TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

TAPIO LINNIMÄKI

ONTOLOGIAN TIEDONHAKUPALVELU KÄYTTÄEN JDO:TA JA DOKUMENTTITIETOKANTAA

Diplomityö

Tarkastaja: Yliopistonlehtori Timo Aaltonen
Tarkastaja ja aihe hyväksytty
Tieto- ja sähkötekniikan tiedekunta-neuvoston kokouksessa 04.03.2015

TIIVISTELMÄ

Tapio Linnimäki: Ontologian tiedonhakupalvelu käyttäen JDO:ta ja dokumenttitietokantaa
Tampereen teknillinen yliopisto
Diplomityö, 51 sivua
Toukokuu 2015
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Ohjelmistotuotanto
Tarkastaja: yliopistonlehtori Timo Aaltonen

Avainsanat: ontologia, JDO, MongoDB, SPARQL, JDOQL

Ohjelmistotieteessä ontologialla tarkoitetaan formaalia kuvausta sovellusalueen käsitteistä ja käsitteiden välisistä suhteista. Ontologioiden käytön tarkoituksena on kuvata sovelluksen tietosisältö niin, että myös tiedon merkitys ja käyttötarkoitus (semantiikka) ovat koneellisesti tulkittavissa. Ontologioiden avulla aihealueen tietämystä pystytään käyttämään uudelleen ja jakamaan eri tietojärjestelmien välillä, mikä mahdollistaa mm. järjestelmien välisen yhteensopivuuden.

Tässä diplomityössä toteutettiin ontologian tiedonhakupalvelu käyttäen ontologiadatan tietovarastona MongoDB-dokumenttitietokantaa JDO-ohjelmointirajapintaa hyödyntäen. Tiedonhakupalvelu ottaa vastaan SPARQL-tyylisiä kyselyitä ja palauttaa kyselyyn vastaukseksi ontologian instansseja. Vaatimuksena oli myös ontologiaskeeman ajonaikaisen muokkaamisen mahdollistaminen. Ontologian tiedonhakupalvelusta oli olemassa relaatiotietokantaa käyttävä toteutus, josta voitiin uudelleenkäyttää komponentteja uudessa toteutuksessa. Tietokantaa käyttävä osuus ontologian tiedonhakupalvelusta toteutettiin uudestaan. Tähän kuului mm. tietokannan tietomallin suunnittelu ja SPARQL-tyylisten kyselyjen ohjelmallinen muuntaminen JDOQL-kyselykielelle.

Työn alkupuolella esitellään toteutuksessa käytetyt teknologiat ja ontologioita yleisesti. Tämän jälkeen kuvataan toteutuksessa käytetyt arkkitehtuuri-, tietomalli- ja toteutusratkaisut, sekä vastaan tulleita ongelmia. Erityisesti JDO-toteutuksen MongoDB-tietokannan tuen puutteet tuottivat ongelmia tiedonhakupalvelun suorituskyvyn suhteen. Lopuksi esitellään hakupalvelun testitulokset ja jatkokehitystarpeita.

ABSTRACT

Tapio Linnimäki: Ontology search service using JDO and a document-oriented database
Tampere University of Technology
Master of Science Thesis, 51 pages
May 2015
Master's Degree Programme in Information Technology
Major: Software Engineering
Examiner: University Lecturer Timo Aaltonen

Keywords: ontology, JDO, MongoDB, SPARQL, JDOQL

In computer science an ontology is a formal definition of the entities and their interrelationships for a certain application area. The purpose for ontologies is to define the data content of an application in a way that also the meaning (semantics) and purpose of the data can be programmatically interpreted. By utilizing ontologies the application area knowledge can be shared and reused between software systems which also enables interoperability between these systems.

This thesis describes the implementation of an ontology search service using JDO API and MongoDB as the underlying database system for ontology data. The implemented search service takes in SPARQL-like queries and returns ontology instances as query results. One requirement was also to enable runtime modification of the ontology schema. Some components from an existing relational database using implementation of the ontology search service could be reused. The database specific parts of the search service were reimplemented. This included among other things designing the data model of the database and programmatic conversion of SPARQL-like queries to JDOQL query language.

At the beginning of this thesis the technologies used in the ontology search service implementation are described. A generic introduction to ontologies is also given, including languages for describing and querying ontologies. After that, the data model, architecture and implementation decisions are presented along with problems encountered. Especially the JDO-implementation's incomplete support for MongoDB caused some performance issues. Finally, the search service test results are presented.

ALKUSANAT

Tämä diplomityö on tehty osana tietotekniikan koulutusohjelman diplomi-insinööritutkintoani Tampereen teknillisessä yliopistossa. Haluan kiittää työnantajaani Insta DefSec Oy:tä diplomityöntekomahdollisuudesta ja kaikkia työn tekemistä edesauttaneita henkilöitä. Erityiskiitokset työn ohjaajalle Said Benamarille ja työn tarkastajalle Timo Aaltoselle hyvästä ohjauksesta ja kommenteista. Lisäksi haluan kiittää vanhempiani tuesta opintojen aikana.

Tampereella, 29.4.2015

Tapio Linnimäki

SISÄLLYSLUETTELO

LYHENTEET JA MERKINNÄT

BSON	Binary JSON
CRUD	Create, Read, Update, Delete
IRI	Internationalized Resource Identifier
JDO	Java Data Objects
JDOQL	Java Data Objects Query Language
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
OWL	Web Ontology Language
POJO	Plain Old Java Objects
QBE	Query By Example
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
SPARQL	SPARQL Protocol and RDF Query Language
URI	Uniform Resource Identifier
W3C	The World Wide Web Consortium
XML	Extensible Markup Language

1. JOHDANTO

Ohjelmistotieteessä *ontologia* on formaali kuvaus sovellusalueen käsitteistä ja käsitteiden välisistä suhteista. Ontologioiden käytön tarkoituksena on kuvata sovelluksen tietosisältö niin, että myös tiedon merkitys ja käyttötarkoitus (semantiikka) ovat koneellisesti tulkittavissa. Ontologioiden avulla aihealueen tietämystä pystytään käyttämään uudelleen ja jakamaan eri tietojärjestelmien välillä, mikä mahdollistaa järjestelmien välisen yhteensopivuuden. Lisäksi järjestelmän on mahdollista päätellä ontologiaa hyödyntäen automaattisesti lisäinformaatiota mallinnettavasta sovellusalueesta, mikä mahdollistaa järjestelmän entistä älykkäämmän toiminnan.

Ontologiatiedon tallennukseen ja ontologiasta tiedon hakemiseen käytettävää ohjelmistokomponenttia kutsutaan *ontologiamoottoriksi*. Tämän työn tavoitteena on siirtää relaatiotietokannan päälle rakennetun olemassa olevan ontologiamoottorin tiedonhakupalvelu käyttämään ontologiadatan säilytykseen MongoDB-dokumenttitietokantaa [11] JDO-ohjelmointirajapintaa [1] hyödyntäen. Toteutettu tiedonhakupalvelu tarjoaa rajapinnan tiedon hakemiseen ontologiasta käyttäen SPARQL-tyylistä [23] kyselykieltä ja mahdollistaa ontologiaskeeman ajonaikaisen muokkaamisen. Motivaationa siirrolle on saada ontologiapalvelu käytettäväksi osana isompaa sovelluskokonaisuutta, jossa tiedon säilytykseen käytetään yllä mainittuja teknologioita.

Tässä diplomityössä kuvataan teknologiat, joita uudessa, JDO:ta ja MongoDB-dokumenttitietokantaa käyttävässä, ontologian tiedonhakupalvelun toteutuksessa hyödynnettiin. Palvelun toteutuksessa vastaan tulleet ongelmat ja niiden ratkaisussa käytetyt arkkitehtuuri-, tietomalli- ja toteutusratkaisut esitellään yksityiskohtaisesti ja kerrotaan kuinka niihin päädyttiin. Lisäksi esitellään kuinka toteutettu palvelu testattiin.

Luvussa 2 esitellään tässä työssä toteutetun ontologian tiedonhakupalvelun toteutuksen kannalta keskeiset teknologiat. Esiteltäviä teknologioita ovat toteutuksessa käytetty ohjelmointikieli, MongoDB-dokumenttitietokanta, JDO ja käytetty yksikkötestauskehys.

Luvussa 3 käydään läpi yleisesti ontologioita ja ontologioihin liittyviä teknologioita. Ensin esitellään yleisesti mitä ontologiat ohjelmistotieteen näkökulmasta ovat. Seuraavaksi perehdytään ontologioiden kuvauksessa, ontologioista tiedon hakemisessa ja ontologioiden tallennuksessa käytettäviin tekniikoihin.

Luvussa 4 kuvataan aluksi tarpeellisin osin ontologiamoottorin nykyistä relaatiotietokantatoteutusta, sekä mitä nykyisestä toteutuksesta muutetaan ja miksi. Sitten esitetään ontologian tiedonhakupalvelun uuden toteutuksen arkkitehtuuri. Ontologiapalvelun ra-

kenteesta ja sen liittymisestä osaksi muuta järjestelmää annetaan kokonaiskuva. Tämän jälkeen esitellään yksityiskohtaisemmin toteutetun tiedonhakupalvelun ohjelmistorakenne ja tietokannan tietomalli. Luvussa kuvataan myös palvelun suunnittelussa vastaan tulleita ongelmia ja kuinka ne on pyritty ratkaisemaan esitellyillä arkkitehtuuriratkaisuilla.

Luvussa 5 kuvataan toteutusyksityiskohdat, kuten kuinka ontologian tiedonhakupalvelu muodostaa ja suorittaa ontologiakyselyn. Luvussa kuvataan myös toteutuksessa vastaan tulleita ongelmia ja millaisilla toteutusratkaisuilla ne on pyritty ratkaisemaan. Myös toteutuksen testaus kuvataan ja testauksen tulokset esitetään.

Luvussa 6 esitellään työn tulokset. Luvussa kerrotaan mitä työssä saatiin aikaiseksi ja arvioidaan toteutuksen onnistumista. Lisäksi pohditaan mitä jatkokehitystarpeita toteutetussa komponentissa on.

Luvussa 7 esitetään yhteenveto koko diplomityöstä ja sen tuloksista. Lisäksi arvioidaan diplomityön onnistumista.

2. KÄYTETYT TEKNOLOGIAT

2.1 Java

Java on Sun Microsystemsin kehittämä oliopohjainen vahvasti tyypitetty ohjelmointikieli [8]. Se on ohjelmointikielen suosiota mittaavan langpop.com-sivuston [13] mukaan yksi suosituimpia ohjelmointikieliä. Javan mukana tulee erittäin kattava standardikirjasto. Standardikirjastoon kuuluu mm. graafisten käyttöliittymien luontiin tarkoitettu Swing-kirjasto ja rinnakkaisuuden hallinta [8]. Uusimmissa Java-versioissa on mukana myös modernimpi Javafx-käyttöliittymäkirjasto [9], jonka on tarkoitus olla Swing-kirjaston seuraaja ja korvata tämä Javan standardina käyttöliittymäkirjastona. Lisäksi tarjolla on runsaasti kolmansien osapuolien kehittämiä kirjastoja, kuten yksikkötestikehys JUnit [10]. Tässä työssä käytetty Java-versio on Java 7.

Eräs Javan eduista on Javalla kirjoitettujen ohjelmien käyttöjärjestelmäriippumattomuus [8]. Tämä johtuu siitä, että Java-koodia ei käännetä suoraan konekielelle, kuten esimerkiksi C++-kielessä. vaan lähdekoodi käännetään tavukoodiksi, jota ajetaan Javan virtuaalikoneessa (Java Virtual Machine, JVM). Javan virtuaalikone toimii myös ns. hiekkalaatikkona Java-ohjelmalla, joka voi rajoittaa Java-ohjelma pääsyä isäntäkoneen muihin prosesseihin ja resursseihin. Tämä parantaa Java-ohjelmien turvallisuutta. Haittapuolena tässä Java-ohjelmien ajotavassa kuitenkin on, että Java-ohjelmien suoritus on hiukan hitaampaa, kuin konekielisten ohjelmien.

Lisäksi Java tarjoaa ohjelman muistinhallintaa helpottavan roskienkerääjän, joka poistaa automaattisesti muistista tiedot, joihin ei enää viitata [8]. Esimerkiksi C++-kielessä dynaamisesti varatun muistin vapauttaminen täytyy tehdä manuaalisesti ohjelmassa, joka tosin tarjoaa ohjelmoijalle suuremman vapauden muistinkäytön optimointiin, mutta on myös virheherkempää.

Eräs Javan tässä työssä keskeisistä ominaisuuksista on annotaatiot (annotations) [8]. Mahdollisuus käyttää annotaatioita Java-koodissa tuli Javan versiossa 1.5. Se on syntaktinen tapa lisätä Java-koodiin metadataa, joka antaa lisätietoa siitä koodin kohdasta, johon annotaatio on lisätty ja tuo myös toiminnallisuutta. Annotaatioita voidaan lisätä kohtiin, joissa esitellään jokin ohjelman elementti. Tällaisia kohtia ovat esimerkiksi, kun esitellään luokka, luokan kenttä, luokan metodi, metodin parametri tai muuttuja.

Java-kielessä on sisäänrakennettuna joitakin annotaatioita, mutta uusien annotaatiotyyppien määritteleminen onnistuu hyvin samaan tapaan kuin tavallisten Javan rajapintojenkin. Ohjelmistokehykset tarjoavatkin usein omia annotaatioitaan, joilla luokkaan

pystytään kätevästi lisäämään kehyksen tarjoamaa toiminnallisuutta erillisten konfigurointitiedostojen sijaan. Esimerkiksi kohdassa 2.3 esiteltävä JDO-ohjelmointirajapinta määrittelee omia annotaatioita tähän tarkoitukseen.

Ohjelmassa Ohjelma 1 on esimerkki Javaan sisäänrakennetun @O*verride*-annotaation käytöstä.

```
public class MyClass {
      @Override
      public void overriddenMethod() {
      }
}
```

Ohjelma 1. *Javan annotaatioiden käyttö.*

Annotaatiot alkavat aina @-merkillä. Esimerkkiohjelmassa *@Override*-annotaatio koskee *overriddenMethod()*-metodia. Sen tarkoitus on ilmoittaa Java-kääntäjälle, että kyseisen metodin on tarkoitus ylikirjoittaa sen toteuttaman rajapinnan tai kantaluokan kyseinen metodi. Koska esimerkkiohjelmassa *MyClass*-luokalla ei ole kantaluokkaa, eikä se myöskään toteuta mitään rajapintaa, jossa kyseinen metodi olisi esitelty, antaisi kääntäjä tästä virheilmoituksen.

2.2 JUnit

JUnit [10] on yksikkötestauskehys Java-ohjelmointikielelle. Tässä työssä käytetty JUnit versio on 4.10. JUnit-kehyksessä testit jaetaan luokkiin, jotka sisältävät *@Test*-annotoituja metodeja. Jokainen tällainen metodi on yksi testitapaus. Testitapauksia sisältävät luokat voidaan vielä jakaa omiin *TestSuite*-kokonaisuuksiin. Tavallisesti jokaista ohjelman testattavaa luokkaa kohden luodaan sitä vastaava JUnit-testiluokka, jossa on testitapaukset testattavan luokan metodeille. Sitten yhteenkuuluvat JUnit-testiluokat voidaan vielä jakaa *TestSuite*-kokonaisuuksiin. Yksi *TestSuite*-kokonaisuus voi esimerkiksi sisältää ohjelman jonkin tietyn ominaisuuden testaavat luokat. On myös mahdollista määritellä JUnit-kehyksen annotaatioita käyttämällä testiluokassa metodit, jotka suoritetaan aina ennen testin suoritusta ja testin suorituksen jälkeen ja metodit, jotka suoritetaan ennen ensimmäistä testiä ja viimeisen testin jälkeen. Näissä voidaan tehdä testien tarvitsemat alustus- ja siivoustoimenpiteet.

Ohjelmassa Ohjelma 2 on esimerkki JUnit-testiluokasta, jossa testataan *MyClass*-luokan toimintaa. *Setup()*-metodissa tehdään ensin testin vaatimat alustustoimenpiteet. Itse testitapaus on *@Test*-annotaatiolla merkitty *test()*-metodi, jossa kutsutaan testattavan luokan tarjoamaa *doSomething()*-metodia ja tarkistetaan, että tulos on haluttu. Lopuksi *tearDown()*-metodissa tehdään siivoustoimenpiteet testin suorituksen jälkeen.

```
public class MyClassTest {
	private MyClass testObject;

	@Before
	public void setup() {
		testObject = new MyClass();
	}


	@Test
	public void test() {
		int value = testObject.doSomething(1);
		Assert.assertEquals(1, value);
	}

	@After
	public void tearDown() {
		...
	}
}
```

Ohjelma 2. *Testaus JUnit-yksikkötestauskehystä käyttäen.*

2.3 JDO

2.3.1 JDO-standardi

JDO (Java Data Objects) [1] on standardi tapa käyttää POJO-olioita (Plain Old Java Object) sovelluksen tietokantaan tallennetun datan tallentamiseen ja hakemiseen. JDO:ta käytettäessä tietokantaan tallennettavien Java-luokkien olioiden ei tarvitse toteuttaa mitään rajapintoja tai periytyä jostain tietystä kantaluokasta. Java-luokkien persistoituvuusasetukset määritellään XML-metadatatiedostossa. Vaihtoehtoisesti samat metadatamääritykset voidaan lisätä Java-luokille myös käyttäen persistoituviksi haluttavien Java-luokkien lähdekoodissa JDO-ohjelmointirajapinnan tarjoamia Java-annotaatioita. Tässä työssä käytetty JDO-toteutus on DataNucleus versio 3.1.0.

JDO-toteutuksen tarjoama JDO-enhancer-prosessi käsittelee luokkaa käännösvaiheessa sen annotaatioilla tai XML-metadatatiedostossa annettujen määritysten perusteella, mahdollistaen sen olioiden tallennuksen tietokantaan. JDO:n etuina on, että toteuttajan ei tarvitse toteuttaa olioiden persistointiin tarvittavaa infrastruktuuria, vaan toteuttaja määrittelee vain metadatalla sovelluksen persistoituvat Java-luokat. JDO hoitaa näiden Java-luokkien olioiden persistoinnin tietokantaan läpinäkyvästi. JDO-spesifikaatio sisältää myös transaktioiden hallinnan, jolloin useista yksittäisistä tietokantaoperaatioista voidaan muodostaa yksi atominen kokonaisuus, olettaen JDO:n peittämä tietokanta tukee transaktioita.

Tehokkuussyistä JDO-olioita tietokannasta ladattaessa ei kaikkia olion kenttiä ladata heti [1]. JDO tarjoaakin mahdollisuuden määrittää Java-luokille hakuryhmiä (fetch group), joissa määritellään mitkä luokan kentät ladataan samalla, kun kyseisen luokan olio ladataan tietokannasta.

