A Semantic Matchmaking System for Online Dating

by

Emily Wilson

Bachelor of Science, Mount Allison University, 2011

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of

Master of Computer Science

in the Graduate Academic Unit of Computer Science

Supervisor: Kenneth B. Kent, Computer Science

Examinining Board: Michael Fleming, Computer Science, chair
Dawn MacIsaac, Computer Science
Sal A.M. Saleh, Electrical and Computer Engineering

This thesis is accepted by the
Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

May 2013

© Emily Wilson 2013
ABSTRACT

The popularity of the online dating industry has grown immensely over the past decade. There is an abundance of online dating websites with various features to attract users. The Semantic Web is a major endeavor that aims to have information on the web be not only machine-readable but also machine understandable. Online dating is a good candidate for such a service since it is based primarily on user provided information. By organizing user information in a comprehensive knowledge base users can be matched more efficiently. In this thesis, semantic web tools, such as ontology languages and reasoning software, were investigated to determine which ones would work best in the online dating website model. An ontology is presented that models the properties of user profiles on a dating website, as well as a semantic matching system. Rules and reasoning are used to infer additional facts about users to be used in the matching process, therefore providing a more accurate match. This prototype, a semantic matchmaking system for online dating, has been implemented in Java using the Jena interface. Results of running the prototype on a commercial dating website are reported.
ACKNOWLEDGEMENTS

Thanks to my supervisor Dr. Kenneth Kent for all of his help, support and supervision. I would also like to thank him for his guidance and encouragement during the completion of my MCS.
# Table of Contents

ABSTRACT ................................................................................................................................. ii

ACKNOWLEDGEMENTS ......................................................................................................... iii

List of Tables ................................................................................................................................... viii

List of Figures .................................................................................................................................. ix

Chapter 1 ........................................................................................................................................ 1

Introduction ................................................................................................................................... 1

1.1 Overview: Online Dating ........................................................................................................ 1

1.2 Introduction to the Semantic Web .......................................................................................... 2

1.3 Thesis Objectives .................................................................................................................. 4

1.4 Thesis Organization ............................................................................................................... 4

Chapter 2 ........................................................................................................................................ 5

Background ................................................................................................................................... 5

2.1 Semantic Matchmaking ......................................................................................................... 5

2.2 Current Online Dating Technologies .................................................................................... 8

2.3 Previous Semantic Matchmaking Prototype .......................................................................... 11

2.3.1 Previous Ontology .......................................................................................................... 11

2.3.2 Previous Knowledge Base ............................................................................................ 12

Chapter 3 ........................................................................................................................................ 13

Rule Languages and Tools .......................................................................................................... 13

3.1 Rule Languages .................................................................................................................... 13

3.2 Jena .......................................................................................................................................... 14
3.2.2 Architecture of Jena

3.2.2.1 RDF API

3.2.2.2 Ontology API

3.2.2.3 SPARQL API

3.2.3 Rule Languages and Reasoning with Jena

3.3 Sesame

3.3.1 The Repository API

3.3.2 The Graph API

3.3.3 The SAIL API

Chapter 4

4.1 Proposed Matching System

4.1.1 Statistics Canada and U.S. Census Data Integration

4.2 Building on the Matching Implementation in the Previous Prototype

4.2.1 Creating User RDF

4.2.2 SPARQL Queries

4.3 Ontology Languages

4.3.1 Features of RDFS and OWL

4.3.2 Reasoning and Scalability

4.4 Ontology Design

4.4.1 Previous Ontology

4.4.2 Statistics Canada and U.S. Census Ontology

4.4.3 Level 4 Ontology

4.5 Knowledge Base Design
List of Tables

Table 1. The structure of a city's table ..................................................................................... 29
Table 2. Statistics for language used the most at home in Fredericton ................................. 30
Table 3. Example facts used in the beginning of a user's questions ..................................... 59
Table 4. Language Statistics about Windsor, NS .................................................................. 69
Table 5. Generation Statistics for Windsor, NS ..................................................................... 69
Table 6. One Year Mobility Statistics for Windsor, NS ......................................................... 69
Table 7. Five Year Mobility Statistics for Windsor, NS .......................................................... 70
Table 8. Income Statistics for Windsor, NS .......................................................................... 70
Table 9. Test case users and their match levels .................................................................... 74
List of Figures

Figure 1. Jena Architecture Overview [41]........................................................................................................ 15
Figure 2. Structure of Inferencing Model in Jena [42] ...................................................................................... 20
Figure 3. Sesame architecture [51] .................................................................................................................. 22
Figure 4. Four level architecture of matching system ....................................................................................... 27
Figure 5. Ontology structure associated with the religion attribute............................................................... 41
Figure 6. Ontology structure associated with income property ...................................................................... 43
Figure 7. Ontology structure associated with level 4 ...................................................................................... 44
Figure 8. Steps performed in system prototype ............................................................................................... 45
Figure 9. User profile on the online dating website ......................................................................................... 64
Figure 10. User question from the online dating website ................................................................................ 65
Figure 11. Screenshot of MySQL database entry for test user ...................................................................... 67
Figure 12. Screenshot of Eclipse's output when queried for inferred facts ...................................................... 68
Figure 13. Screenshot of income question asked to the user ........................................................................... 71
Figure 14. Screenshot of mobility question asked to the user ......................................................................... 71
Figure 15. Snapshot of results from running the first test case ...................................................................... 72
Figure 16. Snapshot of results after altering user 002's level 1 properties ..................................................... 73
Figure 17. Snapshot of results from running second test case ....................................................................... 75
Figure 18. Registration information for Jane Doe ............................................................................................ 76
Figure 19. List of matches returned the first time Person1 requests their matches ................................. 77
Figure 20. List of matches returned after Person1 has specified an age restriction .................................. 77
Figure 21. John is returned as Person1's top match ......................................................................................... 79
Figure 22. Person1's results after John no longer matches their preference on level 1 .... 79
Chapter 1

Introduction

The popularity of the online dating industry has grown immensely over the past decade [24]. According to the 2012 report, “Dating Services in the US: Market Research Report”, there are over 1500 dating sites in the U.S. market alone. People may seek out online dating websites as a means of finding a potential partner more efficiently. Users enter information about themselves as well as the characteristics that they are seeking in a partner and that information is used in a matchmaking process to select the matches that best suit that user. Producing a set of matches for a user that they will deem suitable is one of the major challenges in the online dating industry [13]. It is the job of the matchmaking system to use the user provided information to find the best matches for a user. Using semantic web techniques can enhance the organization and integration of user data.

1.1 Overview: Online Dating

When users join an online dating website they are asked a series of questions about themselves and what they are looking for. The answers to these questions are used to eliminate other users who do not meet their criteria from their list of matches and also used to rank partial-matching members in terms of how well they meet the user's criteria for a partner. There are several downfalls of using solely this type of approach when implementing an online dating system. Firstly, there is an assumption that users have been completely honest when entering information about themselves and have not
embellished certain characteristics. This is a commonly found problem on online dating sites [15]. Secondly, to provide an accurate match, some websites ask users many range based questions and users may become frustrated with the amount of information they have to enter and discontinue the registration process with the website. Conversely, some websites ask too few questions and users are returned matches that they deem unsuitable for reasons not covered in the questions. Despite the widespread market, internal research indicates a general dissatisfaction in the quality of matches generated by mainstream sites [13]. Popular consumer product reviews search engine, Consumer Search, reported in 2011 that “negative reviews far outweigh positive reviews when it comes to online dating sites”. This is further supported by the number of niche sites that are growing and expanding their market shares [40] [53]. In terms of appropriately matching their users, online dating sites need to be able to ask a limited number of relevant questions, but also have enough information to make an intelligent match.

1.2 Introduction to the Semantic Web

Currently, most of the data on the web can only be interpreted and understood by human users. The semantic web aims to enhance the web by enriching its content with meaningful metadata (structured information that describes an information resource) that is machine-readable. It is largely based on the Resource Description Framework (RDF) [19], which organizes the meaning of the data into triples. Each triple contains a subject, a property and an object describing some data. It also uses ontologies to formally represent the data concepts and the relationships between them, and uses automated reasoning to apply inferences.
Online dating is a good candidate for a service such as the semantic web since it is based primarily on user provided information. The semantic web allows for large data sets to be queried and linked together. Therefore, a case can be made that by applying semantic web techniques to online dating one can potentially link user information to additional data sets and consider more than just parameter based criteria. An important question that must therefore be addressed is, how one can utilize semantic web technologies to build a semantic framework for an online dating model. More specifically, what ontology language and reasoning engine is best suited to perform such a linking between user information and additional data sets? Moreover, how can one develop the ontology to effectively express the online dating domain? Once the ontology has been developed, evaluation of whether the ontology has true semantic expressive power for use in the system is needed. By effectively organizing user information into an ontology, rules can be written to be used by a reasoner (reasoning software) to infer additional properties about the user to be further used in the matching process. By inferring additional properties about a user based on the information they have provided another level of matching can be added without burdening them with additional questions. As a result, by using leading edge semantic technologies, matching results can be improved based on underlying facts and rules extracted from users’ input and provide accurate matches of higher semantic quality. The application of such a system could have clear benefits to not only an online dating model, but to other similar matchmaking models discussed in the next chapter.
1.3 Thesis Objectives

The main objectives of this thesis are to:

- Study and evaluate current matchmaking techniques.
- Extend the existing ontology that structurally organizes user input.
- Design an ontology to represent geographical information.
- Research potential reasoners that will be suitable in terms of performance and scalability.
- Build on the current matchmaking system to provide an inferred level of matching by using external sources of data such as Statistics Canada and the U.S. Census.
- Evaluate the performance and quality of the matchmaking system.

1.4 Thesis Organization

This thesis is organized in the following manner. A background study of previous semantic matching techniques, current matchmaking websites, along with a brief description of the matching system that was built upon in this thesis are presented in Chapter 2. In Chapter 3, the rule languages and their implementation that were used are introduced. Next, in Chapter 4 the design of the knowledge base is presented. In Chapter 5, the implementation details of the prototype and the results are discussed. Finally, in Chapter 6 the conclusions and future work are described.
Chapter 2

Background

This chapter describes the general concepts of the semantic web and matchmaking in online dating. It also explores previous approaches to matchmaking using semantic web tools. In Section 2.1 semantic techniques that have been previously employed in matchmaking systems are introduced. This is followed, in Section 2.2, by detailed descriptions of today's top online dating websites, their strengths, and their shortcomings. Finally, Section 2.3 concludes this chapter by describing the previous prototype that this project is built upon.

2.1 Semantic Matchmaking

When discussing a matchmaking system the term Offers will be used to describe one side of a match and Requests will be used to describe the other. Therefore the goal is to efficiently match Offers to Requests. There exist two main approaches to automated matchmaking, namely syntactic ones and semantic ones [9]. Syntactic approaches often use keyword based search methods to compare similarities between Offers and Requests to be matched. They make little use of semantics about available data and are generally a weaker form of matching. Thus employing semantic web techniques has become an attractive option for various kinds of matching. In semantic approaches ontologies are often used. Ontologies provide background knowledge and formalize offers and requests with semantics.

An existing application of semantic matchmaking that can be easily contrasted
with online dating is that of applying it to employment or human resource management. In an online recruitment system, companies seeking employees present a list of requirements potential employees must meet, much like the list of requirements a user searching for a partner presents in an online dating system. Businesses specify must-have constraints and desired constraints in terms of skills and competency level. Job seekers who do not meet the must-have constraints are eliminated, and the remaining job seekers are then ranked in terms of how well they meet the desired constraints. The same process occurs in the online dating matchmaking process on a larger scale (in that there are usually more constraints). Another closely related field of semantic matching is in electronic markets. To match Offers to Requests, both similarity-based and logic based matchmaking approaches have been taken [29] [8]. Colucci et al. [8] present a framework for matching the most skilled individuals to a given task by using description logic inferences in an ontology supported framework. They find that this approach overcomes simple subsumption matching (matching based on subsumption hierarchies of the domain ontology) and allows match ranking and classification. Hybrid strategies using both similarity-based and logic based matchmaking approaches have also been proposed [23].

Some approaches express Offers and Requests as Description Logic concepts and exploit inference methods to compute different matches. In [21], subsumption checking is used to compute several kinds of matches. They examine potential matches for a request, that is, an Offer matching only a portion of the Request. In their approach their reasoner needs to check the satisfiability of a Request with each existing Offer, and therefore the question of scalability arises. They found that if an advertisement has already been
classified then the reasoning time to respond to a request is very small. However, the
process of classifying advertisements is quite time consuming, therefore performance will
be slower if a lot of new advertisements need to be classified.

