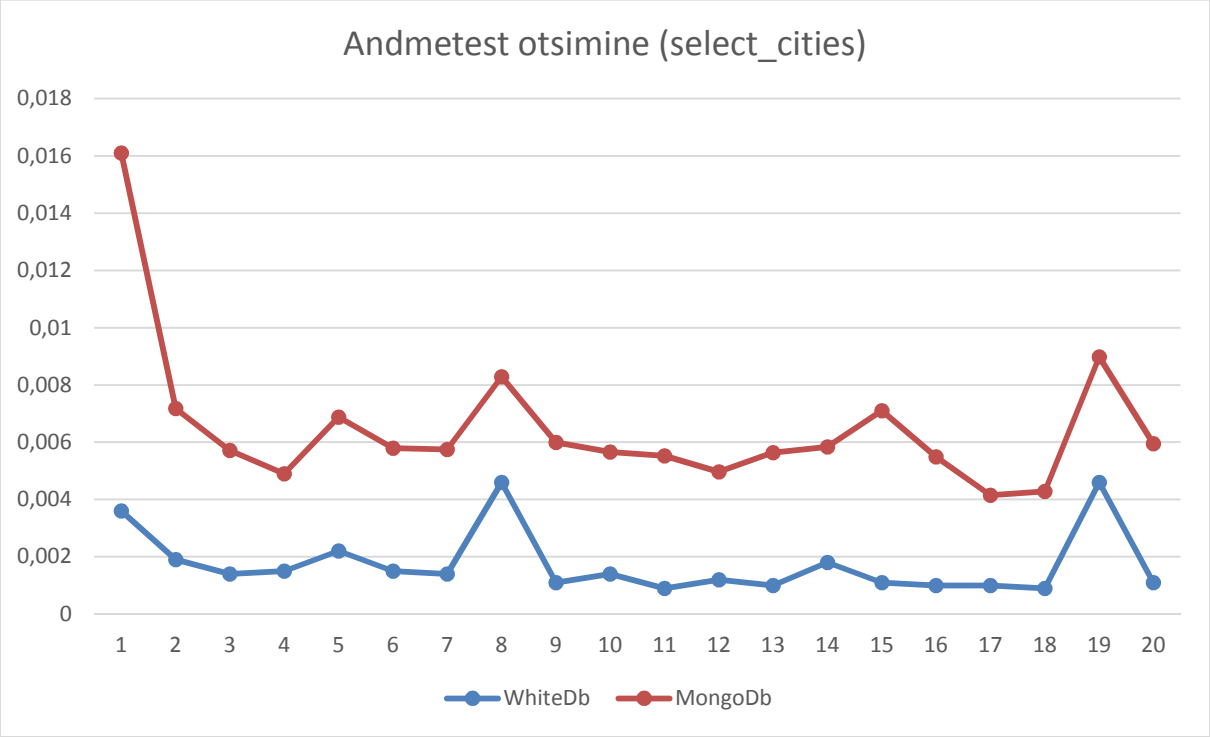
	WhiteDB	**MongoDB**
Minimaalne ajakulu	9,477s	72,058s
Maksimaalne ajakulu	9,624s	87,678s
Keskmine ajakulu	9,546s	76,914s

4.2 Otsing indeksi järgi

Antud ülesanne näitab, kui edukalt suudab andmebaas leida indekseeritud andmete seast ning alusandmestikuks võeti asulate andmebaas, kuhu loodi indeksid asukohariigi, nime ja rahvastiku arvule.

Asulate andmebaasist on vaja lugeda mitmele asulale on asukohariigiks märgitud Eesti. Selliseid asulaid on 9377, mis moodustab koguandmestikust 0,00295%. Järgnevalt on ülesanne leida üles Tallinna linn ning leida, mitu asulat on sarnase rahvastiku arvuga. Erinevus võis olla 1% mõlemale poole. Selliseid asulaid on andmebaasis 7.

Kuigi WhiteDB osutus testis kiiremaks siis erinevus on vähenenud, mis annab alust arvata MongoDB paremast päringute sisemisest optimeerimisest.