Ohjelmakoodissa Ohjelma 3 on esimerkki JDO persistoituvasta Java-luokasta, jossa luokan persistoituvuusasetukset on määritelty JDO-annotaatioilla. *PersistentClass*-luokka on määritelty persistoitavaksi *@PersistenceCapable*-annotaatiolla. Lisäksi luokan oliot on määritelty irroitettaviksi (detachable). Tämä tarkoittaa, että kun luokan olio on ladattu tietokannasta, voidaan oliota käsitellä esimerkiksi muokkaamalla kenttien arvoja ilman, että muutokset heijastuisivat samalla tietokantaan. Tällä tavalla JDO:n persistoituvuusgraafista irrotettu olio voidaan kiinnittää siihen takaisin myöhemmin, jolloin muutokset tallentuvat tietokantaan.

Osa luokan kentistä, kuten *createDate*-kenttä, on määritetty persistoituviksi *@persistent*-annotaatiolla. Nämä kentät tallentuvat tietokantaan. *notPersistentField*-kenttä sen sijaan on merkitty *@NotPersistent*-annotaatiolla, jolloin kyseinen kenttä ei tallennu ollenkaan tietokantaan. Kaikki luokan persistoituvat kentät on merkitty oletuksena ladattavaksi (defaultFetchGroup = ”true”) samalla, kun luokan olio ladataan tietokannasta. Koska luokalle ei ole määritelty *@PrimaryKey*-annotaatiolla pääavainta, luo JDO-toteutus luokan tietokantakuvaukselle surrogaattiavaimen [4]. Surrogaattiavain tarkoittaa keinotekoista yksikäsitteistä tunnistetta tietokantaoliolle, jota ei ole johdettu mitenkään olion datasta. Surrogaattiavain voi olla esimerkiksi juokseva numerointi tai satunnainen merkkijono.

```
@PersistenceCapable(detachable = "true")
public class PersistentClass {
      //A field that will be persisted in the database
      //and fetched when an object is loaded
      @Persistent(defaultFetchGroup = "true")
      private int persistentField;

      //A field that won't be persisted in the database
      @NotPersistent
      private int notPersistentField;

      @Persistent(defaultFetchGroup = "true")
      private EmbeddedPersistentClass embeddedObject;

      @Persistent(defaultFetchGroup = "true")
      private Date createDate;
      ...
}
```

Ohjelma 3. *JDO persistoituva Java-luokka.*

JDO:n tavoitteena on lisäksi mahdollistaa sovelluksen riippumattomuus alla olevasta tietokannasta [1]. JDO-apin näkökulmasta ei ole väliä onko tietovarastona relaatiotietokanta, dokumenttipohjainen tietokanta vai jokin muu, sillä se käsittelee vain perinteisiä Java-luokkia. Esimerkiksi tässä työssä käytetty JDO-toteutus DataNucleus [4] tukee sekä relaatiotietokantoja, että joitakin yleisimmistä NoSQL-tietokannoista, kuten MongoDB [11].

Koska JDO:n käyttäjä ei suoraan määrittele tietokannan skeemaa, muodostuu skeemaksi käytännössä JDO persistoituvien luokkien luokkarakenne ja niiden metadatamäärittelyt. Käytettäessä dokumenttipohjaista tietokantaa, kuten MongoDB, muodostuu kokoelman dokumentit siis luokan persistoituviksi määritellyistä kentistä.

Olioiden persistointi ja hakeminen tietokannasta tapahtuu käyttäen JDO-toteutuksen tarjoamaa *PersistenceManager*-rajapintaa. Ohjelmassa Ohjelma 4 on esitetty kuinka *PersistenceManager*-rajapinnan käyttämä tietokanta konfiguroidaan ja *PersistentClass*-luokan olio tallennetaan tähän tietokantaan käyttäen JDO:n tarjoamia transaktioita.

```
Properties properties = new Properties();
properties.setProperty("javax.jdo.PersistenceManagerFactoryClass",
                "org.datanucleus.api.jdo.JDOPersistenceManagerFactory");
properties.setProperty("javax.jdo.option.ConnectionURL",
"jdbc:mysql://localhost/myDB");
properties.setProperty("javax.jdo.option.ConnectionDriverName",
"com.mysql.jdbc.Driver");
properties.setProperty("javax.jdo.option.ConnectionUserName","login");
properties.setProperty("javax.jdo.option.ConnectionPassword","password");
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory( prop-
erties );

PersistenceManager pm = pmf.getPersistenceManager();
Transaction tx = pm.currentTransaction();
try {
            tx.begin();
            PersistentClass obj = new PersistentClass();
            pm.makePersistent(obj);
            tx.commit();
}
catch (Exception e) {
            ...
}
finally {
      if (tx.isActive()) {
            // Error occurred so rollback the transaction
            tx.rollback();
      }
      pm.close();
}
```

Ohjelma 4. *JDO:n PersistenceManager-rajapinnan käyttö. Mukailtu lähteestä [4].*

Ohjelmassa määritetään ensin käytetyn tietokannan yhteysasetukset, jonka jälkeen luodaan *PersistenceManagerFactory* näillä asetuksilla. *PersistenceManager*-olio aloittaa uuden transaktion. Jokaiseen *PersistenceManager*-olioon liittyy oma transaktio-instanssinsa, jota käyttäen transaktio aloitetaan (begin), lopetetaan (commit tai rollback) ja voidaan säätää transaktion asetuksia, kuten esimerkiksi eristyvyystasoa (isolation level). Transaktion tila muuttuu aloitettaessa aktiiviseksi ja lopetettaessa jälleen epäaktiiviseksi. Transaktiossa luodaan uusi *PersistentClass*-olio ja tallennetaan se tietokantaan *pm.makePersistent(obj)*-kutsulla. Lopuksi tarkastetaan, että transaktioon sitoutuminen (*tx.commit*) onnistui ja suljetaan *PersistenceManager*. Virhetilanteessa perutaan transaktio kutsulla *tx.rollback()*.

2.3.2 JDOQL-kyselykieli

JDO määrittelee oman kyselykielen, JDOQL (JDO Query Language), persistoitujen Java-olioiden tietokannasta hakemista varten [1]. JDOQL muistuttaa rakenteeltaan paljon SQL-kyselykieltä, mutta on oliopohjainen. Tämä tarkoittaa sitä, että JDOQL-kyselyn kirjoittaja pystyy käyttämään tietokantahauissa esimerkiksi parametreina samoja Java-luokkia kuin sovelluslogiikassakin. Kieli on pyritty tekemään mahdollisimman

intuitiiviseksti Java-kehittäjille. Listauksessa Ohjelma 5 on esitetty JDOQL-kyselyn rakenne.

```
SELECT [UNIQUE] [<result>] [INTO <result-class>]
        [FROM <candidate-class> [EXCLUDE SUBCLASSES]]
        [WHERE <filter>]
        [VARIABLES <variable declarations>]
        [PARAMETERS <parameter declarations>]
        [<import declarations>]
        [GROUP BY <grouping>]
        [ORDER BY <ordering>]
        [RANGE <start>, <end>]
```

Ohjelma 5. *JDOQL-kyselyn rakenne [1].*

JDOQL-kyselykielen suhde Java-ohjelmointikieleen näkyy mm. kyselyrakenteen *import declarations*-kohdasta. Tässä kohdassa täytyy luetella Java-koodin tapaan import-lausekkeilla kyselyssä käytettyjen olioiden luokat, jotka eivät ole perustietotyyppejä (Integer, Boolean jne.) ja eivät ole java.lang -pakkauksessa [1]. Tällaisia voivat olla esimerkiksi parametreina käytettyjen olioiden luokat. Ohjelmassa Ohjelma 6 esitetyssä yksinkertaisessa JDOQL-esimerkkikyselyssä haetaan tietokannasta kaikki *PersistentClass*-luokan (Ohjelma 3) instanssit, joilla *persistentField*-kentän arvo on 1 ja *createDate*-kentän arvo on suurempi kuin kyselyssä parametrina annettu *date_limit* päivämäärä, joka on Javan java.util.Date-tyyppiä. Esimerkissä on käytetty eksplisiittistä parametrien määrittelyä, mutta PARAMETERS-kohdan voisi myös jättää kokonaan pois ja käyttää implisiittisiä parametreja. Tällöin parametrit merkittäisiin kaksoispiste-etuliitteellä, esimerkiksi *:date_limit*. Lopuksi tulosinstanssit järjestetään luontiajan mukaan. Koska SELECT-lauseke on jätetty tyhjäksi, palauttaa kysely Javan List-tyyppisen listan kokonaisia PersistentClass-luokan olioita.

```
SELECT
FROM mydomain.PersistentClass
WHERE persistentField == 1 && createDate > date_limit
PARAMETERS Date date_limit
import java.util.Date
ORDER BY createDate DESC
```

Ohjelma 6. *Esimerkki JDOQL-kyselystä.*

Ohjelmassa Ohjelma 7 on esitetty kuinka ohjelman Ohjelma 6 kysely suoritetaan JDO:n *PersistenceManager*-rajapintaa käyttäen, kun käytetty *queryString*-muuttuja on alustettu kyseisellä kyselymerkkijonolla.

```
String queryString = ...;
Date date = new Date();
Query q = pm.newQuery(queryString);
List results = (List)q.execute(date);
```

Ohjelma 7. *JDOQL-kyselyn suoritus JDO-ohjelmointirajapintaa käyttäen.*

JDOQL-kyselyn SELECT-lausekkeen result-kohdassa voidaan määritellä mitä kyselyn halutaan palauttavan [1]. Result-kohdan voi jättää tyhjäksi, jolloin kysely palauttaa oletuksena FROM-lausekkeessa määritellyn kandidaattiluokan tyyppisiä olioita, kuten ohjelmassa Ohjelma 6. Result-kohta voi sisältää mm. seuraavanlaisia määrityksiä tulokselle:

- *this*, eli FROM-lausekkeessa määritellyn kandidaattiluokan instanssi. Jos result-kohta jätetään tyhjäksi, on *this* oletusarvo. Esimerkiksi seuraava kysely palauttaisi kaikki *PersistentClass*-luokan oliot.

```
SELECT this
FROM mydomain.PersistentClass
```

- Kandidaattiluokan kentän nimi. Esimerkiksi seuraava kysely palauttaisi kaikkien *PersistentClass*-luokan olioiden *persistentField*- ja *createDate*-kenttien arvot.

```
SELECT persistentField, createDate
FROM mydomain.PersistentClass
```

- Kentästä toiseen ketjuna navigoiva lauseke. Esimerkiksi seuraava kysely palauttaisi kaikkien *PersistentClass*-luokan olioiden *embeddedObject*-attribuuttiolioiden *field1*-kenttien arvot.

```
SELECT embeddedObject.field1
FROM mydomain.PersistentClass
```

- Erillinen tulosluokan olio, joka luodaan kyselyyn vastaukseksi valituista tuloskentistä. Esimerkiksi seuraava kysely antaisi tulokseksi *mydomain.ResultClass*-luokan olioita, joiden field1- ja field2-kenttien arvot on alustettu kyselyn tulokseksi saamilla arvoilla.

```
SELECT persistentField as field1, createDate as field2 INTO mydo-
main.ResultClass
FROM mydomain.PersistentClass
```

- Aggregaattifunktio (count(), avg(), sum(), min(), max()). Esimerkiksi seuraava kysely laskisi kuinka monta *PersistentClass*-luokan oliota tietokannassa on.

```
SELECT count(this)
FROM mydomain.PersistentClass
```

Edellä mainittuja määrityksiä voi myös yhdistellä keskenään result-kohdassa. Kyselyn tuloksen tyyppi riippuu siitä, mitä result-kohdassa on määritelty. Kyselyn tulostyyppi voi olla jokin seuraavista Java-tyypeistä [1]:

- *Object* palautetaan, kun kyselyn tuloksessa on vain yksi rivi ja yksi sarake. Esimerkiksi kun SELECT-lausekkeessa on käytetty UNIQUE-määrettä tai vain yhtä aggregaattifunktiota, kuten edellä olleessa esimerkissä.

- *Object[]* palautetaan, kun kyselyn tuloksessa on vain yksi rivi, mutta monta saraketta. Esimerkiksi jos result-kohdassa on enemmän kuin yksi aggregaattifunktio.
- *List<Object>* palautetaan, kun tuloksessa on useita rivejä, mutta vain yksi sarake. Esimerkiksi, jos result-kohdassa valitaan vain yksi kandidaattiluokan kentistä tai jätetään result-kohta tyhjäksi, kuten ohjelman Ohjelma 6 kyselyssä.
- *List<Object[]>* palautetaan, kun kyselyn tuloksessa on useita rivejä ja sarakkeita. Esimerkiksi, jos result-kohdassa valitaan palautettavaksi useampi kuin yksi kandidaattiluokan kentistä.

JDOQL-kyselykieli sisältää lisäksi joukon apumetodeita, joita voidaan käyttää kyselyissä käytettävien Java-tyyppien yhteydessä asettamaan rajoituksia kyselyn tulosjoukolle [1]. Näillä metodeilla voidaan esimerkiksi käsitellä kyselyissä käytettäviä säiliötä (Javan Collection) tai merkkijonoja (String). Tällaisia ovat mm. säiliöille *contains(value)*, jota voi käyttää JDOQL-kyselyssä samaan tarkoitukseen kuin SQL:ssä IN-operaattoria ja merkkijonoille *matches(String pattern)*, jota voi käyttää kuten SQL:n LIKE-operaattoria. Ohjelman Ohjelma 8 kyselyssä on esimerkki kuinka JDOQL-metodeita voidaan käyttää kyselyissä. Kyselyssä käytetään *contains(value)*-metodia SQL:n IN-operaattorin tapaan. Kysely palauttaa kaikki sellaiset *PersistentClass*-luokan oliot, joiden *persistentField*-kentän arvo löytyy *allowedValues*-joukosta, eli arvon täytyy olla 1 tai 2.

```
List<Integer> allowedValues = new ArrayList<Integer>();
allowedValues.add(1);
allowedValues.add(2);
Query query = pm.newQuery(
          "SELECT FROM mydomain.PersistentClass "
          + "WHERE allowedValues.contains(persistentField)"
          + "PARAMETERS List allowedValues"
          + "IMPORT java.util.List");
List results = (List)query.execute(allowedValues);
```

Ohjelma 8. *JDOQL-metodien käyttö kyselyssä.*

2.4 MongoDB-dokumenttitietokanta

NoSQL-tietokanta on yhteinen nimitys kaikille perinteisestä relaatiotietokantojen relaatiomallista poikkeaville tietokannoille [14]. Niissä ei siis ole käytössä relaatiokannoissa käytettävä kiinteä taulukkoskeema. NoSQL-tietokannat ovat kasvattaneet suosiotaan relaatiotietokantojen rinnalla ohjelmistotuotannossa. Eräs NoSQL-tietokantojen alakategoria on *dokumenttipohjaiset tietokannat*. Tässä työssä käytettävä MongoDB-tietokanta [11] on dokumenttipohjainen tietokanta. MongoDB:stä käytettiin tässä työssä versiota 2.4.5. Muita NoSQL-tietokantojen kategorioita ovat mm. *graafitietokannat* (graph database), *avain-arvo-tietokannat* (key-value database) ja *sarakepohjaiset tietokannat* (column-oriented database) [14].

Relaatiotietokannoissa tiedon rakenne (taulut, taulujen sarakkeet, taulujen vierasavain yhteydet jne.) määritetään etukäteen, jonka jälkeen kaiken tietokantaan tallennettavan datan täytyy noudattaa tätä rakennetta. Rakenteen muuttaminen jälkeenpäin voi osoittautua hankalaksi. Dokumenttipohjaiset tietokannat sen sijaan eivät seuraa mitään tällaista ennalta määritettyä kiinteätä taulukkoskeemaa, vaan ideana on, että skeema on dynaaminen. Tämä tekee dokumenttipohjaisista tietokannoista joustavampia, sillä tiedon rakenteen muuttaminen jälkeenpäin on helpompaa kuin relaatiotietokannoissa. Toisaalta järjestelmän tietojen eheydestä on vaikeampi varmistua. MongoDB-dokumenttitietokanta ei esimerkiksi tarjoa useista yksittäisistä tietokantaoperaatioista koostuvia atomisia transaktioita, kuten relaatiotietokannat. [14]

Dokumenttipohjaisissa tietokannoissa käsiteltävä data on relaatiotietokantojen taulujen ja rivien sijasta dokumentteja. Dokumentin määritelmä voi olla erilainen riippuen dokumenttipohjaisen tietokannan toteutuksesta. MongoDB-tietokannassa dokumentit ovat JSON-tyylisiä [5] tietorakenteita muodostuen avain-arvo pareista [11]. Sisäisesti MongoDB käyttää JSON-dokumenttien esittämiseen binäärimuotoisesta formaattia nimeltä BSON (Binary JSON) [3], joka laajentaa JSON-mallia uusilla tietotyypeillä. JSON-formaattiin verrattuna BSON on myös tehokkaampi tallennustilan ja skannausnopeuden suhteen. Muita dokumenttipohjaisten tietokantajärjestelmienkäyttämiä dokumenttiformaatteja ovat mm. XML-muotoiset dokumentit.

Samankaltaiset tai muuten yhteenkuuluvat dokumentit muodostavat MongoDB:ssä kokoelmia (collection). Jos verrataan MongoDB-tietokantaa relaatiotietokantoihin, voidaan kokoelman ajatella olevan relaatiotietokannan taulu ja kokoelman dokumenttien olevan relaatiotietokannan rivejä ja dokumenttien kenttien olevan relaatiotietokannan rivin sarakkeita. Erona kuitenkin on, että MongoDB-dokumentit ovat heterogeenisiä, eikä dokumenttien rakennetta ole sidottu ennalta määriteltyyn skeemaan. Tämä tarkoittaa, että jopa saman kokoelman sisällä olevien dokumenttien rakenne saattaa erota toisistaan. [11] Ohjelmassa Ohjelma 9 on esimerkki tästä dokumenttien heterogeenisyydestä. Ohjelmassa lisätään MongoDB:ssä tietokannan *persons*-kokoelmaan kaksi dokumenttia. Henkilöä kuvaavat dokumentit eroavat toisistaan rakenteeltaan, mutta pystyvät silti sijaitsemaan samassa MongoDB-tietokannan kokoelmassa.

```
db.person.insert({
      firstname: "Alan",
      lastname: "Turing",
      birthDate: new Date('Jun 23, 1912'),
      deathDate: new Date('Jun 07, 1954')
});
db.person.insert({
      firstname: "John",
      lastname: "Smith",
      hobby: "Football"
});
```

Ohjelma 9. *Dokumenttien luominen MongoDB-tietokannan kokoelmaan.*

Ainoa vaadittava ominaisuus dokumentilla on _id-kenttä, joka on dokumentin yksikäsitteinen tunniste [11]. MongoDB luo tämän kentän automaattisesti, jos sitä ei erikseen insert-lauseessa määritetä.