There have already been semantic approaches to creating matches in an online
dating system, by matching user profiles [7]. In this application they use a specifically
tailed form of Description Logic for describing profiles and model the matching
process as a special reasoning service about profiles. The main focus of this approach is
to create a matching system that is able to match profiles when the Requester's and
Offerer's profiles can have missing or conflicting information, and their algorithm
calculates a penalty based on the missing or conflicting information. They also keep track
of this information so that end users can request justifications for their suggested
matches. However, their approach is very generalized and they use only “hasInterest”
properties to express interest in topics. A similar approach using Description Logics was
proposed for mobile dating in P2P environments [27].

In general, most of the proposed frameworks work for a given circumstance but
are not easy to scale to real-life applications that involve a large number of Offers and
Requests because it is too time consuming to produce Description Logic concepts for
each of them. However, in the proposed application, computing such classifications
offline as much as possible and keeping these classifications in memory will cut down on
the time it takes to classify a user’s information, and only requires classifying new user
information. Most of the approaches that use Description Logic have limited the full
expressiveness of OWL DL to cut down on computational costs. In the proposed
application there is a high degree of information; therefore scalability and performance of
the ontology and reasoner become an important issue that needs evaluation. Also, the current applications of semantic matchmaking in dating systems do not utilize user provided information to infer additional details about the user. As a result, this is a new aspect of the matching that will be implemented. Overall, this application is intended to extend the ideas presented in previous work and explore the benefits and liabilities of using an ontology-based approach in an online dating matchmaking context.

2.2 Current Online Dating Technologies

Dating websites today use some sort of computerized matching algorithm to match their users. Advances in computing power and storage space in servers allows one to process mass amounts of data in a relatively short amount of time, therefore dating websites can ask users many questions to try to record as much information about the user as they can. Most of this information is obtained through initial questionnaires; however, popular online dating website Match.com [45] also generates matches by observing whose profiles users are spending the most time reading and whose profiles other users like them have liked [18]. For example, if a user says that they are not interested in someone who has been married before but browses profiles of users who are divorced then Match.com’s algorithm determines that that user does not weigh that preference as highly as their other preferences. That is, Match.com begins weighing variables differently according to how users behave. While seemingly clever, this can be frustrating for users who stand firm on their preferences but have stumbled across profiles that don’t meet them for one reason or another. Online dating websites use this technique among others to create matches.
There are two main types of sign up processes for online dating websites. On one side of the spectrum some online dating websites try to retrieve as much information about the user as they can by asking a large number of questions in the sign up process. Two of today’s most popular dating websites according to Consumer-Rankings.com, Match.com and eHarmony [39], both ask an extensive number of questions in their registration process which can be quite time consuming for the user. Match.com’s registration process takes approximately 30 minutes [36]. Users fill out several pages of questions that fall into three main categories. The “about me” section consists of approximately 40 questions about the user. Once this section is complete users must fill out 4 more pages of questions about what they are looking for in a partner. Finally, users are asked to write a paragraph about themselves to complete the registration process. By asking so many questions Match.com also allows users to browse other members based on many different search options. One of the negative features of Match.com as noted by Consumer-Rankings.com is that having so many options can be overwhelming.

The registration process for eHarmony takes even longer, approximately 1 hour [36] and consists of 258 questions. Having to answer so many questions can frustrate users and force them to end the registration process, or make the user become bored and choose randomly selected options just to finish the registration process. The goal of asking this many questions is to get a clear idea of a user’s preferences and relationship goals so that eHarmony’s matching system can provide thoroughly compatible matches. By retrieving this amount of data about their users these sites can make matches on a variety of levels and users need not spend much time searching through the site. In fact, eHarmony doesn’t allow users to search for other users at all, they may only see the list
of matches that they have been given. This can be frustrating for users who are not happy with the matches that have been suggested to them.

On the other side of the spectrum some online dating websites ask very few questions to allow for a speedy registration process and it is up to the user to search through pages of potential matches to find what they are looking for. One such example is the dating website Lavalife.com. Lavalife [44] allows users to search for their own matches. Their registration process consists of only a few basic questions about a user’s physical characteristics, personal details, and an about me section titled “in my own words”. This is the only information that users enter about themselves, therefore the registration process is quick and easy. LavaLife allows users to search for other users with specific features. The disadvantage here is that users must then do all the work finding their own matches.

In the proposed prototype a matching algorithm has been developed that takes advantage of both sides of the registration spectrum. The system has been designed to retrieve a reasonable amount of relevant information from users in the registration process and then goes on to use this information in the matching. In addition, semantic web techniques are used to infer additional information about users so that more information can be used in the matching process than what is retrieved from the users initially. This way the registration process can be kept to a reasonable length and thus users are not burdened by additional questions, but still provide enough information to generate quality matches.
2.3 Previous Semantic Matchmaking Prototype

The research presented in this thesis builds on an existing matchmaking prototype. The previous prototype used a two level matching system that did not use any reasoning. Although the prototype did use semantic web techniques, it did not do anything that could not have been done by a different, non-semantic implementation.

2.3.1 Previous Ontology

Since it is important to have a common view of the online dating website profiles, a common understanding of the relevant concepts and relationships between them must be established. The previous matchmaking prototype described user properties, preferences, and relations in an RDFS ontology. An ontology is commonly defined as a specification of a conceptualization. They are structured formal definitions used to describe entities in the domain of interest [26]. Rules are logical constructs that can be used to derive new knowledge or define processes. Used together, ontologies and rules provide a good basis for reasoning and classifying information. Therefore the ontology and rules in this prototype are used to conceptualize the dating domain and define the matching process respectively.

The ontology used in the previous prototype is comprised of classifications of data items for a user's attributes and preferences. It considers each property associated with a user as a separate class, e.g. Employment_Status, Ethnicity, Religion, Body_Type, etc. The options for each of these properties are described as resources and are specified as being of the type of the class, e.g. christianity is of type Religion. New classes and relationships have been added to the ontology in the new prototype.
2.3.2 Previous Knowledge Base

The previous prototype uses a locally stored Knowledge Base (KB) to store the information from the RDFS ontology as well as information regarding a user’s attributes and preferences. The KB consists of object-centric facts, which are structured by the ontology. In the two basic levels of matching implemented in this prototype, SPARQL queries are used on the KB to filter a user’s list of potential matches based on that user’s preferences. The previous prototype does not use a reasoner to reason over the KB, and therefore no new information is inferred. The main changes between the previous prototype and the new version are the newly added inference capabilities. The following changes are made to the previous ontology:

- New properties are added to the ontology to incorporate geographical information about each user.
- A reasoner and a set of rules are implemented to infer new properties about users.
- This newly generated information is used in a new level of matching.
Chapter 3

Rule Languages and Tools

This chapter presents the rule languages and semantic tools that are relevant to this thesis. In Section 3.1, a brief introduction to rule languages that are prevalent on the semantic web today is presented followed in Section 3.2 by an introduction to Jena, the toolkit that is used to implement a portion of the semantic aspects of the prototype. Also in Section 3.2 a detailed description of the APIs that are used in this project is given. Finally, in Section 3.3 the Sesame framework used to store and access the knowledge base is described.

3.1 Rule Languages

Rules and ontologies play a key role in the Semantic Web’s layered architecture. The ontology layer of the Semantic Web has had much development, and Ontology Web Language (OWL) [48] is a W3C recommendation. However, the rule layer of the Semantic Web is far less developed and an active area of research [10]. Rules can be used to express diagnosis rules for engineering applications, business rules for commerce applications, legal reasoning for law applications, eligibility and compliance for medical applications, and in the case of the online dating applications rules can associate properties about the city that a user is from to that user. There have been efforts to standardize inference rules by the creation of popular rule languages such as Rule Markup Language (RuleML) [4], the Semantic Web Rule Language (SWRL) [16], and others. The Rule Interchange Format (RIF) working group of W3C [3] aims to develop a
standard exchange format for rules on the semantic web. The aim of this project was to choose the rule language that best suited the purposes of the matchmaking model. Before examining the types of rules that were used in this project, the toolkit used to create the rules and perform the inferencing is introduced.

### 3.2 Jena

The previous prototype of the matching engine is written in Java so the extension of that matching engine produced during this project was also written in Java. Jena [28], a popular open source toolkit for developing semantic web applications in Java, is used to manipulate aspects of the ontology, reasoning engine and rules. Jena is ideal for the prototype because it provides programmers with a framework to process and write RDF in many different formats. It also provides an ontology API to handle OWL and RDF Schema (RDFS) ontologies. To implement rules it provides a rule-based inference engine for reasoning with RDF and OWL data sources. To query the knowledge base it provides a query engine compliant with the latest SPARQL specification. Therefore Jena provides a platform to write programs in Java that can perform all the necessary semantic operations and has already been extensively used in a variety of semantic web applications.

#### 3.2.2 Architecture of Jena

To be able to effectively model information using Jena, one first needs to be able to understand the architecture. Jena primarily stores information as RDF triples and allows for code to be written to manipulate that information. An RDF dataset (a graph) can be viewed as a set of edges acting as the predicate linking the subject and object of an
RDF triple. There are three main APIs that the application code interacts with, namely the RDF API, the ontology API, and the SPARQL API. Figure 1 summarizes the interactions between the three APIs as well as the API that is in the architectural level below them, the Inference API.

![Figure 1. Jena Architecture Overview [41]](image)

### 3.2.2.1 RDF API

The RDF API [37] provides access RDF triples, graphs and their various components. Different components in this API are `Resources`, representing RDF resources, `Literals`, representing data values, `Statements`, representing the entire RDF triple and `Models`, representing whole graphs. The `Model` class is used as the primary container of RDF information contained in graph form. The RDF graph is stored in an abstraction called `Graph`, which is easier to manage than a `Model`. With this API
entire ontologies can be added from external sources and also outputted in various forms (RDF, N-triple, RDF-XML, etc.). Functions that can be performed through this API include adding or removing triples from graphs, and finding triples that match a particular pattern.

As noted above, the Statement Java class is used to represent a single RDF triple. Each Statement contains three parts that make up the RDF triple: the subject, the predicate, and the object. The subject of the Statement can only be represented by a Resource, but the object of the triple can be a Resource or a Literal, while the predicate will be a Property according to the RDF specification. Jena provides methods to search for the subject, object or predicate of an RDF triple. The RDF Model is represented as a set of Statements. The Model does not include duplicate Statements.

3.2.2.2 Ontology API

The ontology API [43] can be used to create and manipulate an RDFS ontology or an OWL ontology. Since Jena is fundamentally an RDF platform, the available ontology support is limited to ontologies that are built on top of RDF. Jena aims to provide a consistent programming environment regardless of which ontology language is being used.

The RDFS ontology language [49] is a semantic extension of RDF that is a less expressive ontology language than OWL. RDFS can be used to build simple hierarchies of concepts and properties. Properties in RDFS are the relationships between the subject and the object in an RDF triple. The basic constructs of RDFS are Class corresponding
to entities, Property corresponding to relations or attributes and Container corresponding to complex or structured values. Special meaning is given to certain properties, such as rdfs:subClassOf and rdfs:subPropertyOf, rdfs:domain and rdfs:range, as well as several other properties.

Properties can be associated with each of the classes in a RDFS ontology. There also may be instances associated with each class. In general, the class hierarchies in RDFS are contained in a graph rather than a tree (can contain cycles of properties). Class hierarchies can be used to fill in missing information about individuals in the ontology. For example, rdfs:domain and rdfs:range can be used to state the domain and range of a class, which can then be used by an RDFS reasoner to infer additional details about individuals.

OWL ontologies offer much more expressivity than RDFS ontologies. OWL adds several features beyond RDFS’ capabilities to describe hierarchies. Classes in OWL can be described in much more complex ways. For example, it can be indicated that two classes are disjoint (no individual can belong to both classes), and can also describe the intersection of two classes. Also in an OWL ontology properties can be denoted as transitive, symmetric, functional, or the inverse of another property. There are three subdivisions of the OWL language (from the least expressive to the most expressive): OWL Lite, OWL DL, and OWL Full. OWL Lite is used primarily for describing a classification hierarchy and simple constraints. For example, it supports cardinality constraints but only allows the value of cardinality to be 0 or 1. Because of its simplicity OWL Lite has a lower formal complexity than OWL DL or OWL Full. OWL DL is much more expressive than OWL Lite in that it includes all OWL language constructs, but they are
used under certain restrictions so that computational completeness is retained. Finally, OWL Full is just as expressive as OWL DL but without restriction, so there is maximum expressiveness with no computational guarantees. Variations of each of these ontology languages can be used in Jena.