Joonis 9 Andmebaaside võrdlus – otsimine

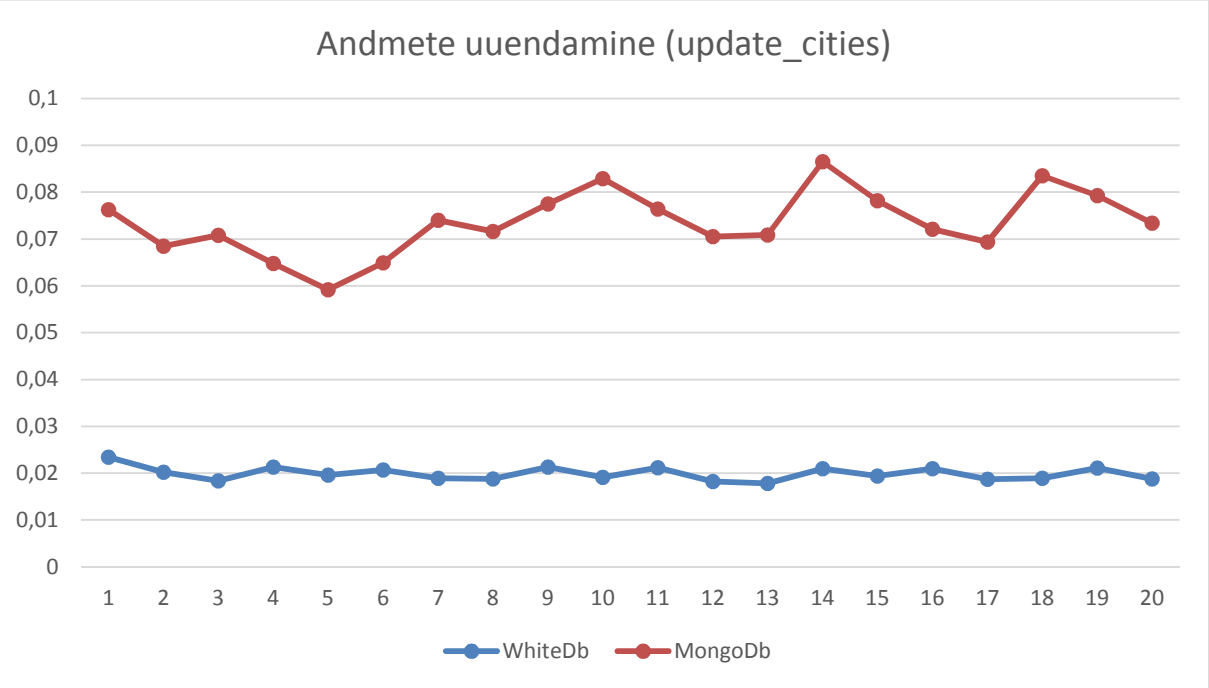
Tabel 8 Andmebaaside võrdlus - otsimine

	WhiteDB	**MongoDB**
Minimaalne ajakulu	0,0009s	0,0032s
Maksimaalne ajakulu	0,0046s	0,0125s
Keskmine ajakulu	0,0018s	0,0047s

4.3 Andmete uuendamine

Andmete uuendamise ülesandeks oli leida asulate andmebaasist kõik Eesti asulad ning muuta nende elanike arv. Oluline on märkida, et kuna elanike arvu väli on indekseeritud, siis andmete muutmine sisaldab ka indeksi uuendamist.

Mõlemad andmebaasid on väga kiired ning kiiruseerinevus oli võrreldes eelmise ülesandega samaväärne. Otsingu puhul oli erinevus 2,6 korda ning uuendamisel 2,7 korda.



Joonis 10 Andmebaaside võrdlus – uuendamine

Tabel 9 Andmebaaside võrdlus - uuendamine

	WhiteDB	**MongoDB**
Minimaalne ajakulu	0,018s	0,040s
Maksimaalne ajakulu	0,023s	0,066s
Keskmine ajakulu	0,020s	0,054s

5. Kokkuvõte

WhiteDB-l puudus avalik ja vabalt kasutatav liides .NETi arendajatele, mis piiras WhiteDB levikut ja populaarsuse kasvu. Antud töö eesmärk oli võimaldata .NETi platvormil kasutada WhiteDB funktsionaalsust ning pakkuda lisaks lihtsustatud andmete salvestamist ja pärimist.

Töös oli planeeritud programmeerida ja avalikustada C# API ja jõudlustestid erinevate olukordadeks. Jõudlusttestid, mis võrdlesid WhiteDB erinevaid APIsid, aitasid uurida ja välja selgitada, kuidas C# API kiirus on võrreldes teiste avalike WhiteDB API-dega. Võrdluses MongoDB-ga uuriti, kui kiire on WhiteDB võrreldes MongoDB-ga ning missugust erinevust võib jõudluses oodata ka päriselulistel rakendustel.