MongoDB-tietokannassa voi optimoida tiedonhakua luomalla indeksejä [11]. MongoDB-tietokannan indeksien luonti vastaa periaatteiltaan täysin relaatiotietokantojen indeksejä. Indeksin voi luoda kokoelmassa mihin tahansa dokumentin kenttään tai joukkoon kenttiä, aivan kuten relaatiotietokannassa voi luoda indeksin taulun sarakkeisiin. Haut indeksoitujen kenttien perusteella nopeutuvat, koska MongoDB:n ei tarvitse skannata kaikkia kokoelman dokumentteja läpi. Kokoelman dokumenttien heterogeenisyyden vuoksi kaikilla kokoelman dokumenteilla ei välttämättä ole indeksoitua kenttää. Tällöin indeksiin tallentuu vain null-arvo kyseisille dokumenteille.

MongoDB-tietokannassa ei ole mahdollista määritellä dokumenttien välille relaatiotietokantojen vierasavainviittauksien tyylisiä rajoitteita [11]. Sen sijaan dokumenttien välisten suhteiden määrittämiseen voidaan MongoDB-tietokannassa käyttää kahta eri tapaa. Ensimmäinen vaihtoehto on dokumenttien upottaminen dokumentin aladokumenteiksi [11]. Toinen vaihtoehto taas muistuttaa relaatiotietokantojen vierasavainviittauksia, mutta siihen ei sisälly samalla tavalla tietokannan valvomia rajoitteita, kuten viiteeheys. Tässä vaihtoehdossa dokumentissa määritellään yksi kenttä, joka viittaa esimerkiksi jonkin toisen dokumentin id-kenttään [11]. Ohjelmassa Ohjelma 10 on esimerkki, kuinka sama tietojen välinen suhde voidaan esittää molemmilla eri tavoilla. Ensimmäisessä tapauksessa asunnon omistajaan viitataan omistajadokumentin id:llä, jota käyttäen pystytään hakemaan omistajan tiedot. Jälkimmäisessä tapauksessa omistajan tiedot on upotettu suoraan asuntodokumenttiin.

```
db.apartment.insert(
  {
    rooms: 3,
    ownerId: new ObjectId("5099803df3f4948bd2f98391")
  }
);
db.apartment.insert(
  {
    rooms: 3,
    owner: { firtname: "Alan", lastname: "Turing" }
  }
);
```

Ohjelma 10. *MongoDB-dokumenttien väliset suhteet.*

Upottamisen etuna on, että lukuoperaatioiden suorituskyky voi olla parempi, jos upotettua dokumenttia käytetään paljon sen dokumentin yhteydessä, johon se on upotettu, eikä usein yksinään. Jos tällaisessa tapauksessa kaikki dokumenttiin liittyvä data sisällytetään yhteen samaan dokumenttiin useisiin dokumentteihin jakamisen sijaan, tarvitaan usein vähemmän kyselyjä tiedon hakemiseen, kuin viittauksia käytettäessä. Upottaminen myös mahdollistaa dokumenttiin liittyvien tietojen päivittämisen yhtenä atomisena operaationa [11]. MongoDB-tietokannasta nimittäin puuttuu tuki useista tietokantaoperaatioista koostuville ja useisiin eri dokumentteihin vaikuttaville atomisille transaktioille. Edellisten operaatioiden peruminen voi olla hyvin vaikeaa, kun yksi operaatio epäonnistuu. Relaatiotietokannat tarjoavat tällaiset transaktiot ja rollback-mahdollisuuden, mutta MongoDB-tietokannassa tämä jää tietokantaa käyttävän sovelluksen vastuulle. Yhteen dokumenttiin kohdistuvat operaatiot ovat kuitenkin aina atomisia [11].

Tällaisella tietokannan denormalisoinnilla on myös haittapuolensa. Tieto saattaa monistua, mikä hankaloittaa tiedon päivityksiä. Esimerkiksi jos ohjelmassa Ohjelma 10 sama henkilö omistaa useita asuntoja ja henkilön tiedot on sisällytetty *apartment*-dokumenttiin, täytyy henkilön tietojen muuttuessa päivittää kaikki *apartment*-dokumentit, joissa kyseinen henkilö on omistajana. Ongelmaksi voi myös muodostua dokumenttien paisuminen kooltaan suuriksi.

Vierasavaintyyppisiä dokumenttiviittauksia kannattaa siis käyttää laajojen ja monimutkaisten tietohierarkioiden esittämiseen, joissa upottamista käytettäessä dokumentit paisuisivat liian suuriksi tai tiedon monistuminen aiheuttaisi ongelmia. Vierasavaintyyppisiä dokumenttiviittauksia kannattaa myös käyttää silloin, kun dokumentteja käytetään paljon yksinään, eikä jonkin muun dokumentin käytön yhteydessä. Tämä voi kuitenkin tehdä tiedon hakuoperaatioista hitaampia, sillä sovellus saattaa joutua tekemään useita jatkokyselyitä dokumenttiviittausten seuraamiseksi.

Relaatiotietokannoissa kyselykielenä käytetään yleensä SQL-kieltä CRUD-operaatioihin (Create, Read, Update, Delete). Koska dokumenttipohjaiset tietokannat saattavat erota toisistaan huomattavasti riippuen toteutuksesta, ei ole olemassa SQL:n kaltaista standardia kyselykieltä, jota voitaisiin käyttää kaikissa dokumenttipohjaisissa

tietokannoissa. Dokumenttipohjaiset tietokannat tarjoavatkin toteutuksesta riippuen oman kyselykielensä tai ohjelmointirajapinnan CRUD-operaatioiden suorittamiseksi [14].

MongoDB-tietokannan kyselyissä käytetään sen omaa kyselykieltä [11]. Myös kyselyt ovat rakenteeltaan JSON-tyylisiä, aivan kuten tietokantaan tallennettavat dokumentitkin. Kyselyn kohteena on aina jokin tietty kokoelma. Kokoelmien liitokset kyselyissä SQL:n taulujen JOIN-liitosten tapaan eivät ole mahdollisia. Jos täytyy hakea kahdesta eri kokoelmasta löytyvien dokumenttien tietoja, tarvitaan tähän kaksi eri kyselyä, joissa ensimmäisessä haetaan tarvittavat tiedot toiseen kyselyyn. Samaan tapaan myös SQL-tyyliset sisäkkäiset kyselyt eivät ole MongoDB-kyselykielessä mahdollisia, vaan tarvitaan useita peräkkäisiä kyselyitä. Tämä on mahdollista välttää käyttämällä edellä mainittua dokumenttien upottamista dokumenttien sisään tai lisäämällä kyselyissä usein tarvittavat kentät molempien kokoelmien dokumentteihin. Tällöin täytyy kuitenkin muistaa tästä denormalisoinnista aiheutuvat haitat. Ohjelmassa Ohjelma 11 esitetyssä MongoDB-kyselyssä haetaan kaikki ohjelman Ohjelma 9 *person*-dokumentit, joissa henkilön sukunimi on ”Turing” ja etunimi ”Alan”. Tulosdokumentit järjestetään nousevaan järjestykseen syntymäajan mukaan.

```
db.person.find({
  $query:
      {
            lastname: "Turing",
            firstname: "Alan"
      },
  $orderby:
      {
            birthDate: 1
      }
});
```

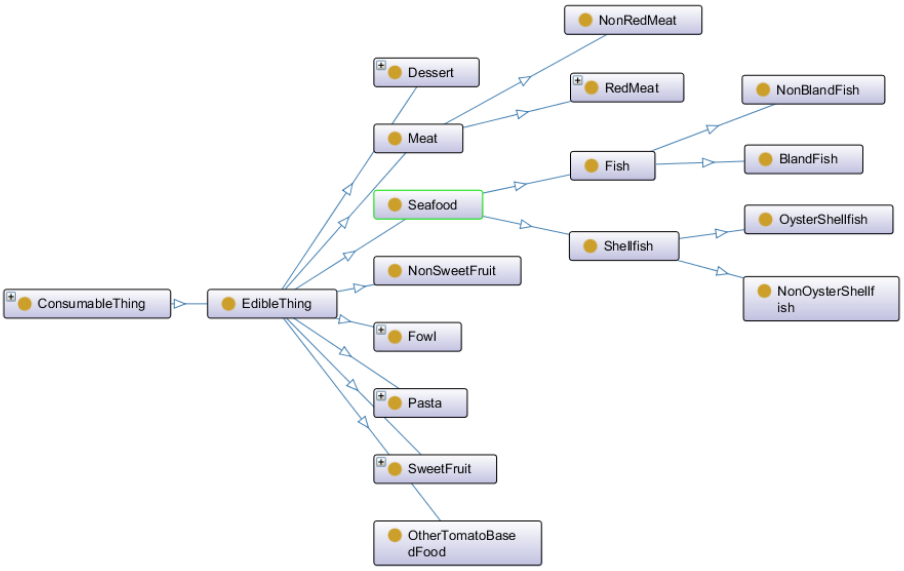
Ohjelma 11. *Hakuesimerkki MongoDB-kyselykielellä.*

Kuten ohjelmasta Ohjelma 11 on huomattavissa, noudattaa MongoDB-kyselykieli QBE (Query by Example)-konseptia [24]. QBE-konseptin kyselyissä kysely itsessään mallintaa tulokseksi haluttavaa kohdetta. Eli koska tavoitteena on löytää dokumentti, jossa *lastname*-kentän arvo on ”Turing” ja *firstname*-kentän arvo ”Alan”, annetaan kyselylle tällainen esimerkkidokumentti jättäen muiden kenttien arvot määrittelemättä. Tuloksena saadaan tätä esimerkkiä noudattavat dokumentit, jotka toki sisältävät näiden kenttien lisäksi myös kaiken muun kyseessä olevan dokumentin datan.

3. ONTOLOGIAT

3.1 Ontologioiden esittely

Ohjelmistotieteessä *ontologialla* tarkoitetaan jonkin tietyn sovellusalueen käsitteiden ja niiden välisten suhteiden formaalia kuvausta [1]. Ontologioita hyödyntäen voidaan sovelluksen tietosisältö kuvata niin, että myös tiedon käyttötarkoitus ja merkitys on koneellisesti tulkittavissa. Ontologiat esitetään suunnattuina graafeina. Kuvassa 3.1 on esitetty esimerkkinä pieni osajoukko W3C:n (The World Wide Web Consortium) luomasta viiniontologiasta [22], joka kuvaa syötävien asioiden ontologiaa. Kokonainen viiniontologia olisi liian iso tässä yhtenä kuvana esitettäväksi.



Kuva 3.1: Syötävien asioiden ontologia. Perustuu lähteeseen [22].

Kuvan 3.1 ontologiagraafin kantakäsitteenä on *ConsumableThing*-käsite. Käsitteiden välille on ontologiassa määritelty *subClassOf*-suhteita, jotka on esitetty kuvan graafissa käsitesolmuja yhdistävinä kaarina. Nämä käsitteiden väliset *subClassOf*-suhteet määrittävät ontologian käsitehierarkian. *ConsumableThing*-käsitteellä on alakäsite *EdibleThing*, jolla taas on mm. alakäsitteet *Seafood*, *Meat*, *Pasta* jne. Käsitteillä voi olla myös muunlaisia suhteita muihin ontologian käsitteisiin. Esimerkiksi viiniontologiassa käsitteellä *MealCourse* on vielä *hasFood*-suhde *EdibleThing*-käsitteeseen.

Vaikka ontologia olisi suunniteltu jonkin tietyn sovelluksen käyttöön, pystytään sitä käyttämään uudelleen missä tahansa sovelluksessa, joka liittyy aihealueeseen, jonka käsitteistön ontologia kuvaa. Nämä sovellukset pystyvät jakamaan ja uudelleen käyttä-

mään aihealueen tietämystä, sekä toimimaan yhdessä aihealuetta varten suunnitellun ontologian avulla [1].

Eräs ontologiapohjaisen sovelluksen hyödyistä on myös mahdollisuus käyttää ns. *päättelijöitä* (reasoner) lisäinformaation päättelemiseksi mallinnettavan aihealueen käsitteistä [21]. Esimerkiksi kuvan 3.1 ontologian käsitteiden välisistä suhteista voisi päättelijätoteutus automaattisesti tehdä lisäpäätelmän, että osteri (*OysterShellfish*) on simpukka (*Shellfish*), joka taas on meren antimia (*Seafood*) ja syötävää (*EdibleThing*). Päättelijätoteutus voi lisätä tämän tiedon ontologian suhdejoukkoon, jolloin se voidaan ottaa huomioon hauissa ontologiasta, vaikka sitä ei alkuperäisessä ontologiadatassa suoraan ollutkaan.

Ontologiat ja niiden kuvaus- ja kyselykielet ovat eräitä *semanttisen webin* keskeisistä toteutusteknologioista [20]. Semanttisen Webin perusperiaatteena on pelkkien linkitettyjen dokumenttien sijaan mahdollistaa verkosto linkitettyä data. Verkkosivut ovat tällä hetkellä tarkoitettu ihmisten luettavaksi ja verkkosivulla olevan tiedon merkitys ei ole koneellisesti pääteltävissä. Käyttämällä ontologioita voidaan dataa rikastaa lisäämällä siihen myös tietoa sen merkityksestä ja suhteista muuhun dataan. Kun tiedon merkitys on koneellisesti tulkittavissa, mahdollistaa tämä entistä älykkäämpien sovelluksien kehittämisen, jotka pystyvät myös päättelemään lisäinformaatiota datasta, kuten aikaisemmassa ontologian päättelijäesimerkissä. Tämä helpottaa tiedon hakemista, tiedon yhdistelemistä eri lähteistä ja tiedon jakamista sovellusten välillä koneellisesti.

3.2 Ontologian kuvauskielet

Riippumatta kielestä, jolla ontologia kuvataan, muodostuvat ontologiat seuraavista yleisistä komponenteista.

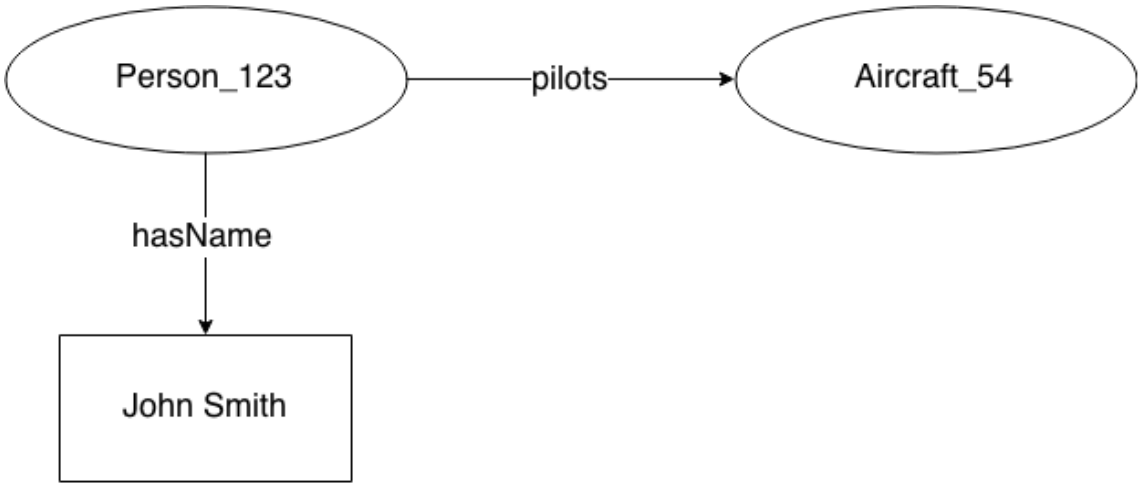
- Ontologialuokat, eli ontologian käsitteet
- Luokkien ominaisuudet
- Luokkien väliset suhteet
- Luokkien ilmentymät (individual), tässä työssä käytetään termiä luokan instanssi

RDF (Resource Description Framework) [19] on W3C:n suosittelema standardi tiedon mallintamiseen ja esittämiseen *resursseina*. Resurssi voi olla mikä tahansa tunnistettavissa oleva käsite, esimerkiksi henkilö. RDF:ssä kaikki tieto mallinnetaan *subjektista*, *predikaatista* ja *objektista* muodostuvilla lausekkeilla (triple RDF-terminologiassa). Joukko tällaisia RDF-lausekkeita yhdessä muodostavat suunnatun graafin. Subjekti on resurssin tunniste, IRI (internationalized resource identifier). IRI laajentaa URI-nimeämisrakennetta mahdollistamalla laajemman merkkijoukon käytön tunnisteessa, esimerkiksi kyrillisen kirjaimiston käytön. Predikaatti on jonkin resurssia kuvaavan ominaisuuden (property) IRI ja tarkoittaa suhdetta subjektin ja objektin välillä. Objekti on arvo, joka resurssilla on kyseiselle ominaisuudelle. Objekti voi olla joko literaali tai

jonkin toisen resurssin IRI. Tämä työn tekstissä kolmikoista käytetään notaatiota <s,p,o>, missä s on kolmikon subjekti, p on kolmikon predikaatti ja o on kolmikon objekti. Kuvassa 3.2 on esimerkki taulukon Taulukko 1 RDF-kolmikoista muodostuneesta ontologiagraafista.

Taulukko 1. *RDF-kolmikoita*

Subjekti	**Predikaatti**	**Objekti**
Person_123	hasName	”John Smith”
Person_123	pilots	Aircraft_54



Kuva 3.2: Taulukon Taulukko 1 RDF-kolmikoista muodostuva graafi

Kolmikon <*Person_123,hasName,John Smith*> objekti on literaali, kun taas kolmikon <*Person_123,pilots,Aircraft_54*> objekti on Aircraft-tyyppisen resurssin tunniste.