The Ontology API is language neutral, that is, the class names do not depend on the language that is being used. To specify which language is being used each ontology language has a profile. This profile is bound to an ontology model (OntModel), which is an extended version of Jena’s Model class that provides capabilities for handling ontologies. In the declaration of the Ontology model the programmer can pass the URI of the ontology language. An OWL class or RDFS class are both represented by the OntClass Java class. Relationships between classes, such as “subclass of” or “instance of” relationships, are represented by the OntProperty Java class. All the state information of an ontology in Jena is represented in RDF triples which are accessed by the Statements class.

3.2.2.3 SPARQL API

The SPARQL API [32] can be used to query the RDF triples in an RDF graph. Jena uses the ARQ query engine, which supports the SPARQL RDF query language. SPARQL Protocol and RDF Query Language (SPARQL) is the W3C standard language for querying RDF data. SPARQL is one of the key technologies of the semantic web. There are several query variations for different purposes. One can query for triples, construct new triples, query for a true/false result and query for information about the actual RDF graph. SPARQL queries are performed by matching the triple patterns in the
SPARQL query with triples in the RDF graph. SPARQL is data oriented and only queries the information already held in the graph. SPARQL does not provide any kind of inferencing. The SPARQL API conforms to the published standards and in each revision of Jena the API is updated to correspond with any new SPARQL standards that there may be. To use this API programmers write SPARQL queries in the same manner that they would if they were querying RDF in any other setting.

### 3.2.3 Rule Languages and Reasoning with Jena

The three APIs described above each interact with the API that is in the architectural level below them, the Inference API [50]. One of the key features about semantic web applications is that the semantic rules of RDF, RDFS and OWL can be used to infer additional information that is not explicitly contained in the graph. For example, if B is a subclass of A and C is a subclass of B then by implication C is also a subclass of A. Jena’s inference API can be used to add these implied triples to the graph.

Jena includes a built-in rule-based inference engine to perform reasoning on OWL and RDFS ontologies. There are also a range of external reasoners that can be plugged into Jena, such as a description logic engine, to provide the same reasoning but by different reasoning algorithms. Jena’s generic rule-based reasoner can be used for many RDF processing tasks. There are two internal rule engines in the general-purpose reasoner, one forward chaining engine which uses the RETE algorithm [12] and one tabled datalog engine. Rules are written for the general-purpose rule reasoner by creating a Java Rule object with a list of body terms (premises) and a list of head terms (conclusions). That is, rules are written in an If-Then construct.
Figure 2 illustrates the overall structure of how inferencing works in Jena. The ModelFactory class is normally used by applications to associate a dataset with some reasoner to form a new Model. Queries on this new model will include all RDF statements from the previous model and also include the additional RDF statements that were derived by using the reasoner. Reasoners can be linked via the Ontology API to the appropriate ontology model OntModel. To access the underlying inference graph the RDF API provides an InfModel, an extension to the normal Model interface. Once a reasoner has been created and configured via the ReasonerRegistry it can then be attached to the InfModel. However, the original model can still be accessed by invoking the getRawModel method on the InfModel. The contents of the InfModel can still be added to and removed from by the normal add and remove class to the InfModel. If any of these operations are performed this will cause the current deductions and temporary rules to be discarded and for the reasoning to start again.

Figure 2. Structure of Inferencing Model in Jena [42]
3.3 Sesame

Jena is primarily used for performing inferencing on the ontology. To store data in the knowledge base a Sesame repository is used. Sesame [5] is an architecture for efficient storage and querying of RDF and RDFS. Sesame uses Data Base Management Systems (DBMSs) for a scalable repository. There are a number of DBMSs that have been developed, which makes it is impossible to know which DBMS should be used for which application domain; thus all the architectural code specific to DBMS is part of an architectural level of Sesame, the Storage and Inference Layer (SAIL). The SAIL API translates RDF-specific methods to calls to its specific DBMS. By introducing an additional layer it makes it possible to implement Sesame on a variety of repositories without having to change any of Sesame’s other components. SAIL is considered the bottom layer of Sesame that abstracts from the storage format used. The layers above the SAIL layer include the Repository API and the Graph API, and Sesame’s Access APIs. The Repository API allows one to query, store or extract RDF files; that is, it provides high-level access to the repositories. The Graph API provides a lower level of access to the actual RDF data. Figure 3 summarizes Sesame’s architectural layers.

3.3.1 The Repository API

The Repository API [34] can be used to query and update data in a local or remote repository. A repository is represented by a SesameRepository object. This type of object is used to create a local repository for a program. There are three types of repositories that data can be stored in: a memory store (a memory based RDF repository), a native store (a repository that uses on-disk data structures) or a RDBMS store (a RDF
repository that stores the data in a relational database). An in-memory inferencing or non-inferencing repository can be created by creating a repository object that contains a Boolean parameter “inferencing” which indicates whether the repository should perform RDFS inferencing. This kind of in-memory repository is volatile; that is, its contents will be lost when the object is garbage collected or the program is shut down. On the other hand, a native store can be used to save the RDF store contents into a file. On creation of this kind of repository a particular file should be specified for persistence. There are many ways to fine-tune the configurations of a repository by using the SailConfig and RepositoryConfig classes. It is also possible through this API to connect to a

![Sesame architecture](image)

*Figure 3. Sesame architecture [51]*
remote repository.

Once a connection to the repository has been established the SesameRepository can then be queried. Queries can be constructed by creating a String object containing the query. This String is then passed to a TupleQuery object. The execution of the query will return a TupleQueryResult object. A BindingSet object can then be used to retrieve the different values of the variables in the query. RDF data can be added to the repository in the form of a file, a location on the web, or a Java String object. However, individual RDF statements cannot be added through this API; that task is performed in the Graph API.

3.3.2 The Graph API

The Graph API [34] can be used to represent an RDF graph in the form of a Java object. The main interface for a graph in this API is org.openrdf.model.Graph. This object offers a way to handle RDF graphs from the Java code. Once a graph object has been created statements can be added to and removed from it. To add statements to the graph, one first needs to create a subject (URI object), predicate (URI object), and an object (URI or Literal object). This can be done by obtaining a ValueFactory object from the Graph and using that object to create the corresponding URI objects and Literal objects. Once the subject, predicate and object have been created they are added in the form of a triple to the repository. Full graphs can be added and removed from the repository.

Graph queries can be used to create a graph that contains statements in a local Sesame repository. Graph queries in addition to the Graph API can be useful in updating
a large number of statements in the repository. For example, queries can be used to derive the new statements that the old statements should be replaced by and add these new statements to the repository. A second query can then be used to select all the old statements that are no longer needed and remove them from the repository.

3.3.3 The SAIL API

The SAIL API [33] defines a set of Java classes and methods for storing and performing inferencing on RDFS. The main interface of the API is the Sail interface. Other interfaces that the Sail interface creates are StatementIterator, NamespaceIterator, and Transaction. There is also a Query interface that the Sail interface evaluates. A SailRepository is the class most commonly used when accessing a local Sesame repository [35]. They operate on Sail objects for storing and retrieving RDF data. The Sail that a repository operates on determines the behavior of the repository. The constructor of the SailRepository class accepts any kind of object of type Sail. Examples of Sail objects that can be passed to the SailRepository are MemoryStore, NativeStore, and RdbmsStore. The advantage of using Sail objects is that they can be stacked on one another. For example, the Sail object ForwardChainingRDFSInferencer can be stacked on top of a MemoryStore object. This means that, RDFS inferencing can be used on an RDF stored in main memory.

In conclusion, Chapter 3 has summarized the rule languages and the semantic tools that are relevant to this thesis. It introduced the Jena, a semantic Java toolkit. In the following chapter it will be describe how Jena is used in this project to construct general
purpose rules and perform the reasoning. Chapter 3 also gave a detailed description of the Sesame framework used in this project. In the following chapter the type of Sesame store that was used to store the knowledge base will be discussed. The following chapter will also describe the functionality of the matchmaking system.
Chapter 4

Architecture Design

This chapter discusses in detail the architecture design of the online dating model. In Section 4.1 the matching system that is implemented in this project is discussed. Next, in Section 4.2 a brief introduction to the functionality of the match making system that was extended is given. In Section 4.3 research on potential ontology languages is presented, and the design decisions that are made in forming the ontology and rules are discussed. In the subsequent sections the ontology (Section 4.4), knowledge base (Section 4.5) and rules (Section 4.6) used in this prototype are described.

4.1 Proposed Matching System

The proposed matching system uses a level based architecture. There are four levels in total. Level 1 of the matching system matches a user based on their basic information (gender, looking for, etc.). Any other users who do not meet the level 1 requirements are eliminated from the list of potential matches. The results from level 1 are used in each of the other level as can be seen in Figure 4. Next, a check is performed to see how well users match on their level 2 information, information about a user’s priorities. Users can select which priorities they weigh most importantly and a score is calculated for each other member who matched the user in level 1. This score is further increased if users match on answers to the questions in level 3, which deal with psychological aspects of relationships. When users have been compared on each level their scores from each level are accumulated.
Level 4 is the main level that has been developed in this project. The fourth level matches users based on additional facts that have been inferred about the user. Here, data is inferred based on a user’s geographical area. From geographic data the following things can be inferred about the user: income level, primary language, generation status, mobility (how frequently the user changes residence) in the past year, mobility in the last five years, and level of education. To gain this information about each Canadian city Statistics Canada’s 2006 Census of Population [52] datasets are used. Similarly, datasets from the U. S. Census Bureau, 2011 American Community Survey [31] are used to get data about certain American cities. Any properties associated with a city are also associated with users from that city. For example, if a city has a certain average income
level, then that income level is associated with every user from that city. Once users have answered level 1, level 2, and level 3 questions they will then have the opportunity to answer questions regarding things that have been inferred about them based on where they are from. If a user’s answer is different than what has been inferred about them then the inferred fact is removed and their answer added. However, until a user answers these questions the inferred answers will be used in the matching and users will be given a higher score, and therefore rank higher in the list of potential matches, if they match on these properties.

4.1.1 Statistics Canada and U.S. Census Data Integration

The advantage of using semantic web techniques to model the matching system is that new information can be easily integrated. Entire data sets can be added to the RDF set as long as that information is also in an RDF format. Therefore the knowledge base may contain not only information about users but also external information such as characteristics about the places they are from, beliefs about their astrology sign, or any other information about characteristics that they have provided in their profile. By creating links between datasets, data that wasn’t previously linked can be connected, and thus new pieces of information can be exposed.

One of the main objectives of this project is to provide an inferred level of matching based on external sources of data. A readily available dataset is statistical information about cities in Canada and the United States. This information was provided by Statistics Canada for Canadian cities and by the U.S. Census for cities in the United States.
Both data sources provide their datasets in comma separated value (CSV) format and both provide a large amount of information on age characteristics, marital status characteristics, family characteristics, immigration status, and several other categories. Although not all of this information would be used by the prototype system all of the information was transformed to be in a uniform format so that if it was needed it could be accessed in the future. Since user information is first stored in a MySQL database, a Java program was written to transform each CSV file into a MySQL table in a database specified for geographical information. Each city was given its own table in the MySQL database. Each characteristic about a city was given a unique id and that id, characteristic and its specified value occupied a row in the table. The structure of a city’s table is summarized in Table 1.

Each Canadian and U.S. city in the dataset has the same table format. The information is not immediately transformed into RDF since not all the data will be relevant to the matching system. Since the knowledge base loading and querying times are very important, storing and processing mass amounts of RDF that will never be used is problematic.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Null</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>int(11)</td>
<td>NO</td>
<td>PRI</td>
</tr>
<tr>
<td>topic</td>
<td>varchar(255)</td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td>total</td>
<td>int(11)</td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td>male</td>
<td>int(11)</td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td>female</td>
<td>int(11)</td>
<td>YES</td>
<td></td>
</tr>
</tbody>
</table>

Table 1. The structure of a city's table
The characteristics that should be used from the datasets are the characteristics that firstly, are included in both data sets, and secondly the characteristics that are relevant to a matching. In the match making prototype a Java program is developed that uses an SQL query to return each characteristic that should be added to the knowledge base. For each statement that is added to the knowledge base the subject of that statement is the city name, and the property of that statement is a named property assigned based on the characteristic in the “topic” column. The named property also specifies whether the characteristic refers to the total population, male population or female population. In most cases the value for the object of the statement must be calculated using several rows of the MySQL table. For example, a person’s primary language is inferred using the rows provided in Table 2 regarding the language used at home the most in Fredericton.