Publitseeritud C# API sisaldab lisaks funktsionaalsust, mis võimaldab .NETi arendajatel kirjutada rakendusi kasutades WhiteDB-d ja teadmata tema täpsemaid funktsioneerimise detaile. Päringute teostamise lihtsustamiseks on programmeeritud LINQ provider, mis võimaldab ühtsel viisil andmeid pärida WhiteDB andmebaasist võrreldes teiste andmekogudega. API jõudlustestid näitasid, et ühendus C ja C# vahel on piisavalt õhukene ning kiirusekadu ei mängi rolli.

MongoDB on aeglasem võrreldes WhiteDB-ga, kuid peab arvestama, et MongoDB pakub rohkem võimalusi võrreldes WhiteDB-ga.

Summary

WhiteDB does not have a public and open-source API for .NET developers and this missing feature does not help to rise WhiteDB popularity. Project goal is to build C# API for .NET and add additional features to simplify querying and creating records.

Project planned to develop and publish C# API and performance test for several use-cases. Performance test compared WhiteDB different programming language APIs and helped to research and determine how much overhead is in C# API compared to other WhiteDB APIs. Comparison to MongoDB tries to find out how much difference is between two of them and what is the performance difference expected in real life applications.

Published C# API includes extra functionality and enables .NET developers to write applications using WhiteDB without knowing internal workings of WhiteDB. LINQ provider was implemented to simplify query operations and can be used in some cases as drop-in replacement for other data storages. API performance test results show small overhead in calling C code from C#, but overall performance of C# API is good.

WhiteDB is faster than MongoDB. When considering WhiteDB in order to replace MongoDB then MongoDB's larger feature set should be compared to application requirements and make decision based to findings.

Kasutatud kirjandus

[1] H. Vallaste, „e-Teatmik: IT ja sidetehnika seletav sõnaraamat,“ [Võrgumaterjal]. Available: http://www.vallaste.ee/.

[2] Oracle Corporation, „Oracle Technology Global Price List,“ [Võrgumaterjal]. Available: http://www.oracle.com/us/corporate/pricing/technology-price-list-070617.pdf. [Kasutatud 16 Mai 2015].

[3] Microsoft Corporation, „How to buy SQL Server,“ [Võrgumaterjal]. Available: http://www.microsoft.com/en-us/server-cloud/products/sql-server/purchasing.aspx. [Kasutatud 16 Mai 2015].

[4] Wikimedia Foundation, „NewSQL,“ [Võrgumaterjal]. Available: http://en.wikipedia.org/wiki/NewSQL. [Kasutatud 16 Mai 2015].

[5] P. J. Tanel Tammet, „WhiteDB,“ [Võrgumaterjal]. Available: http://whitedb.org. [Kasutatud 16 Mai 2015].

[6] .NET Foundation, 2014. [Võrgumaterjal]. Available: https://github.com/dotnet. [Kasutatud 16 Mai 2015].

[7] Wikimedia Foundation, „Fluent interface,“ [Võrgumaterjal]. Available: http://en.wikipedia.org/wiki/Fluent_interface. [Kasutatud 16 Mai 2015].

[8] Wikimedia Foundation, „Language Integrated Query,“ [Võrgumaterjal]. Available: http://en.wikipedia.org/wiki/Language_Integrated_Query. [Kasutatud 16 Mai 2015].

[9] „Linq to Wikipedia,“ [Võrgumaterjal]. Available: https://linqtowikipedia.codeplex.com/. [Kasutatud 16 Mai 2015].

[10] „Linq to Active Directory,“ [Võrgumaterjal]. Available: http://linqtoad.codeplex.com/. [Kasutatud 16 Mai 2015].

[11] Microsoft Corporation, „MSDN,“ 2015. [Võrgumaterjal]. Available: https://msdn.microsoft.com/en-us/library/ms235282.aspx. [Kasutatud 16 Mai 2015].

[12] MaxMind, „Free World Cities Database,“ [Võrgumaterjal]. Available: https://www.maxmind.com/en/free-world-cities-database. [Kasutatud 16 Mai 2015].

[13] DB-Engines, „DB-Engines,“ 2015. [Võrgumaterjal]. Available: http://db-engines.com/en/ranking. [Kasutatud 16 Mai 2015].