RDFS (Resource Description Framework Schema) [18] on W3C:n kehittämä merkkauskieli ontologioiden esittämiseen. Se käyttää RDF-mallia ja määrittelee ontologioiden kuvauksessa tarvittavan perussanaston. Se määrittelee mm. seuraavat tärkeät lisämääreet predikaatteina käytettäville ominaisuuksille:

- **rdfs:domain** määrittää ominaisuudelle, että kolmikon subjektin on kuuluttava johonkin tiettyyn ontologialuokkaan.
- **rdfs:range** määrittää ominaisuudelle, että kolmikon objektin on kuuluttava johonkin tiettyyn ontologialuokkaan tai oltava jotain tiettyä tietotyyppiä.
- **rdfs:subClassOf** mahdollistaa luokkahierarkioiden muodostamien ontologialuokista

RDFS:n päälle on rakennettu semanttisesti vahvempia laajennoksia. Yksi näistä on OWL (Web Ontology Language) [16]. OWL mahdollistaa monien lisämääreiden määrittämisen ontologian komponenteille. Tarkoituksena on laajentaa RDFS:n ilmaisuvoimaa, mikä mahdollistaa tiedon merkityksen määrittelemisen entistä paremmin. OWL

mm. jakaa ontologialuokkien ominaisuudet kahteen alakategoriaan: tietotyyppiomaisuuksiin (datatype property) ja suhdeominaisuuksiin (object property). Tarkoituksena on tehdä ero luokkien ominaisuuksien välillä sen mukaan onko kolmikon objektina literaali vai resurssin IRI. Tietotyyppiominaisuuksissa kolmikon subjekti on aina resurssin IRI ja objekti literaali. Suhdeominaisuuksissa sekä kolmikon subjekti että objekti ovat aina IRI-tunnisteita. Esimerkiksi OWL-kielellä määriteltynä kuvan 3.2 ominaisuus *hasName* on tietotyyppiominaisuus, sillä sen objekti on literaali ”John Smith”. *Pilots* taas on suhdeominaisuus, sillä sen objekti on instanssin IRI-tunniste. Se siis määrittää kahden instanssin välisen suhteen.

OWL:ssa ominaisuuksille voidaan myös antaa minimi ja maksimi kardinaliteetti [16]. Tällä voidaan määritellä kuinka monta toisistaan semanttisesti poikkeavaa arvoa tietyllä luokan instanssilla voi kyseiselle ominaisuudelle vähimmillään ja enimmillään olla. Esimerkiksi ohjelmassa Ohjelma 12 OWL-kielisen XML-dokumentin kohdassa määritetään, että henkilöllä täytyy olla vähintään yksi vanhempi ja voi myös olla enintään 2 vanhempaa.

```
<owl:Restriction>
  <owl:onProperty rdf:resource="#hasParent" />
  <owl:maxCardinality
rdf:datatype="&xsd;nonNegativeInteger">2</owl:maxCardinality>
  <owl:minCardinality
rdf:datatype="&xsd;nonNegativeInteger">1</owl:minCardinality>
</owl:Restriction>
```

Ohjelma 12. *Ontologian ominaisuuksien kardinaliteettien määrittäminen OWL-ontologiakielellä.*

Lisäksi OWL:ssa omaisuuden voidaan merkitä olevan *transitiivinen*, *symmetrinen*, *funktionaalinen* tai *käänteisfunktionaalinen* [16]. Transitiivinen ominaisuus muodostaa hierarkisen ketjun ontologialuokkien instanssien välille. Eräs esimerkki transitiivisesta ominaisuudesta voisi olla maantieteellisten paikkojen välillä oleva *isLocatedIn*-suhdeominaisuus. Jos siis on olemassa kolmikko <*Helsinki, isLocatedIn, Finland*> ja kolmikko <*Finland, isLocatedIn, Europe*> ja *Helsinki*, *Finland* ja *Europe* ovat alueita, voidaan tällöin päätellä, että myös *Helsinki*- ja *Europe*-instanssien välillä on voimassa suhdeominaisuus *isLocatedIn* (kolmikko <*Helsinki, isLocatedIn, Europe*>).

Symmetrisellä ominaisuudella tarkoitetaan, että suhdeominaisuus on voimassa molempiin suuntiin. Kolmikosta <*John, friendOf, Peter*> voitaisiin siis päätellä kolmikko <*Peter, friendOf, John*>, jos *friendOf*-suhdeominaisuus on määritelty symmetriseksi.

Funktionaaliseksi määritelty ominaisuus tarkoittaa, että jokaisella instanssilla voi olla kyseiselle ominaisuudelle tasan yksi uniikki kohde. Jos siis instanssilla esiintyy sama funktionaalinen ominaisuus useaan kertaan, voidaan ominaisuuden kohteiden päätellä kuvaavan samaa asiaa. Käänteisfunktionaalinen ominaisuus tarkoittaa, että ominaisuuden käänteisominaisuus on funktionaalinen. Käänteisominaisuudella taas tarkoitetaan

ominaisuutta, joka on määritelty jonkin toisen ominaisuuden käänteiseksi OWL-kielen *inverseOf*-merkinnällä. Esimerkiksi ohjelmassa Ohjelma 12 esiintyvälle *hasParent*-ominaisuudelle voitaisiin määrittää käänteisominaisuudeksi *hasChild*-ominaisuus. Päättelijätoteutus voisi tällöin aina nähdessään kolmikon <A,hasParent,B> päätellä kolmikon <B,hasChild,A>, kun A ja B ovat henkilökäsitteen instanssien tunnisteita. Nämä OWL-ontologiakielen tarjoamat lisämääritykset voivat siis vaikuttaa siihen mitä lisäinformaatiota ontologiasta päättelijätoteutus voi päätellä.

OWL-ontologiakielestä on vielä edelleen olemassa vähemmän ilmaisuvoimaiset OWL DL ja OWL Lite versiot [16]. OWL DL on osajoukko täydestä OWL-ontologiakielestä ja OWL Lite on osajoukko OWL DL -kielestä. OWL DL:ssä syntaksia on rajoitettu niin, että päättelijätoteutukset eivät voi jäädä ikuiseen silmukkaan, vaan kaikki johtopäätökset on pääteltävissä äärellisessä ajassa. Esimerkiksi luokka ei voi olla samalla myös jonkin toisen luokan instanssi. OWL Lite:ssä kieltä on rajattu vielä lisää tarkoituksena tehdä työkalutuen tarjoaminen mahdollisimman yksinkertaiseksi.

OWL-ontologiakielen uusin version on OWL 2 [17], joka lisää uusia ominaisuuksia ja parantaa edelleen kielen ilmaisuvoimaa ollen kuitenkin täysin alaspäin yhteensopiva. Eli kaikki validit OWL 1 -ontologiat ovat myös valideja OWL 2 -ontologioita. Uusia ilmaisuvoimaa parantavia ominaisuuksia ovat mm. mahdollisuus määritellä suhdeominaisuus *asymmetriseksi*, *refleksiiviseksi* tai *irrefleksiiviseksi*.

Asymmetrisyys on aikaisemmin esitellyn symmetrisen ominaisuuden vastakohta. Eli suhde ei voi ikinä olla voimassa molempiin suuntiin. Esimerkiksi *hasChild*-suhdeominaisuus voitaisiin määritellä ontologiassa asymmetriseksi. Eli jos on kolmikko <John Smith, hasChild, Anna Lena> tiedetään, että tämä väittämä ei voi ikinä päteä toisinpäin.

Määrittelemällä suhdeominaisuus refleksiiviseksi voidaan sanoa, että jokaiselle instanssille pätee aina tällainen suhde itseensä. Esimerkiksi, jos *hasRelative*-suhdeominaisuus määritellään refleksiiviseksi, voidaan tästä päätellä, että jokaisella on ainakin itsensä sukulaisena.

Irrefleksiivinen suhdeominaisuus on refleksiivisen vastakohta. Se tarkoittaa, että instanssilla ei voi ikinä olla tällaista suhdetta itseensä. Esimerkiksi määritelemällä *parentOf*-suhde irrefleksiiviseksi luotaisiin ontologiaan sääntö, että kenelläkään ei voi olla *parentOf*-suhdetta itseensä.

Luokan ominaisuuksia voidaan OWL 2 -kielessä määritellä myös avainominaisuuksiksi. Tällöin jokainen tämän luokan instanssi on yksikäsitteisesti tunnistettavissa avainominaisuuksiensa arvojen perusteella. Jos siis kahdella instanssilla on samat avainominaisuuksien arvot, voidaan tästä päätellä, että nämä ovatkin oikeasti yksi ja sama instanssi.

3.3 Ontologian kyselykielet

SPARQL (SPARQL Protocol and RDF Query Language) [23] on W3C:n suosittelema kyselykieli RDF-formaatissa olevan datan kyselyihin. Tällöin koko tietokanta nähdään joukkona RDF:n mukaisia subjekti-predikaatti-objekti-kolmikkoja, jotka muodostavat graafeja, joita pitkin SPARQL-kyselyillä navigoidaan.

SPARQL-kyselykielessä on eri tarkoituksiin neljä erilaista kyselymuotoa: SELECT-, CONSTRUCT-, ASK- ja DESCRIBE-kysely [23]. SELECT-kysely palauttaa ratkaisujoukkona kyselyn muuttujiin sidotut arvot sellaisenaan taulukkomuodossa. CONSTRUCT-kysely muuntaa kyselystä saadun ratkaisujoukon kyselyn mukana annetun mallin mukaiseksi RDF-graafiksi ja palauttaa sen. Tulokseksi saadaan siis joukko kolmikoita. ASK-kysely palauttaa vain totuusarvon sen mukaan löytyikö kyselylle yhtään ratkaisua, itse ratkaisuja ei palauteta. DESCRIBE-kysely palauttaa myös RDF-graafin, mutta graafi ei ole suoraan kyselyn ratkaisujoukko, vaan se sisältää kolmikoita, jotka kuvaavat ratkaisuksi saatuja resursseja. Jos esimerkiksi DESCRIBE-kysely kohdistetaan yhteen *Person*-tyyppisen resurssin tunnisteeseen, voisi DESCRIBE-kysely palauttaa kolmikot, joissa kuvataan henkilön nimi, osoite, puhelinnumero ym. hyödyllistä tietoa henkilöstä. Jää kyselyn vastaanottavan ja suorittavan palvelun päätettäväksi, mitä resurssiin liittyviä kolmikoita vastausgraafiin sisällytetään ratkaisuresursseja kuvaamaan. DESCRIBE-kysely on siis vain tapa sanoa palvelulle: kuvaile tätä resurssia.

Esimerkkinä SPARQL-kyselystä ohjelman Ohjelma 13 SELECT-tyyppinen kysely hakisi kuvan 3.2 mukaisesta ontologiasta niiden lentokoneiden tunnisteet, joiden pilottina ”John Smith” -niminen henkilö toimii. Kyselyn tulosjoukko on esitetty taulukossa Taulukko 2.

```
PREFIX example-ontology: <http://www.example.org/example-ontology#>
SELECT ?person ?aircraft
WHERE {
      ?person example-ontology:hasName "John Smith"^^xsd:string .
      ?person example-ontology:pilots ?aircraft .
}
```

Ohjelma 13. *SPARQL-kielinen SELECT-kysely*

Taulukko 2. *SPARQL esimerkkikyselyn tulosjoukko*

person	aircraft
Person_123	Aircraft_54

Kyselyn alussa on määritelty PREFIX-avainsanaa käyttäen lyhenne ontologian IRI-tunnisteelle, joka toimii kyselyssä ontologian nimiavaruutena. SPARQL-kyselykielessä [23] kysymysmerkillä alkavat merkkijonot ovat muuttujia, joihin kyselyn ratkaisut sidotaan. SPARQL-kysely muodostuu siis joukosta subjekti-predikaatti-objekti-väittämiä ja

väittämän minkä tahansa osan voi korvata muuttujalla. Vastaukseksi kyselyyn muodostuu joukko kolmikoita, joille väittämä pitää paikkansa, joista lopulliseksi vastaukseksi palautetaan taulukkomuodossa kyselyn muuttujiin sidotut arvot. Jokainen kyselyn väittämä täytyy päättää pisteeseen, joka merkitsee väittämän loppua, tosin pisteen voi jättää pois kyselyn viimeisen väittämän kohdalla. Ohjelman Ohjelma 13 kysely siis tekee auki kirjoitettuna seuraavat väittämät:

- Resurssilla on *hasName*-ominaisuus, jonka arvo on merkkijono (xsd:string) ”John Smith”
- Resurssilla on *pilots*-suhde, jonka kohteena on mikä tahansa instanssin tunniste.

?person- ja *?aircraft*-muuttujiin sidotaan niiden resurssien tunnisteet, joihin molemmat väittämät pätevät.

SPARQL-kyselykielellä ei pystynyt alun perin luomaan, muokkaamaan tai poistamaan tallennettuja RDF-graafeja, sillä pystyi tekemään vain hakuoperaatioita. W3C:n SPARQL-kyselykielen versio 1.1 määrittelee kielelle laajennoksen SPARQL/Update, joka mahdollistaa luonti- ja poisto-operaatiot RDF-graafien kolmikoille, sekä kokonaisille graafeille [23]. Tietojen päivitys tapahtuu SPARQL/Update-kielessä poistamalla vanha kolmikko ja lisäämällä tämän jälkeen uusi kolmikko eri arvolla. Esimerkkiohjelmassa Ohjelma 14 luodaan uusi *Person*-instanssi ja lisätään instanssille nimi ”Peter Parker”.

```
PREFIX example-ontology: <http://www.example.org/example-ontology#>
INSERT DATA
{
      ?newPerson a example-ontology:Person .
      ?newPerson example-ontology:hasName "Peter Parker"
 }
WHERE
 {
    BIND(IRI("http://www.example.org/example-ontology#Person_456") AS
      ?newPerson)
 }
```

Ohjelma 14. *Tietojen tallennus SPARQL/Update-kielellä*

Ensimmäisen kolmikon predikaattina käytetty ”a” on SPARQL-kielen varattu sana ja vaihtoehtoinen tapa kirjoittaa IRI ”*http://www.w3.org/1999/02/22-rdf-syntax-ns#type*”. Tällä määritetään, että luotu instanssi kuuluu ontologialuokkaan *Person*. *WHERE*-lausekkeessa luodaan uuden instanssin tunniste ja sidotaan se *?newPerson*-muuttujaan.

Esimerkkiohjelmassa Ohjelma 15 päivitetään henkilön ”John Smith” nimi SPARQL-kyselykielen peräkkäisillä DELETE- ja INSERT-operaatioilla.

```
PREFIX example-ontology: <http://www.example.org/example-ontology#>

DELETE { ?person example-ontology:hasName "John Smith" }
INSERT { ?person example-ontology:hasName "Smith John" }
```

```
WHERE
  { ?person example-ontology:hasName "John Smith"
  }
```

Ohjelma 15. *Tietojen päivitys SPARQL/Update-kielellä*

3.4 Ontologioiden tallennus

RDF-formaatissa olevan datan tallennusta varten on rakennettu useita erilaisia kolmikkotietokantoja (triplestore), kuten esimerkiksi Jena-ontologiamoottorin TBD-kolmikkotietokanta [2]. Osassa kolmikkotietokannoista on rakennettu täysin uusi tietokantamoottori alusta alkaen juuri RDF-mallin subjekti-predikaatti-objekti -kolmikoiden tallennusta varten (esimerkiksi Jenan TBD), kun taas osa käyttää alla olevaa perinteistä relaatiotietokantaa kolmikkotietokantana (esimerkiksi Jenan SDB [2]). Myös dokumenttitietokantojen, kuten MongoDB [11], käyttämisestä kolmikkotietokantana on tehty tutkimusta [15] [12].

Ontologioita voidaan tallentaa myös tekstitiedostoina XML-formaatissa. Tällöin ontologia ladataan sovellukseen XML-lähdetiedostoista ajonaikana. XML-tiedostot ovat hyvin joustava tapa tallentaa ontologia, mutta suurien ontologioiden tapauksessa XML-tiedostojen toistuva parsiminen tai koko ontologian muistiin lataaminen ei ole suorituskyvyn kannalta järkevää. Ohjelmassa Ohjelma 16 on esitetty kuvan 3.2 ontologia RDF/XML-formaatissa tallennettuna.

```
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [
    <!ENTITY owl "http://www.w3.org/2002/07/owl#" >
    <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
    <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
    <!ENTITY example-ontology "http://www.example.org/example-ontology#" >
    <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
]>
<rdf:RDF xmlns="http://www.w3.org/2002/07/owl#"
     xml:base="http://www.w3.org/2002/07/owl"
     xmlns:example-ontology="http://www.example.org/example-ontology#"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
     xmlns:owl="http://www.w3.org/2002/07/owl#"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <Ontology rdf:about="http://www.example.org/example-ontology"/>

    <ObjectProperty rdf:about="&example-ontology;pilots">
        <rdfs:range rdf:resource="&example-ontology;Aircraft"/>
        <rdfs:domain rdf:resource="&example-ontology;Person"/>
    </ObjectProperty>
    <DatatypeProperty rdf:about="&example-ontology;hasName">
        <rdfs:domain rdf:resource="&example-ontology;Person"/>
        <rdfs:range rdf:resource="&xsd;string"/>
    </DatatypeProperty>
    <Class rdf:about="&example-ontology;Aircraft"/>
    <Class rdf:about="&example-ontology;Person"/>
    <NamedIndividual rdf:about="&example-ontology;Aircraft_54">
        <rdf:type rdf:resource="&example-ontology;Aircraft"/>
    </NamedIndividual>
    <NamedIndividual rdf:about="&example-ontology;Person_123">
        <rdf:type rdf:resource="&example-ontology;Person"/>
        <example-ontology:hasName rdf:datatype="&xsd;string">John Smith
        </example-ontology:hasName>
        <example-ontology:pilots
        rdf:resource="&example-ontology;Aircraft_54"/>
    </NamedIndividual>
</rdf:RDF>
```

Ohjelma 16. *Kuvan 3.2 ontologia RDF/XML-formaatissa esitettynä.*

Käytettäessä relaatio- tai dokumenttitietokantaa kolmikkotietokantana, muodostuu ongelmaksi tehokkaiden kyselyjen muodostaminen graafipohjaisessa RDF-mallissa olevalle datalle. Relaatio- tai dokumenttitietokannan taivuttaminen RDF-mallin kolmikoiden tallentamiseen ja juuri tätä tarkoitusta varten suunniteltujen SPARQL-kielisten kyselyjen muuntaminen käytetyn tietokannan natiiville kyselykielelle voi olla vaikeaa, kun kyselyiden pitäisi olla tehokkaita.