<table>
<thead>
<tr>
<th>topic</th>
<th>total</th>
<th>male</th>
<th>female</th>
</tr>
</thead>
<tbody>
<tr>
<td>language_spoken_at_home_total_pop</td>
<td>49980</td>
<td>23580</td>
<td>26400</td>
</tr>
<tr>
<td>language_spoken_at_home_english</td>
<td>46010</td>
<td>21715</td>
<td>24295</td>
</tr>
<tr>
<td>language_spoken_at_home_french</td>
<td>1850</td>
<td>865</td>
<td>985</td>
</tr>
<tr>
<td>language_spoken_at_home_nonofficial</td>
<td>1765</td>
<td>840</td>
<td>925</td>
</tr>
</tbody>
</table>

Table 2. Statistics for language used the most at home in Fredericton

Data regarding the total male population of Fredericton and the number of males from Fredericton whose language spoken at home is English can be used to find that 92% of males from Fredericton speak English. And thus the corresponding RDF triple has the form: Fredericton_NB, inferred_language_male, acu:English. By producing these RDF triples for every city there is now new information in the
knowledge base that can be linked to users using the cities they are from.

4.2 Building on the Matching Implementation in the Previous Prototype

The previous match making system that is extended in this project semantically organizes user input in a general way. The previous prototype organizes properties from user profiles into RDF triples. Specific SPARQL queries are used to find a user’s list of appropriate matches, based on the information they have specified in their profile.

4.2.1 Creating User RDF

Every user in the dating system has various information associated with their dating profile (e.g. age, gender, height, etc.). Each user also has a unique ID associated with their profile. This ID is used as the subject of the RDF triples corresponding to that user. For example, if the user with ID 001 is a 48 year old woman that will be referred to as “Person1” from Whitehorse, Yukon seeking a man who doesn’t smoke or drink, then she is represented by the following RDF triples (a full RDF profile demonstrating all of a user’s possible level 1 and level 2 properties can be found in Appendix A):

\[
\text{acu:001}-\text{firstname}=\text{lastname} \quad \text{acu:first_name} \quad \text{“Person1’s first name”}.
\]
\[
\text{acu:001}-\text{firstname}=\text{lastname} \quad \text{acu:last_name} \quad \text{“Person1’s last name”}.
\]
\[
\text{acu:001}-\text{firstname}=\text{lastname} \quad \text{acu:age} \quad \text{“48”}^{\text{^^xs:int}}.
\]
\[
\text{acu:001}-\text{firstname}=\text{lastname} \quad \text{acu:city_code} \quad \text{acu:whitehorse_YT}. \quad \text{.}
\]
\[
\text{acu:001}-\text{firstname}=\text{lastname} \quad \text{acu:sex} \quad \text{“woman”}. \quad \text{.}
\]
\[
\text{acu:001}-\text{firstname}=\text{lastname} \quad \text{acu:sex_pref} \quad \text{“man”}. \quad \text{.}
\]
\[
\text{acu:001}-\text{firstname}=\text{lastname} \quad \text{acu:pref_drinking_habits} \quad \text{acu:no}. \quad \text{.}
\]
\[
\text{acu:001}-\text{firstname}=\text{lastname} \quad \text{acu:pref_smoking_habits} \quad \text{acu:no}. \quad \text{.}
\]
The subject of each triple about the user uses their ID, first name, and last name to identify that the triple pertains to that particular user. For example, if Person1’s name was Jane Doe, the subject of their triples would be “acu:001-Jane-Doe”.

All information about each city is also modelled in RDF. Each city has a unique city code that is used as the subject of the RDF triple. The property of each triple is a statistical property about the city (e.g. total income, male income, female income, etc.), and the object of the triple is the corresponding piece of data. A complete example of the RDF corresponding to Fredericton, New Brunswick can be found in Appendix A.

4.2.2 SPARQL Queries

To match users in the dating system SPARQL queries are used to run over the RDF. When a user requests their matches several SPARQL queries are made to filter potential matches and to improve a match’s score. SPARQL queries work by matching the triple pattern in the WHERE clause against the triples in the RDF graph. The subject, object and/or predicate of the triple can be left as a variable to be matched. For example, the following query is used to filter for users that meet Person1’s drinking and smoking preferences:

```
1 SELECT ?y
2 WHERE {
3   ?y rdf:type acu:Person
4   FILTER(acu:001-fname-lname!= ?y)
5   acu:001-fname-lname acu:pref_drinking_habits ?dh
6   ?y acu:drinking_habits ?dh
7   acu:001-fname-lname acu:pref_smoking_habits ?sh
8   ?y acu:smoking_habits ?sh
9 }
```

The first line is used to specify that the query should return all values of the variable y.
This variable will represent all the users in the system who have the drinking and smoking habits that Person1 is looking for. The query begins at the WHERE statement. Line 3 filters for any subjects of triples that have type Person. This will be all of the users in the dating system. Line 4 filters to remove Person1’s own profile from the list of returned matches. Next, line 5 is used to retrieve Person1’s preferred drinking habits, and line 6 finds all users whose specified drinking habits that match Person1’s preferences. Similarly, line 7 is used to find Person1’s preferred smoking habits. In line 8, the list of users found in line 6 is used to find the users whose smoking habits are that which Person1 prefers. Therefore the result is a list y of users who meet both of Person1’s preferences.

It can be seen through this example how well SPARQL queries can be used to filter for a user’s matches. Level 1 uses one large SPARQL query to filter for all potential partners who meet a user’s preferences. This ensures that if users differ on even one level 1 property they will not be included in each other’s list of potential matches. In levels 2, 3 and 4 several SPARQL queries are used to find potential matches. For example, in level 4 if two users match on level 1 properties and they match on inferred language, but not inferred income then those users will still be returned as potential partners, however the score of that match will be less than if the users had matched on both properties. The method for calculating a match’s score will be described in Section 5.1.

4.3 Ontology Languages

Ontologies represent real world entities in a systematic way. By providing consistent definitions to real world entities, ontologies serve as a reference model for
their domain. Therefore by organizing the dating model into an ontology the comprehension and knowledge sharing in this prototype can be increased. One key attribute that had to be evaluated before designing the ontology was which ontology language should be used. The primary purpose of the ontology is to classify user and geographical information in terms of its meaning in an online dating context. The Jena API is one of the main toolkits used in this prototype; therefore the ontology can be modeled in either RDFS or OWL.

4.3.1 Features of RDFS and OWL

Both RDFS and OWL ontology languages were evaluated to determine which one would serve the purposes of this prototype best. To determine which ontology language will best meet the prototype requirements, the ontology requirements must first be formally defined. The main requirements of an ontology to model online dating are the following:

- Allow the attributes associated with each question to be meaningfully classified into resources of which the corresponding options can be associated.
- Allow cities to be represented by meaningful classes.
- Allow attributes about cities to be associated with the corresponding city class.
- Allow domain and range of a property to be specified.
- Allow rules to be used by a reasoner to infer additional data.
- Be scalable in that performance does not degrade as the size of the dataset grows.

On the basic level the ontology should be able to classify user attributes and
options for those attributes, as well as cities and properties about those cities. Classes and properties can be defined using either language. Also, domain and range properties can be specified in either language. These features suggest that both RDFS and OWL meet the first four properties. However, as noted in Chapter 3, OWL provides more features to define hierarchies among properties and classes [10], so it is worth examining those options to see if they can be applied to the online dating model.

For properties in an ontology, OWL allows one to specify transitive, symmetric, functional, and inverse properties [48]. Transitive properties are such that if a pair (x,y) is an instance of a transitive property P, and (y,z) is also an instance of P, then by implication the pair (x,z) is also an instance of P. In the online dating model properties that link the user to an attribute are not hierarchal and therefore transitivity is not necessary. A symmetric property has no real meaning in an online dating model. In the knowledge base, most properties will associate a user with an answer to that property, or a city to a statistic about that city. For example, the property religion will associate a user with what religion they describe themselves as being, to have that property be symmetric, or any property in the model does not make sense. Being able to define functional properties could be of some use in an online dating model. A functional property can have only one value y for each instance x. For some preference properties users will be allowed to choose more than one option (e.g. users can request matches who practice Christianity and Catholicism), but most properties about a user’s characteristic will be functional. Each property about a city will be functional, e.g. a city can only have one value for average income. However, by ensuring that the software for retrieving information from users is implemented correctly it can be assured that these properties
only have one value for each instance, and thus there is no need to specify that a property is functional. Finally, a property has a direction from domain to range. The inverse of the property has the opposite direction. It can easily be seen that an online dating model will not need to define the inverse of properties since the model is primarily relating a user to an attribute via a property; therefore there is no need for a property relating a general attribute to a specific user.

For classes in an ontology, OWL allows the specification of complex class descriptions [10]. Class definitions in OWL can be represented in description logic syntax. Complex class descriptions can involve defining classes in terms of union or intersection with other classes. In an online dating model each class describes some attribute associated with a user or a city. The union of two classes and the complement of a class do not provide any useful information in the online dating model. The intersection of each of these classes will always be empty. As a consequence each class can be classified as disjoint. OWL can be used to classify classes as disjoint but because every class will be disjoint that need not be specified. OWL also allows can be used to express explicit equality or inequality relations between individuals by means of the owl:sameAs and owl:differentFrom properties. In the online dating model options are presented to the user (the user may not enter their own options), thus the option to define equality or inequality between individuals is not needed because each distinct individual in the model has been clearly labeled.

Enumeration and property restrictions are also available features in OWL. Enumeration is a list of individuals that are instances of a class. Again, all instances are predefined for the online dating model in this prototype. There are two kinds of property
restrictions in OWL: value constraints, which puts constraints on the range of a property, and cardinality constraints, which puts constraints on the number of values a property can take. Value constraints are not necessary in the ontology because if the range of a property must be restricted then only the options in that range are presented to the user. Cardinality constraints are not necessary in the ontology because if a user can choose more than one option for a property, then they can choose as many options as they want and there are no constraints on that value.

The property and class features listed above are the main features that OWL supports and RDFS does not. Since both RDFS and OWL meet the basic requirements, and OWL’s expressivity offers no additional benefits in the online dating model, the two main points that need evaluation between RDFS and OWL are their rules and reasoning capabilities and scalability.

4.3.2 Reasoning and Scalability

The number of users in the knowledge base will grow; therefore there is a need for a suitable reasoning engine that can cope with performance along with scalability. It has been found that the complexity of reasoning with OWL ontologies is a high upper bound [2]. However, in this application the full expressivity of OWL is not used; only fragments of OWL that are shallow in logical terms. For example, as noted in Section 4.2.1, features that OWL offers such as UnionOf, ComplementOf, IntersectionOf, and enumeration are not needed in this application because that complexity has been put on the system for retrieving information from users, which ensures that these features are not needed. It has been designed in such a way that the intersection of classes will always be
empty, and the union of, and complement of do not provide any useful information. Each of these features is offered by the more expressive OWL DL, and therefore the least expressive version of OWL, OWL Lite, would suffice to express the class and subclass relations used in the ontology. Reasoners for OWL ontologies may be optimized to a larger number of features, in particular those that are not actually used in the ontology. However, there are a number of different reasoners available that take into account different design decisions in addressing complexity and expressiveness versus scalability. In the online dating model scalability is the main concern and complexity and expressiveness are less of a concern.

In [14] Guo et al. construct a benchmark for semantic web based systems that require the processing of megabytes of data. Their benchmarking approach compares three different classes of systems: systems supporting RDFS reasoning, systems providing partial OWL Lite reasoning, and systems that are complete for OWL Lite. They test several knowledge base systems since loading times do not only depend on the ontology language chosen but also the way the knowledge base is stored. For RDFS reasoning they use a memory-based implementation of Sesame (Sesame-Memory) and a database-based implementation of Sesame. For partial OWL Lite they use DLDB-OWL [25], which supports processing and queries on OWL documents and for complete OWL Lite reasoning they use OWLJessKB [20], a description logic reasoner for OWL. They test using a desktop computer with a 1.80GHz Pentium 4 CPU, 256MB of RAM, 80GB of hard disk, Windows XP Professional OS and Java SDK 1.4.1.

Their findings are a good indication of scalability between the different architectures. They find that in loading 103,397 triples Sesame-DB and DLDB-OWL
performed about the same, having loading times of 5:43 minutes and 9:02 minutes respectively, while Sesame-Memory completed in just 13 seconds. OWLJessKB took much longer; it took 3:16:12 hours to load. As the number of triples increases OWLJessKB was unable to load in a measurable time (it ran more than a number of days), while Sesame-Memory continued to load much faster than Sesame-DB and DLDB-OWL. At 1,316,993 triples Sesame-Memory loaded in just 5:40 minutes, DLDB-OWL loaded next in 1:54:41 hours, and Sesame-DB loaded in 12:27:50 hours. However, at 2,782,419 triples Sesame-Memory succumbs to memory limitations, whereas DLDB-OWL loads in 4:22:53 hours and Sesame-DB loads in 46:35:53 hours. It is clear that partial OWL Lite scales the best. However, until Sesame-Memory succumbed to a memory limitation it was loading triples in 5% of the time it took DLDB-OWL to load. Also, in this benchmark they speculate that one of the reasons for the difference in performance between Sesame-DB and DLDB-OWL is related to the way Sesame performs inferencing. It is ideal that the prototype achieve the loading times of Sesame-Memory, so by adding more memory, more triples may be able to be processed. Or by using Sesame to store data without inferencing and Jena to perform inferencing (as will be described in Section 4.4) Sesame-DB can potentially achieve faster load times. For these reasons RDFS has been chosen as the language used to develop the ontology.

4.4 Ontology Design

One of the main objectives of this project is to extend the existing ontology that structurally organizes user input. Concepts (classes) of the online dating subdomains are represented in an RDFS ontology, adapted from the ontology proposed in the previous
prototype discussed earlier (Section 1.3). This ontology is used to structure the profiles of users in an online dating model. The ontology is produced using a previously developed (in the previous prototype) OpenL tablets \[47\] wrapper. That is, the information that will form part of the ontology (classes, properties, static instances, and rules used in SPARQL queries) is described in an MS Excel file, which is used by the OpenL tablets wrapper. By using an Excel spreadsheet, the type of information contained in the ontology can be easily changed as the dating domain expands and changes. This information is then converted into the appropriate ontology format and written out into a text file.

Another objective of this project was to design an ontology to represent geographical information. This ontology is used to model the information pulled from Statistics Canada and the U.S. Census in a uniform way. This ontology shares some common properties with the dating ontology. Instead of using an existing ontology in the area of geography, an ontology was created because of the need for classes corresponding exactly to the information derived from the retrieved data. This ontology is also written in RDFS for easy integration.

### 4.4.1 Previous Ontology

The previous ontology that has been built upon describes a user’s dating profile and reflects the user’s preferences as well as attributes. Every user using the online dating system will answer questions and in doing so associate attributes with themselves or preferences about what they are seeking in a partner. The ontology also describes information about questions that are asked to the user. That is, each question pertains to an attribute or preference of a user and has several options that a user may choose. In the
previous RDFS ontology a user is represented as a Person resource. Each subject of a question is represented as its own class (e.g. religion, body type, education, ethnicity, etc.). Each of the options for a subject is represented as an individual and associated with the corresponding subject class by the rdfs:type relation. Each attribute about the user is represented in the ontology by a property that links the Person resource to the Subject resource. Figure 5 illustrates a portion of the schema of the previously developed ontology. Specifically, it shows the resources associated with the religion attribute. The rdfs:domain property relates the Person class to the religion property. The rdfs:range property associates the religion property that represents a user’s religion, with the Religion class which contains all possible religions that a user may pick from (Note this list has been shortened in Figure 5).
4.4.2 Statistics Canada and U.S. Census Ontology

When registering, each user specifies the city, province or state and country that they are from. Each city will be represented as its own class in the ontology called City. When naming the City classes, the city’s province abbreviation is appended to the end of the class name for all cities. This is done to distinguish between cities with the same name in different provinces. For example, “Windsor, Nova Scotia” is represented by the class “windsor_NS” and “Windsor, Ontario” is represented by the class “windsor_ON”. Each class representing a city is of type City. A Person is linked to the City class by a city_code property. Each city has properties associated with it based on the data that has been pulled from Statistics Canada and the U.S. Census (e.g. average income of total population, female average income, male average income, etc.). Each property about a city in the data is represented as a property that connects the city and a literal value. Figure 6 illustrates a portion of this ontology pertaining to the income property. The rdfs:domain property relates the Person class to the city_code property. The rdfs:domain is also used to associate the income properties (e.g. income_male) with the City for which that income is associated. The rdfs:range property associates the city_code property that represents a user’s city code with the City class which contains all possible cities that a user may be from. The rdfs:range property is also used to associate the income properties with the nonNegativeInteger class. This means that the income property can only have values which are nonnegative integers.

4.4.3 Level 4 Ontology

In level 4, users are asked questions corresponding to the attributes that have been
inferred about them. For example, first rules and reasoning are used to infer a person’s language from the data that has been pulled from Statistics Canada and the U.S. Census. Until a user answers the question corresponding to their language the inferred language is used in the matching process. Once a user specifies which is their primary language, their inferred language is then replaced with that answer, and the new answer is then used in the matching.

Figure 6. Ontology structure associated with income property

The ontology structure in this level is the same as in the lower levels. That is, for the subject of each question (e.g. income, moved in the last year, etc.) that subject is represented by a property associated with the user and all the possible options of that property by classes. For some questions the options to those questions are related to the subject of the question class by the rdfs:type attribute. However, for some of the
subjects, e.g. income, there will be no options and those properties will associate a Person by some kind of literal value. Figure 7 shows a portion of the schema that describes these level 4 properties. Specifically it shows the structure associated with the language attribute. In this example, the rdfs:domain property relates the Person class to the language property. The rdfs:range property associates the language property which represents a user’s actual language (the one that they have chosen, not the one that has been inferred) with the Language class which contains all possible languages that a user may pick from.

Figure 7. Ontology structure associated with level 4

4.5 Knowledge Base Design

The developed prototype system works in several steps that have been summarized in Figure 8. First, the RDFS ontologies are generated to represent user, and geographical
profiles. Next, data is retrieved from the HTTP end point of the dating website about the user and stored in a MySQL database, and is then transformed to the corresponding RDF. The geographical data is also transformed from SQL to RDF. Next, the RDFS ontology and RDF information are stored in a knowledge base using a Sesame native store. A reasoner is then executed over this information and additional RDF triples are inferred and stored in the knowledge base along with the original RDF triples. Next, the information in the knowledge base is used to match users as matches are requested. The process is repeated as users update their information and new users join the dating website.

![Figure 8. Steps performed in system prototype](image)

4.5.1 Knowledge Base Creation

One of the main functions of the Java prototype is ontology creation. Based on the information that has been provided in the Excel spreadsheet an ontology is created and
written to a text file. The RDFS is written in the N-Triples format [46]. Another one of the main functions of the Java prototype is the creation of RDF data based on user provided information and geographical information. User and geographical information is initially stored in two separate MySQL databases. The Java program developed retrieves this information from the MySQL databases and transforms it into RDF. This is done by assigning a specified attribute name to the column in the MySQL table, which acts as the property of the RDF triple. The subject of the triple is a user’s identification string, and the object of the triple is the actual values of the selected attribute. Geographical information is pulled from the database and transformed into RDF in the same way. The RDF representing user and geographical information is also in the N-Triple format.

Once user and geographical information have been transformed into RDF a knowledge base is created to store the data. The number of RDF triples will grow as the number of users joining the dating website grows, therefore the system to store RDF must scale well for this possibly large RDF application. In this dating site application, the knowledge base is scheduled to be updated once a day, which takes approximately 15 minutes to complete. That is, if a user makes a change to their profile, those changes will be reflected in the matching the next time the knowledge base is updated. The knowledge base can be updated manually at any time, therefore a script could be written to update the knowledge base more frequently (during low query demand) if necessary.

The emphasis is on querying the knowledge base, as this is how users receive their matches. When a user requests their matches certain SPARQL queries are run. If a user’s matches have already been calculated then a single SPARQL query is used to retrieve the already pre-computed matches; however, if this is the first time calculating a user’s
matches then several large SPARQL queries must be run, therefore the knowledge base must be able to return results quickly as users request their matches.

4.5.1.1 Choosing how to Store the Knowledge Base

There are several options for RDF stores. In the previous prototype a Sesame store using a PostgreSQL database is used to store the RDF. However another popular option is to use Jena to store the knowledge base. Using Sesame, data can be stored in a memory store (a memory based RDF repository), a native store (a repository that uses an on-disk data structure) or a RDBMS store (a RDF repository that stores the data in a relational database). Jena also allows data to be stored in an in-memory store or one that uses a database.

Scalability is a very important issue in the prototype. The previous system’s storage system (a Sesame store using a PostgreSQL database) worked well for their application. However in this prototype there have been large numbers of RDF triples relating to statistical information added to the knowledge base, therefore there is no guarantee that the same Sesame repository will suffice. For this reason, other popular RDF stores were examined. In [22] Lui and Hu evaluate 7 popular RDF storage systems, 3 of which are Sesame stores (an in-memory store, native store, and RDBMS store) as well as 2 Jena stores (in-memory and RDBMS store), with respect to large data applications. They evaluate storage systems in three areas: data processing times, querying facilities and inference support. They find that the in-memory stores load data the most quickly. However, an in-memory store would not suffice for this application because it is erased each time the session ends, and the program is not constantly running, but runs once to
initialize the knowledge base and once each time a match is requested. It is possible to write the data contained in the knowledge base to disk so that when the memory store is re-initialized the information from the file is read back into the store. However, in-memory stores may not scale well because of possible limitations on the size of available memory [6], which in turn does not leave much space for algorithms to operate on larger knowledge bases [17].

In contrast to the in-memory store results, they find that the results of the Sesame database and Jena database approaches show bad scalability in data loading. They attribute the Sesame database’s poor performance to the fact that the Sesame database employs a unique ID to the RDF node mapping system. They find that the Sesame native store and other native stores perform well in terms of loading data. Their loading times grow linearly as the dataset size increases. Based on this benchmark a Sesame native store would work best for loading data in this application. However, the emphasis of this application is on querying the knowledge base, therefore this metric must be examined.

In the benchmark described above they do not query the knowledge base using SPARQL but instead use another query language. Since the SPARQL query language will be used in this prototype the query results from this benchmark are not reported. In [1] they examine a Sesame native store (Sesame), a Jena database store using a TDB optimizer configured to use the statistics based optimization strategy (Jena TDB), a Jena database store using MySQL (Jena SDB) and two different Virtuoso stores. They measure query performance by running 500 query mixes against the stores. They find that the Virtuoso triple store shows the best overall performance for the datasets with 25 and 100 million triples. However, for the dataset with 1 million triples, Sesame out performs
Virtuoso. For 1 million triples Sesame completed almost 2 times as many queries in an hour as Jena SDB did, and completed approximately 4.5 times as many queries as Jena TDB did. They find that Sesame out performs Jena SDB and Jena TDB on queries on 1 million, 25 million and 100 million triples. Thus the Sesame native store is a better store for this application than any of the Jena stores.

### 4.5.1.2 Native Store vs. RDBMS Store

Since the knowledge base will be queried frequently, a store that has shown to have good query performance is implemented. In this prototype both a Sesame native store and a Sesame RDBMS store using PostgreSQL were tested. The previous semantic prototype implemented a Sesame RDBMS store using PostgreSQL. Sesame’s Sail API allows one to easily change between a relational database store and a native store, therefore the database store used in the current prototype could be easily interchanged with a native store for testing purposes. The dataset to be stored will contain all ontology, user and geographical related information and therefore it will be time consuming to create the repository and also time consuming to reason over this large repository. Thus the number of times that this needs to be done must be minimized. Therefore the knowledge base is created once and stored so that it can be queried over and over again until there is new information to be stored in which case the knowledge base is recreated.

The RDBMS store interacts with the relational database by using a JDBC driver. To create an RDBMS store a `SailRepository` object is created and passed an `RdbmsStore` object with the corresponding information about the PostgreSQL database (username, password, etc.). To create a Native store a `SailRepository` object is
created and passed a `NativeStore` object that contains the path of the directory to where the store will be saved.

Both approaches were implemented on the server that runs the online dating website. When using the RDBMS store, the call to PostgreSQL to store the data consumed a lot of time and processing resources. It took approximately 2 hours and 15 minutes for the initialization (knowledge base to load, inferencing to be performed, and several initialization queries to be run) to finish. When using the native store the initialization was performed in just 15 minutes. The native store also used less CPU percentage, meaning other processes on the server were not slowed down. Because of its clear benefits a Sesame native store is used to store the knowledge base.