Ontologioiden tallentamiseen MongoDB-dokumenttitietokantaan on esitetty muutamia eri lähestymistapoja. Tomaszuk ehdottaa tutkimuksessaan dokumenttitietokantojen käytöstä kolmikkotietokantoina [15] jokaisen RDF-formaatin kolmikon esittämistä yhtenä MongoDB:n dokumenttina. Kolmikon subjekti, predikaatti ja objekti kuvautuisivat jokainen yhdeksi MondoDB:n dokumentin avain-arvo pareista ohjelman Ohjelma 17 esimerkin mukaan.

```
{
      subject: "<http://www.example.org/example-ontology#Person_123>",
      predicate: "<http://www.example.org/example-ontology#hasName>",
      object: "John Smith"
}
```

Ohjelma 17. *RDF-formaatin kolmikon esitys MongoDB-tietokannan dokumenttina*

Toinen vaihtoehto hyödyntää MongoDB-tietokannan upotettuja dokumentteja. Tässä vaihtoehdossa jokainen ontologian instanssi esitettäisiin yhtenä MongoDB-tietokannan dokumenttina ja sisältäisi upotettuina dokumentteina kaikki kolmikot, joissa kyseinen instanssi on subjekti [12]. Ohjelmassa Ohjelma 18 on esimerkki tällaisesta dokumentista. Ohjelman dokumentti kuvaa *Person_123*-instanssia. Instanssilla on tietotyyppiominaisuus *hasName* ja suhdeominaisuus *pilots*. Molemman tyyppiset ominaisuudet on sisällytetty samaan *predicates*-listaan ja ominaisuuden tyyppi on määritelty *type*-kentässä. Dokumentissa voisi toki olla *predicates*-lista jaettuna kahdeksi eri kentäksi, joista toinen sisältäisi tietotyyppiominaisuudet ja toinen suhdeominaisuudet.

```
{
  subject: "<http://www.example.org/example-ontology#Person_123>",
  predicates: [
    {
      uri: "<http://www.example.org/example-ontology#hasName>",
      objects: [
        {
          type: "literal",
          value: "John Smith"
        }
      ]
    },
    {
      uri: "<http://www.example.org/example-ontology#pilots>",
      objects: [
        {
          type: "uri",
          value: "<http://www.example.org/example-ontology#Aircraft_54>"
        }
      ]
    }
  ]
}
```

Ohjelma 18. *Samaan subjekti-instanssiin liittyvät RDF-formaatin kolmikot eitettynä yhtenä MongoDB-tietokannan dokumenttina. Mukailtu lähteestä [12].*

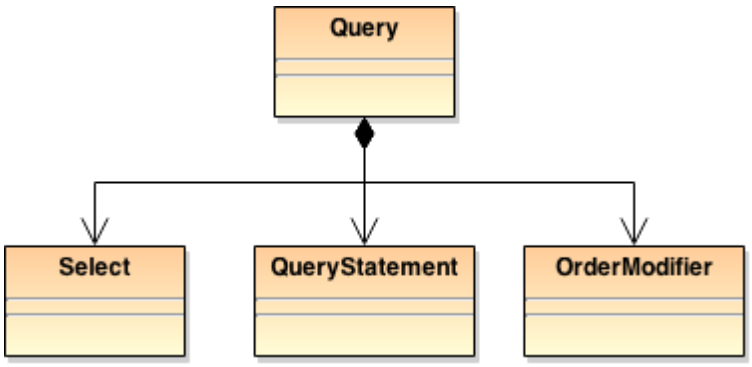
4. ARKKITEHTUURI

4.1 Ontologiapalvelun kyselykieli

Ontologiakyselyissä käytetään järjestelmän sisäistä SPARQL-tyylistä kyselykieltä. Kielen rakenne ja toiminta on muuten samanlaista kuin SPARQL-kyselykieli, mutta sisältää seuraavat poikkeavuudet:

- Kyselyssä väittämän predikaatti ei voi olla muuttuja. Vain väittämän subjekti ja objekti voivat olla muuttujia.
- Kyselyt palauttavat aina ontologialuokkien instansseja.
- Kyselykielellä voi tehdä vain SPARQL:n SELECT-tyyppisiä kyselyitä.
- Kyselyn Select-osuus voi sisältää muuttujien lisäksi haettavan ontologialuokan instanssin URI-tunnisteen. Tällöin haettu instanssi sisältää kaikki sen ominaisuudet.
- Kaikkien kyselyssä käytettävien muuttujien ja URI-tunnisteiden tyyppi (ontologialuokka) tulee olla määritelty kyselyn väittämissä.

Tuloksena kyselyihin hakupalvelu palauttaa Select-osuudessa määritettyihin muuttujiin ja URI-tunnisteisiin sidottuja ontologialuokkien instansseja. Muuten kyselyt toimivat samalla tavalla kuin SPARQL-kyselyt. Kuvassa 4.1 on esitetty ontologiakyselyn rakenne.



Kuva 4.1: Ontologiakyselyn rakenne.

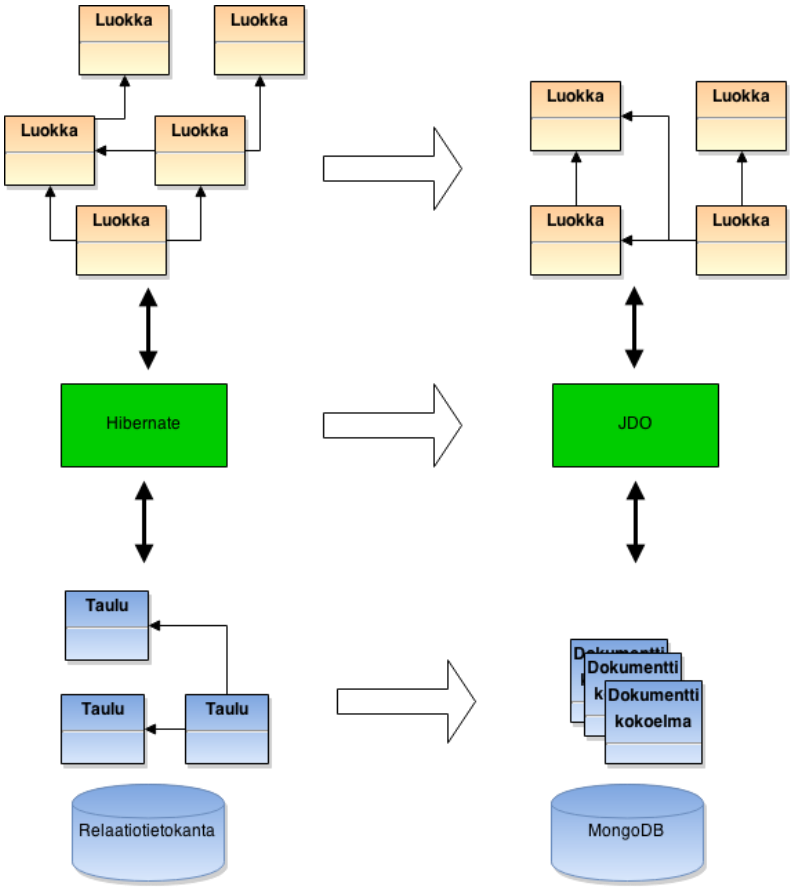
Kysely muodostuu SPARQL-kielen SELECT-kyselyn (katso aliluku 3.3) tapaan *Select*-osuudesta ja *QueryStatement*-väittämistä. *QueryStatement*-väittämät ovat kuten SPARQL-kielessäkin subjektista, predikaatista ja objektista muodostuvia kolmikoita, jotka rajaavat tulosjoukkoa. *Select* on muuttuja tai instanssin URI, johon kyselyn vastaus sidotaan. *OrderModifier*-ehto määrittää kuinka tulosjoukko järjestetään ja tulosjoukon rajauksen, esimerkiksi kuinka monta tulosta maksimissaan halutaan.

4.2 Ontologiapalvelun siirron kuvaus

Ontologian tiedonhakupalvelu tarjoaa ontologian luokkien yksilöiden (instanssien) haut. Palvelu ottaa vastaan aliluvussa 4.1 kuvatulla SPARQL-tyylisellä kyselykielellä tehdyn kyselyn, jolla haetaan ontologian instansseja. Esimerkiksi kysely voisi olla hakea kaikki *Aircraft*-ontologialuokan instanssit. Nykyisessä järjestelmässä palvelu on toteutettu käyttäen tietovarastona relaatiotietokantaa ja hyödyntäen sovitukseen relaatiotietokannan tietomallin ja sovelluksen tietomallin välillä Hibernate-kirjastoa [7].

Tässä työssä ontologian tiedonhakupalvelu siirrettiin tietokantateknologian päälle, jossa tietovarastona käytetään MongoDB-dokumenttitietokantaa JDO-ohjelmointirajapinnan kautta. Tarkoituksena siirrossa on saada ontologiapalvelu käytettäväksi osana järjestelmää, jossa tiedon varastointiin käytetään näitä teknologioita. Myös ontologiaskeeman ajonaikainen muokkaaminen pyrittiin tekemään mahdolliseksi MongoDB-tietokannan päälle siirretyssä toteutuksessa, mikä ei ollut relaatiotietokantatoteutuksessa mahdollista. Kolmannen osapuolen ontologiamoottorin, kuten Jena [1], käyttöönottoakin harkittiin. Tämän arvioitiin kuitenkin olevan huomattavasti työläämpää, kuin siirtää osa olemassa olevasta ontologiamoottorin toteutuksesta käyttämään valittua tietokantateknologiaa.

Kuvassa 4.2 on esitetty konseptuaalinen kuvaus ontologian tiedonhakupalvelun relaatiotietokantatoteutuksen siirtämisestä käyttämään MongoDB-tietokantaa JDO-ohjelmointirajapintaa hyödyntäen. Kuvassa vasemmalla on esitetty nykyisen toteutuksen rakenne ja oikealla uuden toteutuksen rakenne.



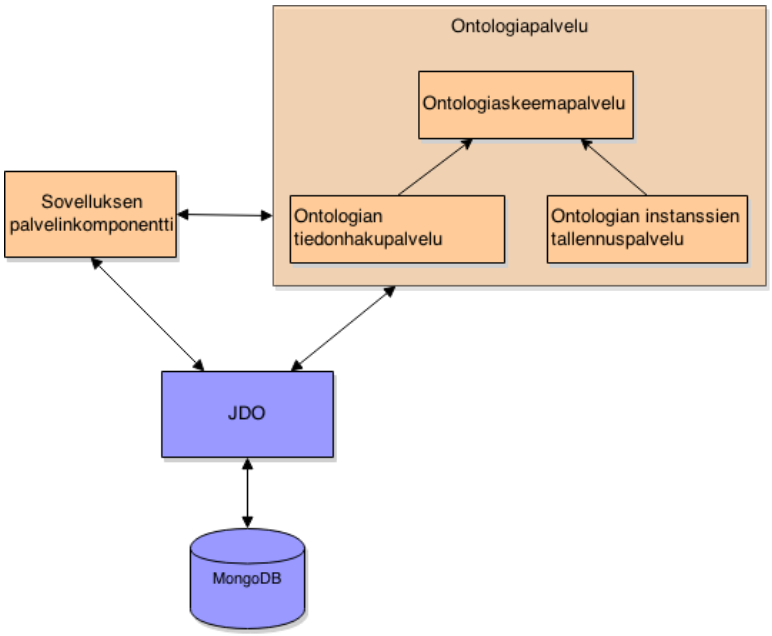
Kuva 4.2: Konseptuaalinen kuvaus ontologian tiedonhakupalvelun relaatiotietokantatoteutuksen siirtämisestä käyttämään JDO:ta ja MongoDB-dokumenttitietokantaa.

Relaatiotietokantojen ja MongoDB-tietokannan eroista johtuen, JDO-rajapinnan käytöstä aiheutuvien rajoitteiden takia, sekä ontologiaskeeman ajonaikaisen muokkauksen mahdollistamiseksi ei relaatiotietokantatoteutuksen taulukkoskeemaa pystynyt siirtämään suoraviivaisesti MongoDB-dokumenttikokoelmiksi. Ontologian tiedonhakupalvelun tietomalli jouduttiin siis suunnittelemaan pitkälti alusta alkaen uudestaan. Koska MongoDB-tietokannan tietomalli muodostuu JDO-kuvauksen kautta Java-koodissa käytettävästä luokkarakenteesta, eroaa myös siirretyn toteutuksen luokkarakenne vanhasta. Tämä tietomalli ja sovelluksen luokkarakenne on kuvattu myöhemmin tässä luvussa.

Ontologian tiedonhakupalvelun relaatiotietokantatoteutuksessa olio-relaatiosovituksessa käytettiin Hibernate-kirjastoa ja hakupalvelu muutti SPARQL-tyyliset kyselyt SQL-kyselyiksi, jotka ajettiin suoraan relaatiotietokannassa. Uudessa toteutuksessa tämän MongoDB-tietomallin ja hakupalvelun tietomallin välisen sovituksen hoitaa JDO. Tästä johtuen MongoDB-toteutuksessa SPARQL-tyyliset kyselyt muutetaan ensin JDOQL-kyselyiksi. JDO-toteutus muuttaa sitten JDOQL-kyselyt abstrahoimansa tietokannan kyselykielelle, eli tässä tapauksessa MongoDB-kyselykielelle.

4.3 Ontologiapalvelun liittyminen osaksi muuta järjestelmää

Kuvassa 4.3 on esitetty kokonaiskuva ontologiapalvelusta ja sen liittymisestä osaksi kokonaissovellusta. Ontologiapalvelu sisältää seuraavat toisistaan erilliset palvelut: ontologiaskeema-, ontologian tiedonhaku- ja ontologian instanssien tallennuspalvelu. Ontologiapalvelu ja sovelluksen palvelinkomponentit käyttävät tietokannan käsittelyyn JDO-ohjelmointirajapintaa. Tässä työssä toteutettu ontologian tiedonhakupalvelu käyttää JDO:n tarjoamaa JDOQL-kyselykieltä ontologiakyselyiden suorittamiseen ja JDO:n tarjoamia annotaatioita tietokannan tietomallin määrittelyyn. Kuvaan sinisellä merkityt komponentit ovat toteutuksessa käytettyjä kolmansien osapuolien komponentteja.



Kuva 4.3: Kokonaiskuva ontologiapalvelusta ja sen liittymisestä osaksi kokonaissovellusta.

Ontologiaskeemapalvelu tarjoaa ontologian rakenteen, skeeman, tarkastelun ja muokkaamisen. Ontologiaskeemalla tarkoitetaan ontologian luokkien, luokkien sallittujen tietotyyppiominaisuuksien ja luokkien välillä sallittujen suhdeominaisuuksien kuvausta. Ontologiaskeema on tallennettu MongoDB-tietokantaan ontologian instansseista erilliseen tietorakenteeseen. Ontologiaskeemapalvelua käyttäen on mahdollista muokata ontologian käsitteitä, käsitteiden ominaisuuksia ja käsitteiden välisiä suhteita ajoaikana. Esimerkiksi voitaisiin lisätä ontologiaan täysin uusi käsite.

Ontologian tiedonhakupalvelu tarjoaa ontologialuokkien instanssien haut SPARQL-tyylisellä kyselykielellä. Tiedonhakupalvelu käyttää ontologiaskeemapalvelua hauissa mm. ontologiakyselyjen oikeellisuuden tarkistamiseen. Ontologian instanssien tallennuspalvelu tarjoaa ontologialuokkien instanssien tallennuksen ja muokkauksen. Instanssien tallennuspalvelu samoin tarkistaa tallennettavien ontologialuokkien instanssien ja instanssien välisten suhteiden oikeellisuuden skeemapalvelua käyttäen. Tässä luvussa

käsitellään vain työssä toteutetun ontologian tiedonhakupalvelun arkkitehtuuria. Ontologiaskeemapalvelua ja ontologian instanssien tallennuspalvelua ei käsitellä tässä työssä.

4.4 MongoDB-dokumenttitietokannan tietomalli

Tässä luvussa esitetään ontologian tiedonhakupalvelun tietokannan tietomalli. Kyseessä on ontologialuokkien instanssien tallennuksessa ja hauissa käytettävä tietomalli. Ontologiaskeema on tallennettu MongoDB-tietokantaan ontologialuokkien instansseista erillään omaa tietomalliaan käyttäen.

JDO:ta käytettäessä tietokannan tietomalli muodostuu JDO-persistoituviksi merkittyjen luokkien luokkarakenteesta ja niiden metadatamäärityksistä [1]. Tästä johtuen tietokannan tietomallin suunnittelu tapahtuu suunnittelemalla sovelluksessa käytettävien JDO-persistoituvien luokkien luokkarakenne. Vaikka JDO abstrahoi alla olevan tietokantatoteutuksen, oli silti tietokantamallia suunnitellessa tiedossa, että tietokantana tullaan käyttämään MongoDB-tietokantaa.

Koska käytetyn DataNucleus JDO-toteutuksen version tuki MongoDB-tietokannalle on vielä osittain puutteellista, täytyi tietomallin suunnitteluratkaisuista osa tehdä JDO-MongoDB-yhteensopivuutta ajatellen, jotta palvelusta saataisiin suorituskykyinen. Tästä huolimatta tietomalliratkaisu toimisi myös vaikka alla olevaksi tietokantatuotteeksi vaihdettaisiin esimerkiksi jokin relaatiotietokanta, tosin suorituskyky ei olisi optimoitu. Tietomallin suunnittelussa otettiin myös huomioon vaatimus, että ontologiaskeeman rakennetta täytyy pystyä muuttamaan ajonaikaisesti. Ontologiaan voidaan esimerkiksi lisätä uusia ontologialuokkia ja ontologialuokkien välisiä suhdeominaisuuksia ajonaikana. Seuraavaksi esitellään kokeillut vaihtoehdot ja millaiseen tietomalliin lopulta päädyttiin.

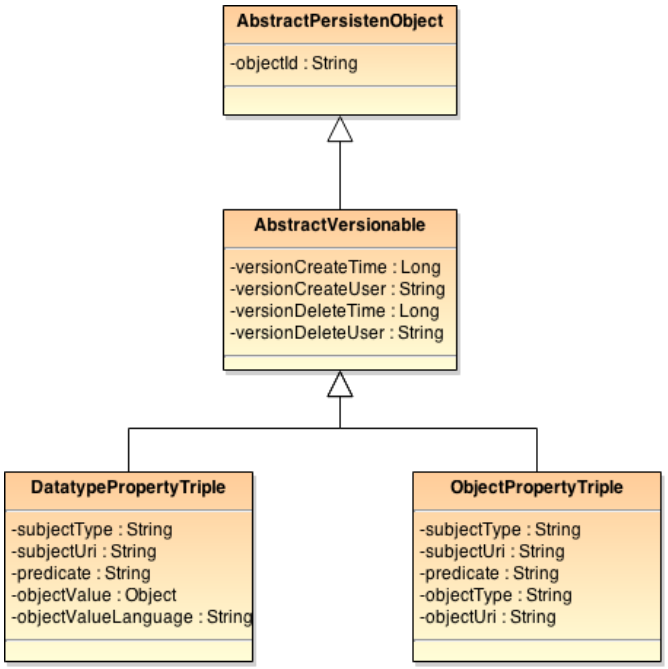
Ensimmäinen suunniteltu vaihtoehto tietokannan tietomalliksi oli kuvata jokainen ontologian käsite omana JDO-persistoituvana luokka. Tällöin jokaiselle ontologian käsitteelle muodostuisi MongoDB-tietokannassa erillinen kokoelma. Ongelmaksi muodostuu ratkaisun staattisuus. JDO-persistoituvat luokat täytyy määritellä annotaatioilla ohjelmointivaiheessa, jotta JDO-enhancer-prosessi (katso kohta 2.3.1) pystyisi lähdekoodia käännettäessä käsittelemään luokat ja mahdollistamaan luokkien persistoinnin. Tästä johtuen ontologiaskeema pitäisi siis olla määritelty kokonaisuudessaan jo ohjelmointivaiheessa. Ontologiaskeeman muuttaminen ei olisi mahdollista ajonaikana. Esimerkiksi ontologiaskeeman muuttaminen lisäämällä uusi käsite tarkoittaisi, että lähdekoodia jouduttaisiin muokkaamaan lisäämällä uutta käsitettä kuvaava JDO-persistoituvat luokka. Ratkaisu jouduttiin siis hylkäämään.

Toinen lähestymistapa oli käyttää yhtä geneeristä JDO-perisistoituvaa luokkaa kuvamaan ontologian käsitteiden instansseja. MongoDB-tietokannassa dokumentit olisivat

tällöin rakenteeltaan kuten aliluvussa 3.4 esitellyssä Nitzschken mallissa (Ohjelma 18). Dokumentti sisältäisi siis instanssin URI-tunnisteen ja instanssin tietotyyppi- ja suhdeominaisuuksien sisältyisivät samaan dokumenttiin listarakenteena. Rakenne mahdollistaisi myös ontologiaskeeman ajonaikaisen muokkaamisen, koska mitään tietorakenteessa ei ole sidottu staattisesti ontologiaskeemaan. Tämä ratkaisuvaihtoehto jouduttiin kuitenkin hylkäämään JDO:n MongoDB-tuen puutteiden takia. MongoDB tukee List-tietorakenteen käyttöä dokumenttien kenttänä ja nopeita hakuja tällaisesta rakenteesta. DataNucleus JDO-toteutuksesta kuitenkin puuttui toteutus, joka olisi suorittanut List-tietorakenteen JDO-metodit, kuten contains(value), osana MongoDB-kyselyä. Sen sijaan testeissä huomattiin, että tällaiset kyselyt suoritettiin muistinvaraisesti lataamalla kaikki kyselyn kohdekokoelman dokumentit muistiin ja suorittamalla rajaus vasta sitten.

Lopullinen tietokannan tietomallin muodostava luokkarakenne on esitetty kuvassa 4.4. Tietomalliratkaisu muistuttaa aliluvussa 3.4 esitettyä Tomaszukin mallia (Ohjelma 17) tallentaa RDF-graafi dokumenttitietokantaan, mutta sitä on laajennettu paremman suorituskyvyn mahdollistamiseksi, kun MongoDB-tietokantaa käytetään JDO-rajapinnan kautta. Kolmikot on jaettu kahteen eri JDO-persistoituvaan luokkaan *DatatypePropertyTriple* ja *ObjectPropertyTriple* sen perusteella onko kyseessä instanssin tietotyyppiominaisuus vai suhdeominaisuus, joka on myös OWL-ontologiakielen tapa ryhmitellä ontologialuokkien ominaisuuksia (katso kohta 3.2). Tämä jakaa kuormaa hakuja suoritettaessa. Tätä tietomallia käytettäessä JDO-toteutus pystyy suorittamaan kaikki tarvittavat haut MongoDB-tietokannassa, eikä muistinvaraista suoritusta käytetä. Lisäksi ontologiskeeman muokkaaminen ajonaikana on mahdollista.

Molemmat ominaisuutta kuvaavat luokat periytyvät *AbstractVerionable*-luokasta, joka sisältää tietokantaan tallennettavan kolmikon versiotiedot mahdollistaen historiatiedon säilytyksen tarvittaessa. Kantaluokka *AbstractPersistentObject* sisältää kolmikon yksilöivän surrogaattitunnisteen.



Kuva 4.4: Ontologian tiedonhakupalvelun JDO-persistoituvien luokkien lopullinen luokkarakenne

Molemmissa kolmikoissa subjektiosa on jaettu kahtia subjektin tyyppiin (*subjectType*-kenttä) ja subjektin tunnisteeseen (*subjectUri*-kenttä). Tämä on tehty seuraavasta syystä. Järjestelmässä, jonka osa ontologian tiedonhakupalvelu tulee olemaan, haut muodostuvat aina haettavien instanssien tyypistä (ontologialuokka) ja kyseisen tyypin instanssien tarkemmista hakurajoituksista. Tästä johtuen tarvittaisiin aina kaksi kyselyä: ensin pitäisi hakea kaikki tietyntyyppiset instanssit, jonka jälkeen tätä joukkoa rajoitettaisiin muilla rajoitteilla. Lisäämällä tyyppitieto osaksi jokaisen kolmikon subjektia, pystytään jokaisessa kyselyssä rajoittamaan muiden rajoitusten yhteydessä myös tulosinstanssien tyypillä, mikä vähentää huomattavasti sellaisen tiedon käsittelyä, mitä ei lopullisessa vastauksessa palauteta. Samasta syystä myös *ObjectPropertyTriple*-luokassa kolmikon objektiosa on jaettu samalla tavalla kahteen osaan.

Tietotyyppiominaisuuden arvo voi olla tyypiltään mikä tahansa, esimerkiksi kokonaisluku, merkkijono, boolean tai päivämäärä jne. Tästä johtuen *DatatypePropertyTriple*-luokan *objectValue*-kentän tyyppinä on *Object*. Tämä ei kuitenkaan aiheuta ongelmia tyyppien suhteen. DataNucleus JDO ja MongoDB osaavat tehdä tyyppimuunnoksen oikein ja tallentavat kentän arvon tietokantaan oikean tyyppisenä, esimerkiksi päivämääränä. Myös kun olio noudetaan tietokannasta, säilyy tyyppi oikeana. Lisäksi vertailuoperaattorit JDOQL-kyselyssä toimivat kentälle oikein ottaen todellisen tyypin huomioon. Esimerkiksi jos kenttään on tallennettu päivämäärä, vertaillaan kenttien arvoja kuin päivämääriä JDOQL-kyselystä muodostetussa MongoDB-kyselyssä.

DatatypePropertyTriple-luokkaan on lisätty vielä *objectValueLanguage*-kenttä. Sen tarkoituksena on mahdollistaa ontologialuokkien instanssien tietotyyppiominaisuuksien arvojen lokalisointi eri kielille.

Tämä kuvassa 4.4 esitetty luokkarakenne kuvautuu JDO-kerroksen kautta ohjelmassa Ohjelma 19 esitetyn kaltaisiksi kahdeksi dokumenttikokoelmaksi MongoDB-tietokannassa.

DatatypePropertyTriple collection:

```
{
      _id: ObjectId("90uad9adajdiojf9sdf")
      subjectType: "<http://www.example.org/example-ontology#Person>",
      subjectUri: "<http://www.example.org/example-ontology#Person_123>",
      predicate: "<http://www.example.org/example-ontology#hasName>",
      objectValue: "John Smith",
      objectValueLanguage: "fi",
      versionCreateTime: NumberLong("142356336094"),
      versionCreateUser: "user1",
      versionDeleteTime: null,
      versionDeleteUser: null
}
```

ObjectPropertyTriple collection:

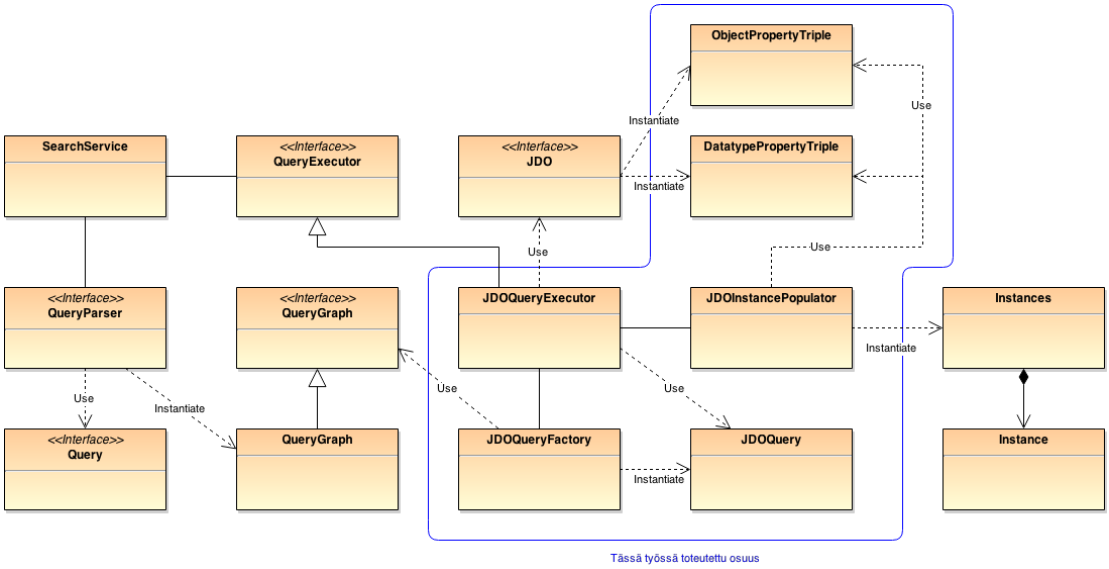
```
{
      _id: ObjectId("9usdf9sduf0s9du85435")
      subjectType: "<http://www.example.org/example-ontology#Person>",
      subjectUri: "<http://www.example.org/example-ontology#Person_123>",
      predicate: "<http://www.example.org/example-ontology#pilots>",
      objectType: "<http://www.example.org/example-ontology#Aircraft>",
      objectUri: "<http://www.example.org/example-ontology#Aircraft_54>",
      versionCreateTime: NumberLong("142356336094"),
      versionCreateUser: "user1",
      versionDeleteTime: null,
      versionDeleteUser: null
}
```

Ohjelma 19. *Ontologian tiedonhakupalvelun MongoDB-dokumenttien rakenne.*

4.5 Ohjelmistorakenne

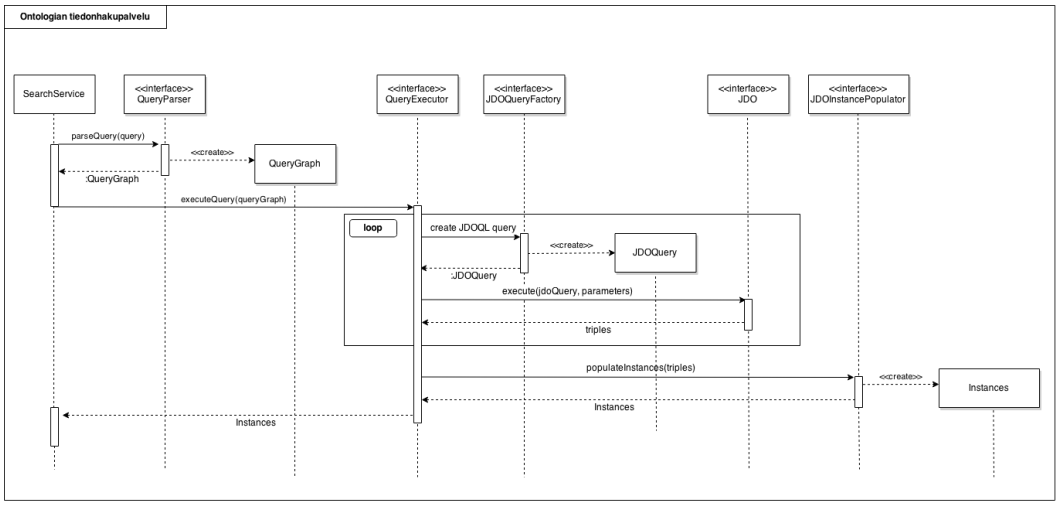
Ontologian tiedonhakupalvelun rakenne on esitetty kokonaisuudessaan kuvassa 4.5. Hakupalvelun relaatiotietokantatoteutuksesta jouduttiin korvaamaan kyselyn muodostava ja suorittava osuus. Myös relaatiotietokantatoteutuksen tietomallissa relaatiotietokannan tauluiksi Hibernate [7] olio-relaatio sovituksella (ORM) kuvautuneet luokat korvattiin uuden tietomallin luokilla. Muut luokat ja rajapinnat pystyttiin ottamaan käyttöön sellaisenaan tai hyvin pienillä muutoksilla aikaisemmasta hakupalvelun relaatiotietokantatoteutuksesta. Kuvassa on rajattu sinisellä uuden JDO-toteutuksen muodostava osuus, johon kuuluvat seuraavat luokat:

- *JDOQueryExecutor*
- *JDOQueryFactory*
- *JDOQuery*
- *JDOInstancePopulator*
- *DatatypePropertyTriple*
- *ObjectPropertyTriple*



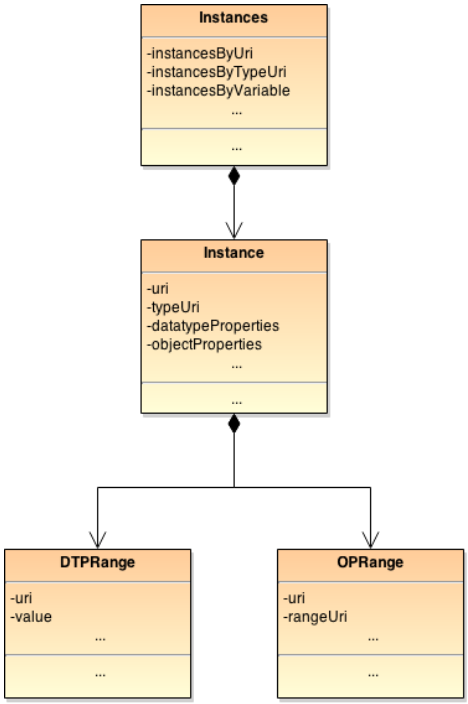
Kuva 4.5: Ontologian tiedonhakupalvelun rakenne.

Seuraavaksi on selostettu hakupalvelun rakenteen toiminta pääpiirteissään. Toiminta on myös esitetty sekvenssikaaviona kuvassa 4.6. Palvelun käyttäjä luo kyselyä kuvaavan *Query*-olion ja antaa sen *SearchService*-rajapinnalle suoritettavaksi. *SearchService* käyttää *QueryParser*-rajapintaa saamansa *Query*-olion muuntamiseen kyselygraafiksi (*QueryGraph*-luokka*). Query*-olio kuvaa SPARQL-tyylistä kyselyä. Se muodostuu kuten SPARQL-kysely kyselyä rajoittavista, RDF-kolmikoina (subjekti-predikaatti-objekti) esitettävistä, lausumista ja muuttujista, joihin ratkaisut sidotaan. *SearchService* sitten antaa luodun SPARQL-kyselygraafin *QueryExecutor*-rajapinnan JDO-toteutukselle. *JDOQueryExecutor* käyttää *JDOQueryFactory*-oliota JDOQL-kyselyjen muodostamiseen SPARQL-kyselygraafista. Muodostetut JDOQL-kyselyt annetaan JDO-rajapinnalle suoritettavaksi, joka palauttaa kyselyn tulokset *DatatypePropertyTriple*- ja *ObjectPropertyTriple*-olioina.



Kuva 4.6: Ontologian tiedonhakupalvelun toiminta.

Ontologiapalvelun ulkopuolelle ontologialuokkien instanssit eivät näy kuvassa 4.4 esitettynä kolmikkorakenteena, vaan näistä kolmikoista *JDOInstancePopulator* muodostaa hakupalvelun ulkopuolella käytettäväksi tarkoitettuja, kokonaisia, ontologialuokan instansseja kuvaavia *Instance*-luokan olioita. Nämä *Instance*-oliot lisätään säiliönä toimivaan *Instances*-luokan olioon, joka palautetaan lopullisena vastauksena hakupalvelun käyttäjälle. Tämä tietorakenne on esitetty kuvassa 4.7.



Kuva 4.7: Hakupalvelun hakujen vastauksena palauttama tietorakenne.

Instance-olio muodostuu sen tietotyyppiominaisuuksista (*DTPRange*-luokka) ja suhdeominaisuuksista (*OPRange*-luokka). Attribuutti *uri* on *Instance*-olion yksilöivä tunniste ja *typeUri*-attribuutti on instanssin ontologialuokan tunniste, eli kertoo minkä ontologi-

an käsitteen ilmentymä *Instance*-olio on. *DTPRange*- ja *OPRange*-luokassa *uri*-attribuutti on ominaisuuden tyypin tunniste. *DTPrange*-luokan *value*-attribuutti on instanssin kyseisen tietotyyppiominaisuuden literaali arvo. *OPRange*-luokassa *rangeUri*-attribuutti on instanssin kyseisen suhdeominaisuuden kohdeinstanssin URI-tunniste. *Instances*-säiliössä *Instance*-oliot on järjestetty *Map*-tietorakenteisiin mm. ontologialuokan mukaan (*instancesByTypeUri*-attribuutti), instanssin URI:n mukaan (*instancesByUri*-attribuutti), sekä sidottu hakupalvelun SPARQL-kyselyssä käytettyihin muuttujiin (*instancesByVariable*-attribuutti). Instansseja voidaan noutaa säiliöstä myös ominaisuuksien arvojen perusteella. Hakupalvelu palauttaa kyselyn tulokset palvelun käyttäjälle tässä tietorakenteessa, jossa niitä käsitellään mm. sovelluksen käyttöliittymässä.

5. TOTEUTUS

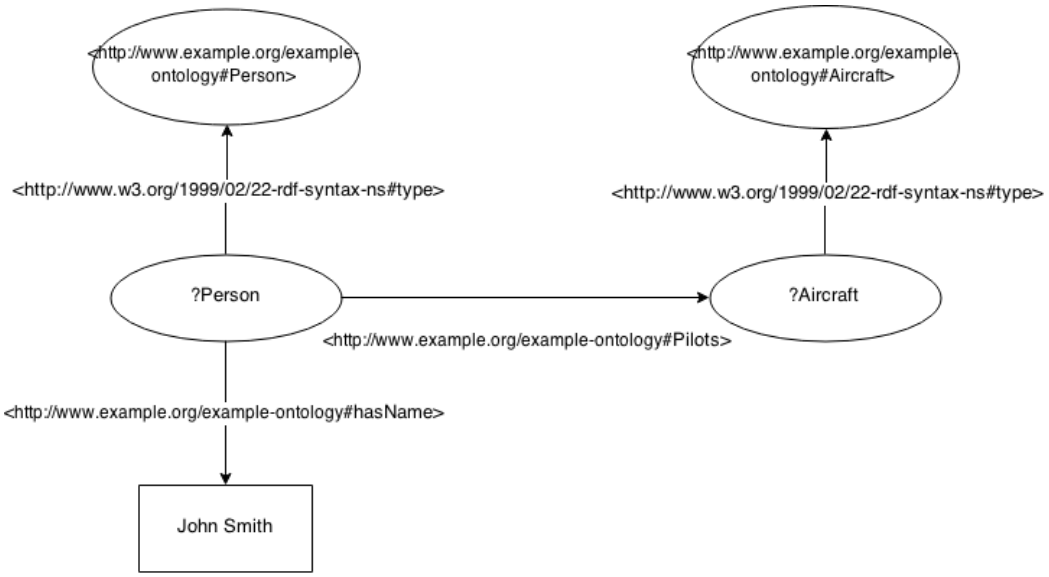
5.1 Kyselyn suoritus

Kyselyn suorittava *JDOQueryExecutor*-luokka saa SPARQL-kyselygraafin suoritettavakseen. Ensimmäinen vaihe kyselyn suorituksessa on muuntaa tämä kyselygraafi JDOQL-kyselykieliseksi kyselyksi. Ohjelmassa Ohjelma 20 on esitetty esimerkki kuvan 3.2 ontologiasta tietoa hakevasta SPARQL-kyselystä, jollaisen käyttäjä voisi antaa ontologian hakupalvelulle suoritettavaksi. Kyselyssä haetaan lentokoneet, joiden pilottina ”John Smith” -niminen henkilö toimii. Kyselyssä on myös määritetty *rdf:type*-predikaatin kolmikoilla muuttujien tyypit, eli minkä ontologialuokan instansseja haetaan.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX example-ontology: <http://www.example.org/example-ontology#>
SELECT ?person ?aircraft
WHERE {
      ?person rdf:type example-ontology:Person .
      ?aircraft rdf:type example-ontology:Aircraft .
      ?person example-ontology:hasName "John Smith"^^xsd:string .
      ?person example-ontology:pilots ?aircraft .
}
```

Ohjelma 20. *SPARQL-kysely, jossa on määritetty muuttujien tyypit*

Kuten kohdassa 4.5 kuvattiin, tämä SPARQL-kysely parsitaan JDOQL-muunnosta varten kyselygraafiksi ja annetaan *JDOQueryExecutor*-luokalle suoritettavaksi. Tämä SPARQL-kyselystä parsittu kyselygraafi on esitetty kuvassa 5.1.



Kuva 5.1: Ohjelman Ohjelma 20 SPARQL-kysely graafina esitettynä.

SPARQL-kielellä kyseessä on yksi kokonainen kysely. Kysely pyrittiin ensin esittämään myös JDOQL-kyselykielellä yhtenä yhtenäisenä kyselyrakenteena käyttäen sisäkkäisiä kyselyitä, jossa ulompi kysely käyttäisi aina alikyselyn tulosta osana omaa kyselyään. JDO-määrittelyssä [1] alikyselyjen tulosten käyttäminen osana pääkyselyä on vielä kuitenkin hyvin rajoittunutta. Esimerkiksi, jos alikyselyssä ei käytetä jotain aggregaattifunktiota, jolloin alikysely palauttaisi listana yhden tai useampia vastauksia (katso kohta 2.3.2), ei tähän alikyselyn tulokseen pystytä käyttämään JDO:n *contains*-metodia pääkyselyssä.

Vaikka JDO:ssa ei olisi näitä rajoitteita alikyselyiden suhteen, joutuisi JDO-toteutus kuitenkin pilkkomaan sisäkkäisistä kyselyistä muodostuvan JDOQL-kyselykokonaisuuden useaksi erilliseksi MongoDB-kyselyksi. Syynä tälle on MongoDB-kyselykielen periaatteellinen erilaisuus JDOQL-kyselykielestä. Kuten kohdassa 2.4 mainittiin, eivät sisäkkäiset kyselyt ole MongoDB-kyselykielessä rakenteena mahdollisia, vaan tarvitaan useita peräkkäisiä kyselyitä. Käytetyn DataNucleus JDO-toteutuksen MongoDB-tuki oli myös vielä tältä osin puutteellista. Testeissä huomattiin, että JDO-toteutus ei suorita tällaista sisäkkäistä JDOQL-kyselyrakennetta tehokkaasti MongoDB-kyselynä, vaan kyselyn suoritus tapahtuu täysin muistinvaraisesti.

Näistä rajoituksista johtuen ei sisäkkäisiä JDOQL-kyselyitä käytetä, vaan SPARQL-kyselygraafi pilkotaan useiksi yksittäisiksi JDOQL-kyselyiksi. Kyselygraafin parsiminen ja JDOQL-kyselyiden muodostaminen tapahtuu ohjelmassa Ohjelma 21 esitetyn algoritmin mukaan.

```
Input: SPARQL query as a graph
Output: result instance URIs bound to query variables

results ← result instance URIs bound to query variables

FOREACH query variable v
  FOREACH datatype property dtp of v
    JDOQLQuery q = createJDOQLQuery(dtp, dtp object value, results.get(v))
    executeQuery(q)
    bind result URIs to variable v in results

  FOREACH object property op of v WHERE op object value is URI
    JDOQLQuery q = createJDOQLQuery(op, op object value, results.get(v))
    executeQuery(q)
    bind result URIs to variable v in results

  FOREACH object property op WHERE op object value is v
  AND op subject value is URI
    JDOQLQuery q = createJDOQLQuery(op, op subject value, results.get(v))
    executeQuery(q)
    bind result URIs to variable v in results

FOREACH object property op WHERE op object value v1 is variable
AND op subject value v2 is variable
  JDOQLQuery q = createJDOQLQuery(op, results.get(v1), results.get(v2))
  executeQuery(q)
  bind result URIs to variables v1 and v2 in results

RETURN results
```

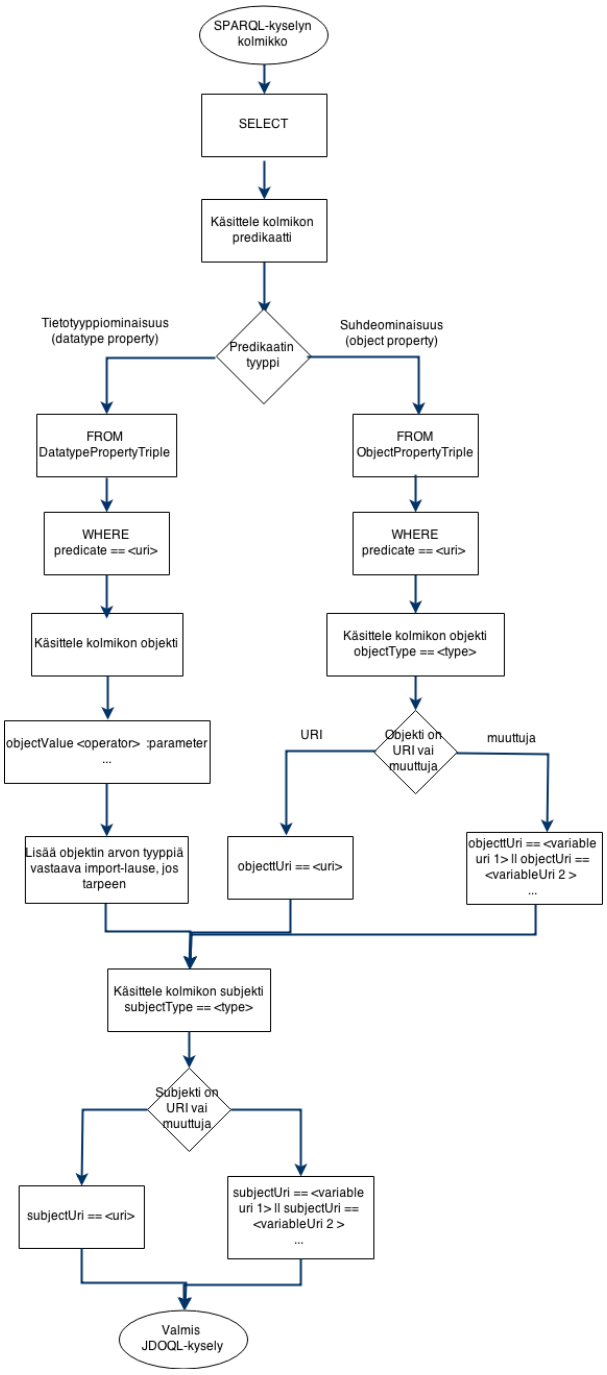
Ohjelma 21. *JDOQL-kyselyjen muodostaminen SPARQL-kyselygraafista.*

Ensin siis graafista haetaan kolmikot, jotka rajoittavat käsiteltävään muuttujaan sidottavaa tulosjoukkoa tietotyyppiominaisuuksien (datatype property) perusteella. Näistä kolmikoista luodaan erilliset JDOQL-kyselyt ja suoritetaan. Tulokseksi saadaan muuttujaan sidottu URI-joukko. Edellisen kyselyn tuloksena saatua URI-joukkoa käytetään aina seuraavassa kyselyssä rajoitteena korvaamalla kyselyssä kolmikon muuttuja tällä URI-joukolla. Tulosjoukko siis supistuu kokoajan jokaisella kyselyllä. Seuraavaksi graafista haetaan kolmikot, jotka rajoittavat käsiteltävään muuttujaan sidottavaa tulosjoukkoa sellaisten suhdeominaisuuksien (object property) perusteella, joissa suhdeominaisuuden kohde (object) on yksittäinen URI ja lähde (subject) käsiteltävä muuttuja. Tämän jälkeen käydään läpi käsiteltävään muuttujaan kohdistuvat suhdeominaisuudet, joissa lähde on yksittäinen URI ja objekti käsiteltävä muuttuja. Näistä kolmikoista luodaan samaan tapaan erilliset JDOQL-kyselyt käyttäen edellisten kyselyjen tuloksia rajoitteena. Viimeiseksi luodaan JDOQL-kyselyt graafin niistä suhdeominaisuuskolmikoista, joissa sekä suhteen lähde että kohde ovat muuttujia.

Graafin kolmikot käydään läpi järjestyksessä, jotta tulosjoukkoa todennäköisesti eniten rajaavista kolmikoista luodaan kyselyt ensin, jolloin tietokannasta haettaisiin ja käsiteltäisiin mahdollisimman vähän sellaista dataa, mitä ei lopullisessa vastauksessa palauteta. Kaikissa kyselyissä käytetään muuttujan tyyppiä (ontologialuokkaa) tulosta rajaava-

na tekijänä, joka karsii pois jo suurimman osan niistä tuloksista, joista ei kyselyssä olla kiinnostuneita. Kun kyselygraafi on tällä tavalla iteroitu läpi, on lopulliseksi tulokseksi saatu kyselyn muuttujiin sidotut instanssien URI-joukot.

Jokaisesta SPARQL-kyselygraafin tietotyyppiominaisuuskolmikosta ja suhdeominaisuuskolmikosta siis muodostuu yksi JDOQL-kysely. JDOQL-kysely muodostetaan SPARQL-graafin kolmikosta kuvassa 5.2 esitetyn vuokaavion mukaisesti, kun tietokannan tietomalli on kuvassa 4.4 esitetty tietomalli.



Kuva 5.2: JDOQL-kyselyn muodostaminen SPARQL-kyselyn kolmikosta

JDOQL-kysely aloitetaan SELECT-lauseella. SPARQL-kolmikon subjekti-, predikaatti- ja objektiosa käydään järjestyksessä läpi ja muodostetaan JDOQL-kyselyn sisältö niiden arvojen perusteella. JDOQL-kyselyn FROM-lause muodostuu SPARQL-kolmikon predikaatin tyypin perusteella. Jos kyseessä on tietotyyppiominaisuus, haetaan MongoDB:n *DatatypePropertyTriple*-kokoelmasta. Jos taas kyseessä on suhdeominaisuus, haetaan *ObjectPropertyTriple*-kokoelmasta.

Sitten muodostetaan JDOQL-kyselyn WHERE-lauseke, johon määritellään tietokannasta haettavien olioiden rajoitteet kolmikon subjektin, predikaatin ja objektin arvojen perusteella. Tietotyyppiominaisuuden tapauksessa *DatatypePropertyTriple*-dokumentin *objectValue*-kentän arvoa rajataan kolmikon objektin arvon perusteella käyttäen JDOQL-kyselykielen implisiittisiä parametreja (katso kohta 2.3.2). Operaattoreita ovat tyypilliset vertailuoperaattorit, kuten yhtäsuuruus, pienempi kuin, suurempi kuin jne. Suhdeominaisuuden tapauksessa kolmikon objektin arvo toimii rajoitteena *ObjectPropertyTriple*-dokumentin *objectUri*-kentälle. Lopuksi käsitellään vielä kolmikon subjekti, josta saadaan rajoite dokumentin *subjectUri*-kentälle. Kuten aikaisemmin todettiin, käytetään tapauksissa, joissa kolmikon subjekti tai objekti on muuttuja, *subjectUri*- tai *objectUri*-kentän rajoitteina kyseiseen muuttujaan aikaisempien kyselyiden suorituksessa sidottuja URI-tunnisteita. Jos aikaisempia tuloksia muuttujalle ei vielä ole, jää kenttä tyhjäksi, eli kentän perusteella ei rajata kyselyä.

Esimerkiksi kuvan 5.1 SPARQL-kyselygraafista muodostuisi kaksi JDOQL-kyselyä, jotka on esitetty ohjelmassa Ohjelma 22. Ensin luotaisiin ylempänä oleva JDOQL-kysely, jolla haettaisiin URI-tunnisteet ”John Smith” -nimisille henkilöille. ”John Smith” -merkkijono välitettäisiin kyselyn suorituksessa parametrin *:param1* arvona. Tämän jälkeen luotaisiin alempana oleva kysely, jolla haettaisiin edellisen kyselyn tuloksena saatujen henkilöiden ohjaamat lentokoneet. Esimerkin vuoksi oletetaan, että edellisen kyselyn suorituksesta saatiin tulokseksi henkilö, jonka URI on *http://www.example.org/example-ontology#Person_123*. Jos jokin edellisen kyselyn tuloksena saatu henkilö ei toimi minkään lentokoneen pilottina, tippuisi kyseinen URI tämän kyselyn tuloksia käsiteltäessä pois *?person*-muuttujaan sidotusta tulosjoukosta. Tuloksena on *?person*-muuttujaan ja *?aircraft*-muuttujaan sidotut URI-tunnisteiden joukot, joille kyselygraafin väittämät pätevät.

```
SELECT
FROM DatatypePropertyTriple
WHERE predicate == "<http://www.example.org/example-ontology#hasName>" &&
      objectValue == :param1 &&
      subjectType == "<http://www.example.org/example-ontology#Person>"

SELECT
FROM ObjectPropertyTriple
WHERE predicate == "<http://www.example.org/example-ontology#pilots>" &&
      objectType == "<http://www.example.org/example-ontology#Aircraft" &&
      subjectType == "<http://www.example.org/example-ontology#Person>" &&
      subjectUri == "<http://www.example.org/example-ontology#Person_123>"
```

Ohjelma 22. *Kuvan 5.1 SPARQL-kyselygraafista muodostuvat JDOQL-kyselyt*

Käytetyn JDO-toteutuksen puutteellinen MongoDB-tuki aiheutti joitakin ongelmia JDOQL-kyselyiden muodostamisessa. Esimerkiksi tiettyä mallia noudattavien merkkijonojen hakemista SQL:n LIKE-operaattorin tapaan ei pystytty toteuttamaan tehokkaasti. JDOQL-kyselykielessä tällaiset kyselyt toteutettaisiin käyttäen JDOQL-kielen säännöllisen lausekkeen parametrinaan ottavaa *matches(String regExpPattern)*-merkkijono metodia [1]. DataNucleus JDO-toteutus ei kuitenkaan muunna tällaista kyselyä MongoDB-kyselykielelle, vaan lataa kaikki MongoDB-kokoelman dokumentit ja suorittaa haun muistinvaraisesti.

Samoin merkkikokoriippumattomien kyselyjen toteutusta hankaloitti sama edellä mainittu ongelma *toLowerCase()*- ja *toUpperCase()*-metodeille. Ongelma voidaan kuitenkin kiertää lisäämällä tietokantaan tallennettaville olioille erillinen kenttä merkkijonoriippumattomia kyselyitä varten, johon alkuperäisen kentän arvo on tallennettu muuttaen kaikki kirjaimet isoiksi tai pieniksi. Esimerkiksi tämä tarkoittaisi kuvassa 4.4 esitetyn tietomallin *DatatypePropertyTriple*-luokalle *objectValueCaseInsensitive*-kentän lisäystä, johon olisi tallennettu alkuperäisen *objectValue*-kentän arvo merkkikokoriippumattomassa muodossa. Tämä ja säännöllisten lausekkeiden käyttäminen kyselyissä jätettiin kuitenkin jatkokehitykseen.

5.2 Kyselyn vastauksen muodostaminen

SPARQL-kyselygraafin perusteella muodostettujen JDOQL-kyselyjen suorittamisesta saatiin tuloksena kyselyn muuttujiin sidottuja URI-joukkoja. Jokainen URI on yhden johonkin tiettyyn ontologialuokkaan kuuluvan instanssin tunniste. Kyselyn suorituksen viimeisessä vaiheessa haetaan tulokseksi saatujen instanssien tunnisteiden perusteella kyseisten instanssien tietotyyppiominaisuudet (*DatatypePropertyTriple*) ja suhdeominaisuudet (*ObjectPropertyTriple*), joista *JDOInstancePopulator* kokoaa kuvassa 4.7 esitetyn tietorakenteen mukaisia *Instance*-olioita.

Instanssien kokoaminen kolmikoista tapahtuu hyvin suoraviivaisesti luomalla jokaisesta erillisestä instanssin tunnisteesta *Instance*-olio ja lisäämällä tälle oliolle sen tietotyyp-

piominaisuudet ja suhdeominaisuudet. Nämä *Instance*-oliot palautetaan vastauksena hakupalvelun käyttäjälle.

5.3 Testaus

Ontologian tiedonhakupalvelu testattiin yksikkötestein hyödyntäen JUnit-yksikkötestikehystä [10]. Kattavien yksikkötestien avulla voitiin myös varmistaa palvelun oikea toiminta muutosten jälkeen, esimerkiksi suorituskykyoptimointeja tehdessä. Palvelulle tehtiin kahdenlaisia yksikkötestejä: palvelun toiminnan oikeellisuuden varmistavia yksikkötestejä ja palvelun suorituskykyä testaavia testejä.

Testausympäristönä toimi ontologian tiedonhakupalvelun kehitykseen käytetty tietokone. Tietokoneen kokoonpano on esitetty taulukossa Taulukko 3. Testauksen aikana koneessa oli ajossa MongoDB-tietokantasovellus, sekä Eclipse-ohjelmointiympäristö, jota käyttäen yksittäiset testit ajettiin.

Taulukko 3. *Testauksessa käytetyn tietokoneen kokoonpano*

Prosessori	Intel Core i5-3470 @ 3,20 GHz
Keskusmuistia	4,00 GB
Kiintolevytyyppi	HDD 7200 rpm (500 GB)
Käyttöjärjestelmä	Windows 7 Enterprise 64-bit

Jokaiselle toteutetulle luokalle tehtiin oikean toiminnan testaavia testitapauksia. Näissä testeissä kutsuttiin luokan tarjoamaa palvelua laillisin syöttein ja tarkistettiin JUnit-kehyksen *Assert*-metodeja hyödyntäen tulosten oikeellisuus. Toimintaa virhetilanteissa testattiin antamalla virheellistä syötettä, esimerkiksi virheellinen kyselygraafi JDOQL-kyselyn muodostavalle metodille, ja tarkistettiin, että toiminta on edelleen loogista. Useimmiten tällaisissa tilanteissa se on oikeanlaisen poikkeuksen heittäminen.

Ontologian tiedonhakupalvelulle tehtiin suorituskykyä testaavia yksikkötestejä. Yksikkötesteillä testattiin seuraavanlaisia järjestelmän kannalta oleellisia kyselytyyppejä. Instanssilla tarkoitetaan luvussa 4.5 esitettyä tietorakennetta, joka kuvaa yhtä ontologialuokan yksilöä.

1. Haetaan kaikki tietyn ontologialuokan instanssit

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX example-ontology: <http://www.example.org/example-ontology#>
SELECT ?person
WHERE {
      ?person rdf:type example-ontology:Person
}
```

Ohjelma 23. *Esimerkki 1. kyselytyypin kyselystä SPARQL-kielellä esitettynä.*

2. Haetaan yksi ontologian instanssi

```
PREFIX example-ontology: <http://www.example.org/example-ontology#>
SELECT ?p ?o
WHERE {
      example-ontology:Person_123 ?p ?o
}
```

Ohjelma 24. *Esimerkki 2. kyselytyypin kyselystä SPARQL-kielellä esitettynä.*

3. Haetaan ontologialuokan instansseja tietotyyppiominaisuuden arvon perusteella.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX example-ontology: <http://www.example.org/example-ontology#>
SELECT ?person
WHERE {
      ?person rdf:type example-ontology:Person .
      ?person example-ontology:hasName "John Smith"^^xsd:string .
}
```

Ohjelma 25. *Esimerkki 3. kyselytyypin kyselystä SPARQL-kielellä esitettynä.*

4. Haetaan ontologialuokan instansseja suhdeominaisuuden kohteen perusteella.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX example-ontology: <http://www.example.org/example-ontology#>
SELECT ?person
WHERE {
      ?person rdf:type example-ontology:Person .
      ?person example-ontology:pilots example-ontology:Aircraft_54 .
}
```

Ohjelma 26. *Esimerkki 4. kyselytyypin kyselystä SPARQL-kielellä esitettynä.*

5. Haetaan ontologialuokan instansseja suhdeominaisuuden lähteen perusteella.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX example-ontology: <http://www.example.org/example-ontology#>
SELECT ?aircraft
WHERE {
      ?aircraft rdf:type example-ontology:Aircraft .
      example-ontology:Person_123 example-ontology:pilots ?aircraft
}
```

Ohjelma 27. *Esimerkki 5. kyselytyypin kyselystä SPARQL-kielellä esitettynä.*

6. Haetaan ontologialuokan instansseja suhdeominaisuuden tyypin perusteella.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX example-ontology: <http://www.example.org/example-ontology#>
SELECT ?person ?aircraft
WHERE {
      ?person rdf:type example-ontology:Person .
      ?aircraft rdf:type example-ontology:Aircraft .
      ?person example-ontology:pilots ?aircraft
}
```

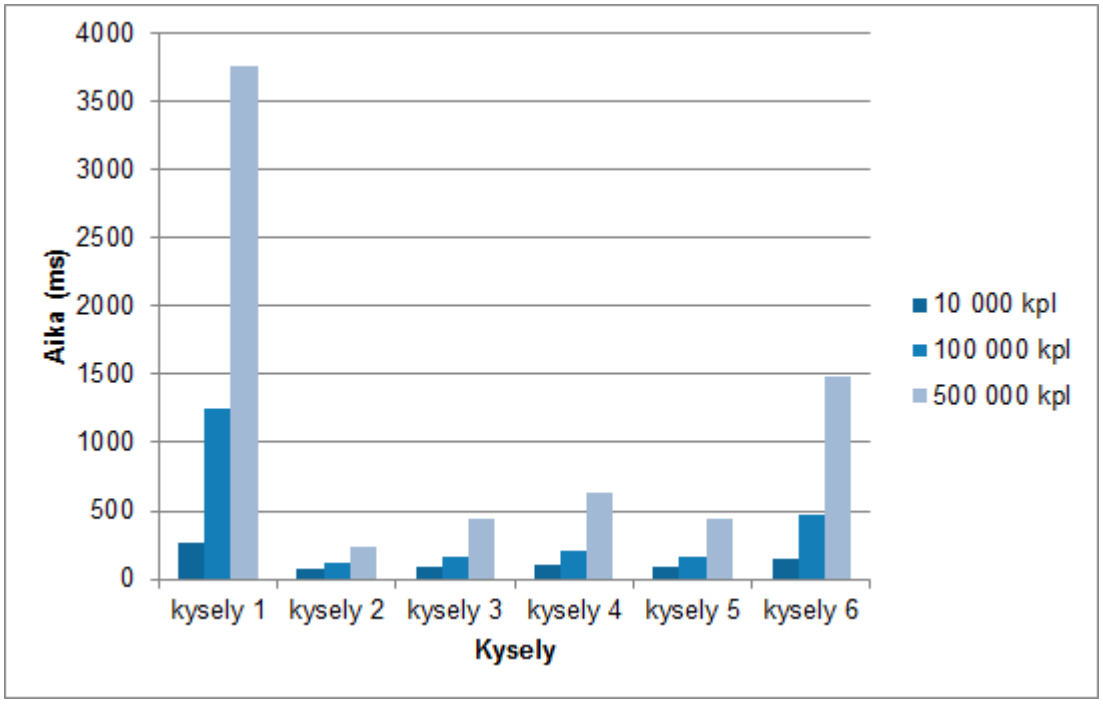
Ohjelma 28. *Esimerkki 6. kyselytyypin kyselystä SPARQL-kielellä esitettynä.*

Tietokantaan populoitiin testidataksi ennen kyselyiden suoritusta eri määriä ontologialuokkien instansseja. Populoidut instanssit jakaantuivat tasaisesti 20 eri ontologialuokkaan, joista jokaisella luokalla oli 5 tietotyyppiominaisuutta ja 5 suhdeominaisuutta, joiden kohdeinstanssit valittiin satunnaisesti. Kyselyt suoritettiin kolmelle eri instanssimäärälle: 1000 instanssille (10 000 kolmikkoa tietokannassa), 10 000 instanssille (100 000 kolmikkoa tietokannassa) ja 50 000 instanssille (500 000 kolmikkoa tietokannassa). Yksi kolmikko on aina yksi dokumentti MongoDB-tietokannassa, kuten luvussa 4 esitetyssä tietomallissa on kuvattu. Testidatan jakautuminen tietokannassa on esitetty taulukossa Taulukko 4.

Taulukko 4. *Ontologiapalvelun testaukseen käytetyn testidatan jakautuminen tietokannassa eri instanssimäärillä.*

Kolmikoita tietokannassa	Instansseja	Instansseja per ontologialuokka
10 000	1000	50
100 000	10 000	500
500 000	50 000	2500

Kuvaajassa 5.3 on esitetty kyselyjen suoritukseen kulunut keskimääräinen aika näillä kolmikkomäärillä. Keskimääräinen aika on mitattu keskiarvona kyselyn 5 peräkkäisen suorituksen ajoista.



Kuva 5.3: Kyselyiden suoritukseen kulunut keskimääräinen aika eri tietomäärillä.

Tuloksista on huomattavissa, että kysely 1, eli kaikkien tietyn ontologialuokan instanssien hakeminen, hidastuu huomattavasti palautettavan instanssimäärän kasvaessa. Kyselyn 1 suoritukseen kuluneissa ajoissa tulee kuitenkin ottaa huomioon, ettei tulosten mää-

rää rajoitettu mitenkään. Tällöin kaikkien ontologialuokan instanssien hakemiseen kulunut aika riippuu ennemminkin kyseisen ontologialuokan instanssien lukumäärästä tietokannassa, kuin kaikkien instanssien kokonaismäärästä tietokannassa. Tämä aiheuttaa kuvaajasta nähtävän hitauden tämän kyselyn suhteen. Myös kyselytyypin 6 kysely on isoja tietomääriä käsiteltäessä hitaahko. Muiden kyselytyyppien suoritus on tasaista ja onnistuu kohtuullisessa ajassa isoillakin tietomäärillä. Tietokantaa ei ollut indeksoitu kyselyjä suoritettaessa. Sopivien indeksien luominen MongoDB:n kokoelmiin saattaisi nopeuttaa kyselyjä.

6. TULOKSET JA JATKOKEHITYSMAHDOLLISUUDET

Diplomityössä saatiin toteutettua toimiva ontologian tiedonhakupalvelu käyttäen MongoDB-dokumenttitietokantaa kolmikkotietokantana JDO-rajapinnan kautta. Hakupalvelulla pystytään hakemaan ontologiasta instanssidataa. Hakupalvelu vastaanottaa SPARQL-tyylisiä kyselyitä, parsii ne kyselypuuksi, ja muuntaa kyselypuun JDOQL-kyselyiksi, jotka ajetaan JDO-rajapintaa käyttäen MongoDB-tietokannassa.

DataNucleus JDO-toteutuksen ja MongoDB-dokumenttitietokannan yhteensopivuus oli vielä osittain puutteellista, mikä aiheutti toteutuksessa ongelmia hakujen suorituskyvyn suhteen. Tästä johtuen osaa aikaisemman, relaatiotietokannan päälle toteutetun, tiedonhakupalvelun ominaisuuksista ei pystytty siirtämään JDO/MongoDB-toteutukseen ilman suorituskykyongelmia isoilla tietomäärillä. Tämä nähtiin myös suorituskykytestien tuloksissa kyselytyyppien 1 ja 6 kyselyjen hitautena.

JDO-toteutus ei tukenut useiden JDOQL-kyselykielen rakenteiden muuntamista MongoDB-kyselykielelle, jolloin JDO-toteutus suoritti tällaiset kyselyiden täysin muistinvaraisesti. Tämä on suuren muistinkulutuksen lisäksi hidasta. Tällaisia ominaisuuksia olivat säännöllisten lausekkeiden käyttäminen kyselyissä SQL:n LIKE-operaattorin tapaan, kirjainkokoriippumattomat kyselyt ja GROUP BY-rakennetta käyttävät kyselyt.

Toteutetussa tiedonhakupalvelussa on siis vielä paljon jatkokehitystarpeita. JDO-toteutuksen uusien versioiden myötä JDO-toteutuksen tuki MongoDB-tietokannalle tulee kokoajan paranemaan. Tämä mahdollistaa tulevaisuudessa suorituskyvyn parantamisen useissa erilaisissa kyselyissä. Tällaisia ovat mm. edelläkin mainitut tiedon hakeminen ontologiasta säännöllisillä lausekkeilla, kirjainkokoriippumattomat haut ja GROUP BY-rakennetta käyttävät kyselyt. Kirjainkokoriippumattomat haut olisi myös mahdollista toteuttaa hyvällä suorituskyvyllä luvussa 5 esitetyllä tavalla lisäämällä *DatatypePropertyTriple*-luokalle ylimääräinen kenttä ominaisuuden arvon kirjainkokoriippumattomassa muodossa tallennusta varten. Tämä ominaisuus jää kuitenkin jatkokehitykseen.

Toteutuksessa on myös vielä puutteita sellaisten SPARQL-kyselygraafien suorituksessa, joissa graafin kolmikossa predikaatti on muuttuja, eikä ominaisuuden URI. Esimerkiksi ohjelman Ohjelma 29 SPARQL-kyselyä ei ontologian tiedonhakupalvelu pysty käsittelemään.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX example-ontology: <http://www.example.org/example-ontology#>
SELECT ?person
WHERE {
      ?person rdf:type  <http://www.example.org/example-ontology#Person> .
      ?person ?pre <http://www.example.org/example-ontology#Aircraft_54> .
}
```

Ohjelma 29. *SPARQL-kysely, jossa kolmikon predikaattina on muuttuja.*

Kyselyssä haettaisiin kaikki sellaiset henkilöinstanssit, joilla on mikä tahansa suhde annettuun lentokoneeseen. Hakupalvelu ei siis tällä hetkellä hyväksy kyselygraafissa kolmikoita, joissa predikaatti on muuttuja. Tämän ominaisuuden toteutus on myös eräs jatkokehitysmahdollisuus.

7. YHTEENVETO

Diplomityön tavoitteena oli toteuttaa ontologian tiedonhakupalvelu käyttäen ontologiadatan tietovarastona MongoDB-dokumenttitietokantaa JDO-ohjelmointirajapintaa hyödyntäen. Ontologian tiedonhakupalvelusta oli olemassa relaatiotietokantaa käyttävä toteutus, josta voitiin uudelleenkäyttää komponentteja uudessa toteutuksessa. Tietokantaa käyttävä osuus ontologian tiedonhakupalvelusta toteutettiin uudestaan hyödyntäen JDO-ohjelmointirajapintaa. Tähän kuului mm. SPARQL-tyylisten kyselyjen muuntaminen JDOQL-kyselykielelle.

MongoDB-tietokannan ja relaatiotietokantojen eroista johtuen, sekä ontologiaskeeman ajonaikaisen muokkauksen mahdollistamiseksi tietokannan tietomalli täytyi suunnitella alusta alkaen uudestaan. JDO:n ja MongoDB-tietokannan yhteensopivuus oli vielä käytetyssä DataNucleus JDO-toteutuksessa puuttellista, mikä aiheutti ongelmia suorituskyvyltään hyvän tietomallin suunnittelussa ja suorituskyvyltään hyvien JDOQL-kyselyjen muodostuksessa. Osa tietomallin suunnitteluratkaisuista jouduttiin siis tekemään puhtaasti suorituskyvyn optimoimiseksi MongoDB-tietokantaa käytettäessä. Alla olevan tietokannan vaihtaminen olisi kuitenkin mahdollista, sillä JDO-rajapinta piilottaa käytetyn tietokannan. Toteutus testattiin toiminnallisuudeltaan ja suorituskyvyltään yksikkötestein.

Diplomityössä saatiin toteutettua toimiva ontologian tiedonhakupalvelu. Myös vaatimuksena ollut ontologian ajonaikainen muokkaaminen on hakupalvelun kannalta mahdollista. Osaa relaatiotietokantatoteutuksen ominaisuuksista ei saatu JDO/MongoDB-yhteensopivuusongelmien takia siirrettyä hyvällä suorituskyvyllä uuteen toteutukseen. Tällaisia ominaisuuksia olivat mm. säännöllisten lausekkeiden käyttäminen tietokantahauissa. JDO/MongoDB-yhteensopivuus tulee kuitenkin tulevaisuudessa jatkuvasti parantumaan, mikä mahdollistaa jatkokehityksessä suorituskyvyn parantamisen useissa erilaisissa kyselyissä.

LÄHTEET

[1] Apache, Java Data Objects specification 3.1, Oct 11 2013, 517p. Saatavissa: http://svn.apache.org/viewvc/db/jdo/trunk/specification/OOO/JDO_3_1-rc1.pdf?view=co

[2] Apache JENA documentation [WWW]. [viitattu: 19.1.2015]. Saatavissa: https://jena.apache.org/index.html

[3] BSON specification, [WWW]. [viitattu: 28.1.2015]. Saatavissa: http://bsonspec.org/

[4] DataNucleus AccessPlatform 4.0 Documentation [WWW]. [viitattu: 20.1.2015]. Saatavissa: http://www.datanucleus.org/products/datanucleus/index.html

[5] Ecma International, The JSON Data Interchange Standard, October 2013, 1st Edition, 14p. Saatavissa: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

[6] T. R. Gruber, Toward Principles for the Design of Ontologies Used for Knowledge Sharing, Stanford Knowledge Systems Laboratory, Aug 23 1993, 23p. Saatavissa: http://tomgruber.org/writing/onto-design.pdf

[7] Hibernate ORM [WWW]. [viitattu: 27.1.2015]. Saatavissa: http://hibernate.org/orm/

[8] Java documentation [WWW]. [viitattu: 29.1.2015]. Saatavissa: http://docs.oracle.com/javase/

[9] Javafx GUI library [WWW]. [viitattu: 06.04.2015]. Saatavissa: docs.oracle.com/javafx/

[10] JUnit. Unit testing framework for Java. [WWW]. [viitattu: 26.1.2015]. Saatavissa: http://www.junit.org/

[11] MongoDB manual [WWW]. [viitattu: 19.1.2015]. Saatavissa: http://docs.mongodb.org/manual/

[12] Nitzschke Marcus, Implementierung einer RDF-Storage Lösung für MongoDB, university of Leipzig, August 2011, 63p. Saatavissa: http://www.kendix.org/media/files/thesis.pdf

[13] Programming Language Popularity [WWW]. [viitattu: 29.1.2015]. Saatavissa: http://langpop.com/

[14] Shashank Tiwari. Professional NoSQL. USA 2011, John Wiley & Sons, Inc. 361 p.

[15] Dominik Tomaszuk, DOCUMENT-ORIENTED TRIPLESTORE BASED ON RDF/JSON, in: Studies in Logic, Grammar and Rhetoric, Vol. 22 Logic, Philosophy and Computer Science, Institute of Computer Science, University of Bialystok, 2010, pp. 125-140. Saatavissa: http://logika.uwb.edu.pl/studies/download.php?volid=35&artid=dt

[16] W3C, OWL Web Ontology Language, 10.02.2004 [WWW]. [viitattu: 16.1.2015]. Saatavissa: http://www.w3.org/TR/owl-ref/

[17] W3C, OWL 2 Web Ontology Language, 11.12.2012 [WWW]. [viitattu: 16.1.2015]. Saatavissa: http://www.w3.org/TR/owl2-primer/

[18] W3C, RDF Schema 1.1, 25.2.2014 [WWW]. [viitattu: 16.1.2015]. Saatavissa: http://www.w3.org/TR/rdf-schema/

[19] W3C, Resource Description Framework (RDF), 25.02.2014 [WWW]. [viitattu: 16.1.2015]. Saatavissa: http://www.w3.org/RDF/

[20] W3C, Semantic Web - Vocabularies [WWW]. [viitattu 22.12.2014]. Saatavissa: http://www.w3.org/standards/semanticweb/ontology

[21] W3C, Semantic Web - Inference [WWW]. [viitattu 22.12.2014]. Saatavissa: http://www.w3.org/standards/semanticweb/inference

[22] W3C, Wine ontology [WWW]. [viitattu 10.3.2015]. Saatavissa: http://www.w3.org/TR/2004/REC-owl-guide-20040210/wine.rdf

[23] W3C, SPARQL 1.1 [WWW]. [viitattu: 19.1.2015]. Saatavissa: http://www.w3.org/TR/sparql11-overview/

[24] M.M. Zloof. Query-by-Example: A Data Base Language, in: IBM Systems Journal 16, 1977, pp. 324-343.