### 4.5.2 Perform Reasoning on the Knowledge Base

As described in the previous section, a native store is used to store the knowledge base. Sesame does provide standard RDFS reasoning support but support for using custom rules and a reasoner is also needed. This is not well documented in Sesame. There is very little documentation on developing rules for reasoning in Sesame. However, the Jena API offers numerous reasoners and reasoning systems. For this reason, Jena is used to produce the reasoner that will reason over the knowledge base.

Once all of the data has been loaded into the knowledge base a query is constructed for all triples in the knowledge base. These are written and saved in a file, that is then read by a Jena `ModelFactory` object. Here, because this model is only being used to perform reasoning, an in-memory store can be used; therefore this store can be written to very quickly. Next, a reasoner object is created and reasoning is performed, which is
described in Section 4.6. The Jena model is then written out and that information gets added to the knowledge base.

4.6 General Purpose Rules

Inference engines in Jena are used to derive additional RDF triples which result from a base RDF graph together with ontology information and rules associated with the reasoner [50]. RDFS allows for additional facts to be inferred from instance data. A general purpose rule-based reasoner is included in Jena which can be used to implement RDFS reasoners. Various engine configurations can be created by using a generic rule reasoner and a specified rule set.

Matches in the dating system are made by querying a set of RDF triples, thus rules are constructed to be used by a reasoner that reasons over the initial RDF and adds new triples based on the new information that is added to the knowledge base in level 4. Jena's general-purpose rule-based reasoner was used in this prototype because it supports rule-based reasoning over the RDF graph and allows one to construct rules for a reasoner that reasons over RDF and adds corresponding new triples.

4.6.1 Rule Syntax and Structure

A Java Rule object is used to define a rule for the rule-based reasoner. Each rule is comprised of premise terms and conclusion terms, as well as an optional name and direction. Each clause is a triple pattern or a call to a built-in primitive. In this prototype, rules have been written in a text file, which gets interpreted by a rule parser, which then passes those rules to the reasoner. A description of the rule syntax used in the prototype is the following:
Forward chaining rules are used by starting with the available data and using inferencing to extract additional data until the goal is reached. That is, if each premise is satisfied then the rule indicates that the conclusion triple should be inferred. Therefore the rule reasoner is configured to use only the forward chaining engine.

Jena aims to keep rules simple by including a set of known prefixes for URI refs such as rdf, rdfs, owl and xsd. Additional prefix mappings can also be defined in the rule file. The following prefix is included on the first line of the rule file (where the-ontology-namespace is the namespace used in the RDFS ontology):

```ruby
@prefix acu: <the-ontology-namespace> .
```

By including prefixes for each of the namespaces being used, the rule file can be read much more easily. Once the list of prefixes has been defined, the rules can be written.

### 4.6.2 Example Rule

A complete set of rules was written to be used in this prototype. The geographical ontology developed in this project contains various data about every Canadian city, and select U.S. cities. There is also a city_code property associated with each user that describes what city that user is from. Using a set of rules the aim is to link information provided by Statistics Canada and the U.S. Census about a city to all the users who are from that city. Statistical data is broken down by gender. That is, population characteristics for the total population, male population and female population are provided. Since the gender of each user is known, gender is also taken into account when
inferring additional facts about that user. For example, if Statistics Canada reports that
the average income of a male in Fredericton, NB is $30,000 then a property called
Inferred_Income should be associated with males from Fredericton with a set value
of 30,000. For each property about a city that is contained in the ontology, a
corresponding property is created for each user from that city. The premise begins with a
term to determine which city a user is from: (?a acu:city_code ?b), where a is
the variable that represents the person’s identification, and b is the variable that describes
the city they are from. Once variables a and b have been given a set of values, a second
premise is then solved that says that the gender of the user must (in this particular case)
be male: (?a acu:sex “man”). This condition will filter all the results from the
first condition. Once these two conditions have been met the income of the city is
retrieved and stored in an additional variable c by the following premise: (?b
acu:income_male ?c). If each of these premises has been met then the conclusion
is examined to determine the next steps. The conclusion is used to link the data that has
been found (?c) to the user (?a), which is done by the following conclusion: (?a
acu:inferred_income ?c). Written in its entirety the rule to infer average income
about a male user looks like:


Thus the triple with property inferred_income is added to the knowledge base when
reasoning occurs. All other rules are written in a similar fashion to this example, where
Inferred_Income is substituted with the specific property that is inferred. Rules are
also included that infer the appropriate values for females.

4.6.3 Creating a reasoner using the rule set

As described in Section 4.3, after the knowledge base has been transformed into a Jena model, an inferencing method is called on that Jena model. A general-purpose rule-based reasoner is created by creating a GenericRuleReasoner instance and passing it the set of rules created. Jena allows one to easily configure the reasoner by using a direct Java method to set a configuration parameter, `Reasoner.setParameter`. This method is used to set the RDFS compliance level. The reasoner can be configured to work at three different compliance levels. When configured to the “Full” mode, all the RDFS axioms and closure rules are implemented. Using this mode is expensive since all statements in the model must be checked for possible use of container membership properties. Also type assertions are generated for all resources and properties. The second mode is the “Default” mode. This mode omits checks for container memberships and is therefore less expensive. This mode does include all axiomatic rules. The third and final mode is the “Simple” mode. In this mode the transitive closure of `subPropertyOf` and `subClassOf` relations, as well as domain and range entailments are implemented. It omits the axiomatic rules.

The “Simple” mode is used for the reasoner since it is the less computationally expensive mode and the most useful mode for this application. Once the reasoner is configured it is attached to the RDF dataset that it will reason over. Doing this creates an inference model, `InfModel`. For some external reasoners a hard separation between schema and instance data is required, but when using a built-in reasoner the separation is
arbitrary. The InfModel object contains the Jena model that includes the original data set as well as all the additional statements that have been inferred.

Jena also provides methods to check the validity of the resulting data by using a ValidityReport object. Validation is used to check whether any constraints have been violated. In this prototype it is primarily the domain and range constraints that may have been violated and if so then there is an inconsistency. An inference model can be tested for inconsistencies by using the InfModel.validate() interface. This interface looks for inconsistencies across the schema and instance data. A ValidityReport is returned that is comprised of a pass or fail flag.

A validity check is implemented in this prototype to assure that there are no inconsistencies from the reasoning. The validity check performs a global check across all schema and instance data. The returned result of this check is a pass/fail flag and a list of specific reports that describe the found inconsistencies. The program has been designed to write the results of the validity check (pass or fail), as well as the descriptions of the reports to the program log file. If there are inconsistencies they will be listed in this file. It is then the job of the programmer to fix the program so that these inconsistencies no longer occur. The program has been designed so that validation can be easily turned on, or turned off if performance becomes an issue.

In conclusion, Chapter 4 has looked at the architecture design of the online dating model. A detailed description of the online dating model was provided in this chapter. In the next Chapter it will be discussed how this dating model was implemented on a commercial online dating website. Chapter 4 also described the design decisions that led to using RDFS as the ontology language and Sesame to store the knowledge base. In this
chapter the general purpose rules used in this prototype were described. In the following chapter the results after implementing these rules on the online dating website are discussed. Also in Chapter 5, the key operations of the prototype as well as experimental results under typical operations are explored.
Chapter 5

Matchmaking Prototype

In the previous chapters the design of the ontologies and the knowledge base was described. In this chapter, the functionalities of the matchmaking prototype developed in this project are described. Key operations of the prototype are listed in Section 5.1. In Section 5.2 the application of the developed prototype to a commercial dating website is explained. Finally in Section 5.3 experimental results under typical conditions are reported.

5.1 Key Operations of Prototype

The developed matchmaking prototype has several main operations. The operations are as follows:

- Infer additional information about users based on the geographical information to be used for:
  - Delivering personalized level 4 questions to a user based on their inferred facts.
  - Adding an inferred level to the matching process.
- Update a user’s answers in the knowledge base based on the information they provided on the online dating website.
- Generate matches using SPARQL Queries.
- Use a calculated score to rank matches.
5.1.1 Personalizing User Questions

The prototype developed in this project begins by taking all information in the MySQL database, and information about the RDFS ontology and loading it into the knowledge base. Reasoning is then performed on the knowledge base. Once the inferred facts have been added, the program generates a list of questions that a user has left to answer. User questions are stored in a MySQL database. Users answer all level 1, 2, and 3 questions; however in level 4 there is a wide range of questions that are dependent on statistical values and users will only be presented with the questions that apply to them.

Questions have been personalized so that the user will feel like the website knows more about them than they have specified. Level 4 questions begin by stating a fact that has been pulled from the statistical data. Examples of these facts have been summarized in Table 3. A user’s personalized questions begin with one of these customized facts. Following the fact a user is asked whether or not their own personal answer to the question corresponds to the fact stated. For example, a user living in an area where the average income is $37,984 will be posed the following question regarding the *inferred_income* property: “The average income in your region is in the $30,000 - $40,000 range. Does your income fall in this range?”. The user is presented with the following option to the inferred income question: “Yes”, “No, higher”, “No, lower”. By including this fact the user gains an interesting piece of knowledge about their neighborhood. By posing this question the website is asking a question about a user’s income in a less intrusive way. The user is not asked what their income is, but just asked if it is in a specified range, if it is not in this range then the user need not specify what range it is in, just whether it is higher or lower than this range. Questions are formulated
and posed in the same way for all other inferred properties.

<table>
<thead>
<tr>
<th>Inferred Property Name</th>
<th>Inferred Property Value</th>
<th>Specialized User Fact</th>
</tr>
</thead>
<tbody>
<tr>
<td>inferred_income</td>
<td>37,984</td>
<td>The average income in your region is in the $30,000 - $40,000 range.</td>
</tr>
<tr>
<td>inferred_mobility_one</td>
<td>15</td>
<td>About 15% of people in your region have moved in the past year.</td>
</tr>
<tr>
<td>inferred_mobility_five</td>
<td>39</td>
<td>About 40% of people in your region have moved in the past five years.</td>
</tr>
<tr>
<td>inferred_language</td>
<td>English</td>
<td>About 95% of people in your region’s primary language is English.</td>
</tr>
<tr>
<td>inferred_generation</td>
<td>Third</td>
<td>About 90% of people in your region are third or greater generation (both your parents born inside the country).</td>
</tr>
</tbody>
</table>

Table 3. Example facts used in the beginning of a user's questions

A person’s list of questions to be answered is added to the knowledge base in the form of an RDF triple with the person’s identification as the subject, questions as the property, and a comma separated numbered list of questions as the value. To select the level 4 questions, the program runs a SPARQL query to obtain inferred information about the user and uses that information to determine which question in each category should be asked. Another SPARQL query is executed to determine if a user has already answered a specified level 4 question, if ‘yes’ then that question is not added to the list. For each level 4 question a user has answered the inferred fact currently in the knowledge base is removed and replaced by the user’s answer. It is this value that will be used in the subsequent matching.

5.1.2 Matching Users on Inferred Properties

Each level of matching in this prototype has its own SPARQL queries based on that level’s properties. The first level consists of one query that filters for all potential
partners who meet the user’s level 1 requirements. If “user1” and “user2” match on level 1 a triple is added to the database in the form:

`acu:user1_id    acu:matchl10    acu:user2_id`

The property `matchl10` (match on level 1) is used by all other queries to assure that users match on the required first level. The complete level 1 and level 2 SPARQL queries for a user can be found in Appendix A.

The level 4 queries were the queries that were developed in this prototype. In level 4 users are matched based on the properties that have been inferred about them. For example, they are matched on the `inferred_income` property. The query that checks to see if users match on this property is the following:

```sparql
SELECT ?y
WHERE {
1   acu:001-fname-lname    acu:matchl10    ?y
2   acu:001-fname-lname    a    acu:Person.
3   ?y    a    acu:Person.
4   FILTER(acu:001-fname-lname!= ?y)
7   FILTER (?in1+10000 >= ?in2) .
8   FILTER (?in1 - 10000 <= ?in2) .
}
```

Line 1 of the `WHERE` clause selects users in the knowledge base that match a user we will refer to as “Person1” on level 1. Next, in line 2 the query checks that each of those results are of type `Person` and line 3 assures that `Person1` themselves is not included in the results. In line 5 `Person1’s` `inferred_income` property is retrieved and stored in variable `in1`. Line 6 finds all values for the `inferred_income` property for all the results found earlier (`y`) and stores those values in `in2`. Lines 7 and 8 are used to eliminate all users
with inferred_income not within $10,000 of Person1’s inferred_income. Therefore, this rule enforces that users will score higher if they have similar average incomes. A similar rule is written for each inferred property assuring that users either match on an inferred property, or are within a certain range of another user’s inferred property.

5.1.3 Scoring a Match

Each match between two users is given a score of how well those users match. Similar to the level 1 SPARQL query, for each SPARQL query that users match upon a new statement is added to the knowledge base in the form:

\[ \text{acu:user1 acu:matchlevel acu:user2} \]

For example, if two users, who will be referred to as “Person1” with ID 001 and “Person2” with ID 002 (a new user), match on two level 4 SPARQL queries then the following two triples would be added to the knowledge base:

\[ \text{acu:001-fname-lname matchl4.1 acu:002-fname2-lname2} \]
\[ \text{acu:001-fname-lname matchl4.2 acu:002-fname2-lname2} \]

By adding these triples another SPARQL query can be added to query for all levels/sublevels that two users match on. This additional SPARQL query is constructed once the matching is done and all of the match level triples have been added to the knowledge base. The query assigns a score value to each level/sublevel two users can match on. Lower levels are weighed more heavily than the level 4 queries, as lower levels use user provided information. For example, suppose there is one query for level 1, one query for level 2, and three queries for level 4 that users can match on. Then the query to find the scores of Person1’s matches is the following:
SELECT ?y (SUM(?score) AS ?s)
WHERE{
    {select ?y (30 as ?score) where {
        acu:001-fname-lname acu:matchl10 ?y. }
    } union
    {select ?y (20 as ?score) where {
        acu:001-fname-lname acu:matchl20 ?y. }
    } union
    { select ?y (3 as ?score) where {
        acu:001-fname-lname acu:matchl41 ?y. }
    } union
    { select ?y (3 as ?score) where {
        acu:001-fname-lname acu:matchl42 ?y. }
    } union
    { select ?y (3 as ?score) where {
        acu:001-fname-lname acu:matchl43 ?y. }
    } union
    }
} group by ?y
order by desc(?s)

Line 1 specifies that the y variable should be returned, and that there should also be another variable score, whose values should be summed and stored in the variable s. In this example, if users match on level 1 a value of 30 is added to their final score. Lines 3 and 4 query for all users who match Person1 on level 1 and adds 30 to the score of the match of any users that do. Next lines 6 and 7 query for all users who match Person1 on level 2 and adds 20 to the score of the match of any users that do. The same is done for each of the level 4 queries, adding a value of 3 to the score of the matches of any users who match on that level. Initially each level 4 query that users may match on adds only a score of 3 to the final score. Each query in the lower levels that two users may match on adds a score between 15 and 30 to the final score. Line 19 orders the list of scored matches in descending order. This ranked list is returned to the online dating website in
that order so that a user’s list of matches will be displayed such that the users they score the highest with are presented first. This system allows for weights of levels to be changed easily and frequently if there is a negative response from users.

5.1.4 Returning a User’s Matches

When a user requests their matches another program of this prototype is executed. The program finds a user’s matches by using specified SPARQL queries. Before creating a list of queries that match users the prototype uses SPARQL queries to check if a match already exists. As discussed in the previous sections when two users match on a level a triple is added to the knowledge base to indicate that they match on that level. Thus the first query that is executed when matching users is one to check if their matches already exist. If they do then that list of matches is used. Otherwise if the query for current matches returns nothing, a series of SPARQL queries are formulated to find a user’s matches. When the personalized SPARQL queries are created they are also stored in the knowledge base in a triple that has the user as the subject, the rule property as the property, and the list of SPARQL queries as the value of the triple. Therefore when rules are being created, there is a query to check whether or not the rules have already been created. This ensures that the processing time of creating and running SPARQL queries does not have to be done more than once. Thus, until a user answers another question about themselves their matches will not change. The method that is called when a user answers a question contains a statement that deletes all of the user’s previous matches from the knowledge base. That is, it deletes all triples whose subject is the current user’s id, and whose property is a match level property. Therefore the next time a user requests
their matches their previously stored matches will no longer be in the knowledge base and thus new SPARQL queries to find their matches will be generated. In the programming environment, once all matches are returned they are given a score and printed out. On the dating website, once all matches are returned they are presented to the user.

5.2 Prototype Application and User Interface

The prototype developed in this project was developed to be used on a commercial online dating website. To run the prototype it is deployed onto the online dating website’s web server which uses Apache Tomcat [30]. Log files are included on the server so that any errors with the prototype can be traced. Once the program has been deployed it is available as web service end points. The front end invokes a post call indicating what action the program should perform (recreate the knowledge base, obtain questions for a user, perform the matching, etc.). Based on the request the system performs the processing and returns the result to the front end, the website.

Figure 9. User profile on the online dating website.
Users are able to sign up for the online dating website. Figure 9 is a snapshot of a user’s profile on the website. Each piece of information that can be seen in this profile is also included as an RDF triple in the knowledge base. Each question that a user is presented with is stored in the database. For example, in Figure 10 the prototype has determined that the user has not yet chosen which religion they practice and they are presented with that question to be answered. The user’s answer is stored in the MySQL database and eventually added to the knowledge base.

![Figure 10. User question from the online dating website](image)

5.3 Experimental Results Under Typical Operation

Testing is an important technique for assessing the quality of the items developed in this prototype. In addition to evaluating the success of the returned matching results, testing also helps detect errors or component failures. In software design there are two basic forms of testing that are commonly used, which are functional (black box) testing and structural (white box) testing [11]. Functional testing accesses the overall behavior of the program and focuses primarily on the outputs generated by the system based on
certain inputs and conditions. It is considered black box testing because no knowledge of the internal system is used to develop test cases. Alternatively, structural testing uses knowledge of internal system structure to develop test cases. It is an analysis of the system’s internal structure design. Ideas from both types of testing are employed when evaluating the design of this prototype.

The prototype developed in this application was tested on two different systems. The system that it was primarily tested on was a computer using the Eclipse IDE [38] and a locally stored MySQL database. Testing in this environment allowed for easy manipulation of user properties and system configurations. When thorough testing on this system was complete, the prototype was tested on a server running an online dating website to assure that behavior remained the same when deployed for use on a dating website. This section lists the cases that were used to evaluate the system.

5.3.1 Inferring Additional Properties

The knowledge base consists of facts about Canadian and U.S. cities which are used to add more data to the knowledge base about users from that city. The reasoner will reason over the data and additional facts will be inferred about users. These facts are not only used in the matching process but also to present users with personalized questions that give the site a personalized feel for the user.

In this test case the user is from Windsor, NS. This location is used to ensure that the program can differentiate between data collected about Windsor, NS and data collected about Windsor, ON, in addition to testing typical system functionality.
5.3.1.1 Program Evaluation

To demonstrate the functionality of the Java program created in this project, a user is added to the MySQL database whose relevant information can be seen in Figure 11. A statement is added to the program code to print out every inferred triple that is associated with Person1. This code as well as the results of executing the program over a database containing these facts can be seen in Figure 12.

![Figure 11. Screenshot of MySQL database entry for test user](image)

The results are printed out in triples, and the full namespace (which has been partially blurred) is printed for each part of the triple. For ease of reading, the results are reproduced below with a prefixed namespace:

(acu:001-fname-lname, acu:inferred_language, acu:English)
(acu:001-fname-lname, acu:inferred_mobility_one, "16"^^<xsd:int>)
(acu:001-fname-lname, acu:inferred_mobility_five, "46"^^<xsd:int>)
(acu:001-fname-lname, acu:inferred_income, "27184"^^<xsd:int>)
It should also be noted that the user Person1 is a female. The results can be checked by referring to the summary of relevant information in Tables 4 through 8. Since the user in question is female, the female population statistics are used to evaluate the results. It can be seen that the inferred language should indeed be English as 97% of the population in that city speak English the most at home. The inferred_language property refers to the percent of people who have moved in the past year, and the inferred_language property refers to the percent of people who have moved in the past 5 years. Calculating this by looking at the number of females who live in the same location as they did one
and five years ago it can be seen that these numbers are correct. Comparing the value in Table 8 and the inferred income value it can be seen that they match. Therefore properties are being inferred correctly in the programming environment.

<table>
<thead>
<tr>
<th>Language spoken most often at home</th>
<th>Total</th>
<th>Male</th>
<th>Female</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Population</td>
<td>3,430</td>
<td>1,540</td>
<td>1,890</td>
</tr>
<tr>
<td>English</td>
<td>3,325</td>
<td>1,470</td>
<td>1,855</td>
</tr>
<tr>
<td>French</td>
<td>10</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>Non-official language</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4. Language Statistics about Windsor, NS

<table>
<thead>
<tr>
<th>Generation Status</th>
<th>Total</th>
<th>Male</th>
<th>Female</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Population 15 years and over</td>
<td>2,865</td>
<td>1,260</td>
<td>1,600</td>
</tr>
<tr>
<td>1st generation</td>
<td>210</td>
<td>105</td>
<td>110</td>
</tr>
<tr>
<td>2nd generation</td>
<td>185</td>
<td>115</td>
<td>70</td>
</tr>
<tr>
<td>3rd generation or more</td>
<td>2,465</td>
<td>1,040</td>
<td>1,425</td>
</tr>
</tbody>
</table>

Table 5. Generation Statistics for Windsor, NS

<table>
<thead>
<tr>
<th>Mobility – Residence 1 year ago</th>
<th>Total</th>
<th>Male</th>
<th>Female</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Population 1 year and over</td>
<td>3,395</td>
<td>1,520</td>
<td>1,875</td>
</tr>
<tr>
<td>Lived at the same address 1 year ago</td>
<td>2,855</td>
<td>1,280</td>
<td>1,570</td>
</tr>
</tbody>
</table>

Table 6. One Year Mobility Statistics for Windsor, NS
<table>
<thead>
<tr>
<th>Mobility – Residence 5 year ago</th>
<th>Total</th>
<th>Male</th>
<th>Female</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Population 5 years and over</td>
<td>3,250</td>
<td>1,460</td>
<td>1,795</td>
</tr>
<tr>
<td>Lived at the same address 5 years ago</td>
<td>1,790</td>
<td>830</td>
<td>960</td>
</tr>
</tbody>
</table>

Table 7. Five Year Mobility Statistics for Windsor, NS

<table>
<thead>
<tr>
<th>Income in 2005</th>
<th>Total</th>
<th>Male</th>
<th>Female</th>
</tr>
</thead>
<tbody>
<tr>
<td>Persons 15 and over with income</td>
<td>2,485</td>
<td>1,090</td>
<td>1,390</td>
</tr>
<tr>
<td>Median income – Persons 15 and over who worked full year, full time</td>
<td>33,774</td>
<td>42,363</td>
<td>27,184</td>
</tr>
</tbody>
</table>

Table 8. Income Statistics for Windsor, NS

5.3.1.2 Application Evaluation

In the application of the system to an online dating website inferred facts are used not only in the matching but also to generate specialized questions, as discussed in Section 5.2. Test cases are generated by testing users from several different locations. Each time a user’s location changes the level 4 questions that user is presented with should change accordingly. The profile in Figure 9 is used to test inferred facts in the application of the system to an online dating model. Again it is expected that the output will pertain to the data associated with Windsor, NS. When the user is prompted to answer level 4 questions, an example of the question they are given can be seen in Figure 13. In the previous section it was noted that the average income of a female in Windsor, NS is $27,184. Therefore this selected question has the appropriate range for the given user. In Figure 14 it can be seen that the correct question regarding the person’s mobility in the last year has been selected. It was noted that 16% of females moved in the past
year, and the question (which rounds that percentage to the nearest multiple of 5) has correctly displayed this information. In each case using different locations the correct questions were presented to the user.

![Figure 13. Screenshot of income question asked to the user](image13)

![Figure 14. Screenshot of mobility question asked to the user](image14)

### 5.3.2 The Matching System

The main function of the prototype developed in this project is to provide users with a set of suitable matches. The score of a match between two users is directly related with properties two users meet on. There are several aspects of this system which must be evaluated when testing the matching system. The following sections include test cases to check the performance of the matching engine both in the programming environment and
in the real-world application of the online dating website.

5.3.2.1 Program Evaluation

To test the prototype in the programming environment several users are added to the MySQL database already containing pre-existing users. In the first test case two users are added to the database who match each other on every level. In this case, when finding the matches for one of those users the other user should always be returned as their top match (unless another user also matches them on every level). In this test case user 001 has location Bronx, New York, and user 002 also has location Bronx, New York. This location is used because the male and female properties associated with this location differ very little, therefore the users will match on all aspects of level 4. Figure 15 displays the textual list of the top 10 matches returned by executing the program over the described database for user 001. It can be seen that user 002 is indeed the top match with

![Match Results]

Figure 15. Snapshot of results from running the first test case
a highest possible score of 100.

Figure 16. Snapshot of results after altering user 002’s level 1 properties

The program is next tested by changing one of user 002’s level 1 properties so that they no longer match user 001 on level 1. In this case the results should not include user 002, but otherwise be the same. The results of running this test case can be seen in Figure 16. The list of results returned is the same as the list returned in the last test case, however in this case user 002 has been removed, and another user is added to the bottom of the list.

The second test case tests the program on a database with users who match user 001 on different levels. Table 9 summarizes the users used in this test case and on which properties they match user 001 (it can be assumed that level 3 scores are arbitrary). This test case is used to examine how level 4 properties affect the match score results. Users that match on level 1 and level 2 should score higher than users who don’t match on level
2 but perhaps match on level 4. All other users are removed from the database for this test case. Figure 17 displays the results in matching order of running the program over the described database.

<table>
<thead>
<tr>
<th>User</th>
<th>Matches on Levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>6074-dave-lname</td>
<td>1, 2.1</td>
</tr>
<tr>
<td>3947-shawn-lname</td>
<td>1, 2.1, 2.2</td>
</tr>
<tr>
<td>8021-james-lname</td>
<td>1, 4.1, 4.2, 4.3, 4.4, 4.5</td>
</tr>
<tr>
<td>5925-matt-lname</td>
<td>1, 2.1, 4.1, 4.2</td>
</tr>
<tr>
<td>6773-jeff-lname</td>
<td>1, 2.2, 4.3</td>
</tr>
<tr>
<td>5467-jake-lname</td>
<td>1, 2.1, 2.2, 4.2</td>
</tr>
<tr>
<td>8474-mike-lname</td>
<td>2.1, 4.3, 4.4</td>
</tr>
</tbody>
</table>

Table 9. Test case users and their match levels

The results show that the user 5467 who matches on levels 1, 2.1, 2.2, and 4.2 scores the highest. This is because they are the only user to match user 001 on both level 2.1 and level 2.2 and a level 4 sublevel. User 8474 does not match user 001 on level 1 and therefore is not included in the results. The user who scores the lowest is user 8021. Even though they match user 001 on four level 4 sublevels, more level 4 matches than any other user, they do not match the user on any of the level 2 sublevels. However, users 5925 and 6773 have scores ranked in the middle of the list because they match on some level 2 sublevels and some level 4 sublevels. Therefore the level 4 inferred properties are not used to create a high scoring match where none existed before, but are used to rank users who have similar scores in the lower levels.
5.3.2.1 Application Evaluation

The prototype was also evaluated in the real-world application of an online dating website. It was tested on a site that has approximately 3000 users. When a user registers for the site they are asked very basic information that includes: their age, their gender, and what gender they are seeking. This information is used by a syntactical algorithm to produce an initial list of matches for that user according to these very basic properties.

This user is not included in the knowledge base until the next time that the knowledge base is recreated as this is where the reasoning happens. Thus, the user will not appear in any other users’ matches until the knowledge base is recreated, which in the site’s current format happens once a day but that user may still request very basic matches.

Figure 17. Snapshot of results from running second test case
In this test the process of creating a user, updating their information and requesting their matches is tested. Suppose that the user’s name is “Jane Doe”, then they create their profile by filling out the information seen in Figure 18. This information is saved in the MySQL database. We will refer to this user as “Person1”. If Person1 requests their matches at this point they will be given a list of users who match their gender preference, which in this case is men. Person1’s profile is then filled out with several properties about themselves. Next, a manual web service call is made on the server to recreate the database and perform reasoning. It takes approximately 15 minutes for the web service call to complete. All the new users in the system will be added to the knowledge base and any information about a user’s location, or anything that affects the reasoning will be reflected in the knowledge base. At this time the website has the same performance as when the web service call is not being performed. Person1 then requests their matches and the list in Figure 19 is returned. Next, Person1 answers several preference questions including the question about their preference of age in a partner. Person1 specifies that the minimum age they will accept in a partner is 24, and the
maximum age that they will accept in a partner is 30. To test that the queries to find their matches have been updated, they then request their matches again and a different list is returned and can be seen in Figure 20.

Figure 19. List of matches returned the first time Person1 requests their matches

Figure 20. List of matches returned after Person1 has specified an age restriction
The information and pictures of the matches returned in Figures 19 and 20 have been blurred because this is information of real users on a commercial dating website. For comparison purposes the age of the user and the city that they are from has not been blurred. It can be seen in Figure 20 that the list of matches has been changed completely compared to the list in Figure 19. All users returned in Figure 20 are in Person1’s preferred age range and match Person1 on the properties that they have specified. Therefore the queries to find Person1’s matches have been correctly updated.

The first test case used in the programming environment is also used in this application. Two users are created with identical properties. Again, both users will have their location as Bronx, New York. Person1’s profile is used and another profile, one of Person2 whose display name is “John”, is created. Both users answer the questions about themselves so that they match the preferences of the other user. After both profiles have been created Figure 21 is a snapshot of the results provided when Person1 requests their matches. The program successfully recognizes that Person2 with username “John”, is their top match. Next, Person2’s *religion* property is changed so that it no longer matches Person1’s preference, which is a level 1 requirement. The results from recreating the database and again requesting Person1’s matches are shown in Figure 22. It can be seen that the same matches that were returned in the first case are again returned except for Person2 who is no longer contained in the list of matches.
Figure 21. John is returned as Person1’s top match

Figure 22. Person1’s results after John no longer matches their preference on level 1
Chapter 6

Conclusions

This thesis focused on the development of a knowledge-based human matchmaking prototype. The knowledge base is comprised of facts about user properties, user preferences and geographical properties which are structured by light-weight ontologies. This well-structured knowledge base is complemented with a set of rules used by a reasoner to add additional facts to the knowledge base. In order to show a real-world implementation of this prototype, the developed code was implemented on a commercial online dating website. In this chapter the research presented in this thesis is summarized and future work in the area is discussed.

6.1 Contributions

The main contribution of this thesis lies in the introduction of semantic web techniques in an online dating matchmaking algorithm. This thesis’s main focus was to design, extend, implement, and demonstrate the possibilities of a semantic matchmaking model for an online dating website. Ontologies were designed using RDFS to capture information about the dating website’s users, as well as Canadian and American cities. Next, general purpose rules were produced to infer additional facts to be added to the knowledge base. Once additional facts about users had been inferred, SPAQRL queries were used to match users on different levels. This prototype has the following distinct operations: infer additional information and add it to the knowledge base, deliver user’s personalized questions, and use SPARQL queries to match users.
The prototype has explored the reasoning potentials of rules over a complete knowledge base. Results of running the prototype show that the rules are inferring the expected facts. The system successfully provides a list of matches using information that has been inferred about users. Evaluation of the prototype also shows that by implementing a Sesame Native store instead of the previously used Sesame RDBMS store, performance of the creation of the knowledge base and performing reasoning could be increased from approximately 2 hours, to 15 minutes.

Integrating datasets to expose new pieces of information is certainly not a new endeavor. It is one of the main focuses of the semantic web. However, this prototype is an original application of integration of datasets on an online dating website. It is built upon the work of others who proposed the original semantic matching. By integrating geographical datasets with the dataset containing user information it is a good starting place for thinking about how other large scaled datasets could be integrated into the system and used in the matching process.

6.2 Future Work

The prototype currently only infers additional information based on which city a user is from. In the future other datasets pertaining to other user facts could be added to the knowledge base and be used to infer additional information. Another property about a user that could be used to infer additional facts is the user’s occupation. A person’s occupation can be mapped to specific personality traits. Therefore by inferring these properties about users with corresponding occupations, even more information can be used in the matching. However this requires that there exist available datasets about the
mappings between occupations and personality types or involves the process of creating these RDF datasets.

In this thesis, only the information from the statistical data has been considered as a means to infer additional information and not the users’ answers themselves. For example, the primary language of a user has been inferred based on what city the user is from. However, if several users from that city who join the dating website indicate that the language that has been inferred about them is incorrect then the answer that should be inferred about users from that city must be re-evaluated. A metric for an inferred answer’s success rate must be developed. Another user aspect that must be taken into account is whether a user is more satisfied with their matches in this new matching system or is less satisfied. If users prefer their list of matches before the additional facts have been added and used in the matching then the weight of those queries pertaining to inferred information needs to be adjusted or information other than geographical information should be used to infer the additional facts about a user.

Another aspect to focus on in the future is the implementation of a system that automatically updates the knowledge base when users join the website. Running the algorithm to recreate the knowledge base and perform reasoning each time a user joins the website could result in repeated calls to that process and cause the server’s CPU to be used completely by those processes which may potentially crash the site. Therefore, potential ways to update a user’s information instantaneously without crashing the website should be investigated. This would increase the utility of the key operations of this prototype.

Lastly, it is possible that the approaches implemented in this prototype are also
applicable to other fields of matching, such as matching customers to products. By using data about where a customer is from to infer additional facts about the user those facts can then be used to better match users to products. For example, if a user lives in an area that has a higher than most average income then that user may be more interested in the more expensive products of the results that they are being returned. Data about products that other users from that city have purchased could also be used in the matching of products to users. Future research is needed to determine the potential of other data integrating semantic matchmaking systems.
References


applications research laboratory. 2004.


Appendix A: RDF Representations

RDF triples for user 001 with first name “fname” and last name “lname”:

acu:001-fname-lname acu:first_name       “fname” .
acu:001-fname-lname acu:last_name        “lname” .
acu:001-fname-lname acu:smoking_habits acu:no .

acu:001-fname-lname acu:pets_view  acu:one-or-two-pets-only  .
acu:001-fname-lname acu:pets_view  acu:one-or-two-pets-only  .
acu:001-fname-lname acu:have_children  acu:yes-live-at-home  .
acu:001-fname-lname acu:pref_age_min  acu:60  .
acu:001-fname-lname acu:pref_age_max  acu:70  .
acu:001-fname-lname acu:max_distance  “1000km”  .
RDF triples about the city of Fredericton:

```
acu:fredericton_NB acu:income_total "36927"^^xs:int .
acu:fredericton_NB acu:income_male "41686"^^xs:int .
acu:fredericton_NB acu:lang_percent_total "92"^^xs:int .
```
acu:fredericton_NB acu:lang_percent_male "92"^xs:int .
acu:fredericton_NB acu:mobility_one_total "21"^xs:int .
acu:fredericton_NB acu:mobility_one_male "21"^xs:int .
acu:fredericton_NB acu:mobility_one_female "20"^xs:int .
acu:fredericton_NB acu:mobility_five_total "49"^xs:int .
acu:fredericton_NB acu:mobility_five_male "50"^xs:int .
acu:fredericton_NB acu:mobility_fiveFemale "48"^xs:int .
.acu:fredericton_NB acu:generation_total "Third" .
acu:fredericton_NB acu:generation_male "Third" .
acu:fredericton_NB acu:generation_female "Third" .
.acu:fredericton_NB acu:generation_percent_total "81"^xs:int .
.acu:fredericton_NB acu:generation_percent_male "80"^xs:int .
.acu:fredericton_NB acu:generation_percent_female "82"^xs:int .

**Level 1 SPARQL query for user 001 Person1:**

SELECT ?y

WHERE {

    acu:001-fname-lname a acu:Person.

    ?y a acu:Person.


}
acu:001-fname-lname acu:age_pref_min ?age1Min.
acu:001-fname-lname acu:age_pref_max ?age1Max.
?y acu:age_pref_min ?age2Min.
?y acu:age_pref_max ?age2Max.
FILTER(?age1 >= xsd:integer(?age2Min)).
FILTER(?age1 <= xsd:integer(?age2Max)).
FILTER(?age2 >= xsd:integer(?age1Min)).
FILTER(?age2 <= xsd:integer(?age1Max)).
Level 2 SPARQL query for user 001 Person1:

```
SELECT ?y
WHERE {


}
```
acu:001-fname-lname a acu:Person.
?y a acu:Person.


}

SELECT ?y
WHERE {
  acu:001-fname-lname a acu:Person.
  ?y a acu:Person.
  FILTER(?xh >= xsd:integer(?yhmin)).
  FILTER(?xh <= xsd:integer(?xmax)).
  FILTER(?yh >= xsd:integer(?xhmin)).
  FILTER(?yh <= xsd:integer(?xmax)).
Curriculum Vitae

Candidate’s full name: Emily Wilson

Universities attended (with dates and degrees obtained): Bachelor of Science, Mount Allison University, 2011

Publications:

Conference Presentations: