Using INMDB for the Semantic Web

By

Patrick Demers

A thesis submitted to the Faculty of Computer Science in partial fulfillment of the requirements for the degree of

Master of Computer Science with Specialization in Data Science

Ottawa-Carleton Institute of Computer Science

Department of Computer Science

Carleton University

Ottawa, Ontario

Abstract

Resource Description Framework (RDF) is a data model that represents information about web resources using triples. Triples represent data at the atomic level using three components: subject, predicate and object. The structure of RDF data is defined by a schema language RDFS. RDF data storage can be classified into five categories: Native, Relational, Object-Oriented, Object-Relational and Expert Systems. Of these, Native and Relational have been the dominant approaches. The problem with all current RDF storage approaches is that none adequately represent the features of RDFS with the schema of their underlying storage architecture. In this thesis, the problem is addressed using Information Network Model Database (INMDB) which attempts to better represent RDFS as database schema. To support our claim, we store the English version of the DBPedia RDF dataset in INMDB. The results show that INMDB offers a more representative way to store and query structured RDF data.

Acknowledgements

I would like to thank Dr. Mengchi Liu for his guidance in choosing a thesis topic, obtaining the INM software for testing, help with troubleshooting, help with understanding the INM software and for his continuous support with revisions of thesis. I would also like to thank the members of the defense committee, Dr. Iluju Kiringa and Dr. Tony White, for their great and clear feedback on the thesis. A would also like to give a special thank you to Dr. David Mould for chairing the committee.

I would especially like to thank my family, co-workers and friends for their support, encouragement and advice. Their academic experience shined a guiding light for me throughout this journey.

Table of Contents

List of Tables

List of Illustrations

Chapter 1

Introduction

The Semantic Web is a new version of the web standardized by the World Wide Web Consortium (W3C) which allows the publishing of machine readable content on the internet. At the core of W3C's Semantic Web is a data model called Resource Description Framework (RDF). RDF is a semi-structured language originally designed to represent web resource information [1]. Today RDF has a much wider use than just for the semantic web. Some of these uses are to represent metadata [2], formal naming and definitions of entities (Ontologies[1]) [3,4], knowledge graph representation [5,6,7] as well as cataloguing webpages, books and articles [8].

Some popular examples of where RDF is applied are: DBPEDIA which extracts information from Wikipedia to enable users to query the data directly, Dublin Core Metadata Initiative [9] a metadata framework which aims to provide a minimal set of descriptive elements for describing and indexing web documents found on the web, and PRISM (Publishing Requirements for Industry Standard Metadata) [2] a metadata framework used by the publishing industry which enables publishers to publish their creative content in many different ways (web, book, email, etc). In the medical field the frameworks SNOMED RT (Systemized Nomenclature of Medicine Reference Terminology) [3] and MeSH (Medical Subject Headings) [4] have important roles in aiding in the dissemination of medical knowledge.

[1] Definition of Ontology: "A set of concepts and categories in a subject area or domain that shows their properties and the relations between them."

The RDF data model is based on statements about resources at the atomic level using an expression called a triple. A triple is a representation of a fact consisting of three parts: A subject, a predicate and an object. For example, on a web page, the following sentence could be encountered: “Wall-E is a robot”. This piece of data can be broken down into a triple (subject-predicate-object), where the subject would be “Wall-E”, the predicate would be “is” and the object would be “robot”. RDF data has a schema language called RDF schema (RDFS) which gives structure to the RDF data by enabling the creation of user defined classes and properties. RDFS classes are similar to those found in other object-oriented data models, where classes can inherit the properties of their parent class. RDFS properties are assigned to classes, can be given a datatype (e.g., String, Date or integer) as well as be defined as sub-property of another property (e.g., “has robot” can be a sub-property of “has machine”). RDF data sets (sets of triples) and RDFS can be naturally seen as a directed multi-graph where each subject and object of a triple is a node and each predicate is a directed edge on this graph. RDF has a standard query language, SPARQL [10,26,27], published by W3C. SPARQL is a graph query language which enables users to query RDF data at a high level.

The amount of RDF data available is continuously growing [11]. In 2007, there were over 2 billion triples in the linked open data cloud [12] and in 2011 the number of triples had grown to 31 billion [13]. By 2014, the number of linked open datasets (which store triples) had grown 271% [14,15] and in 2016 a single incomplete triple collection contained 38.5 billion cleaned triples [16].

1.1 Summary of Existing RDF Storage Solutions

Numerous surveys highlight the effort that has been applied towards the development of different RDF storage systems [17,18,19,20,21,22,23,24,25]. There are five common RDF data storage approaches. These include: Native [28,29,30,31,32,33,34,35,36], Relational [28,37,38,39,40,41,42,43,44,45,46,47,48,49,50], Object-Oriented [51,52], Object-Relational [53,54] and Expert Systems [55,56,57]. Of these, the native-based and relational-backed approaches are the most common [17,19,58].

Native storage is the approach of building a database management system from scratch that is designed specifically for storing and querying RDF data. The strength of these systems is founded on customized data structures, indexes and query processors. The underlying storage representation of the RDF data is modeled to interface with semantic web query languages (such as SPARQL) and thus optimize query answering [17]. Data can be stored either persistently on top of file systems or transiently in main memory. There are three types of native storage: triplestores, graph-based and in-memory.

The relational storage approach makes use of relational database systems to store and represent RDF data. RDF data can easily be mapped to the relational model[2]. These systems take advantage of the fact that relational databases have been under development for over 40 years and many problems related to storing and retrieving data efficiently in

[2] Tim Berners-Lee. "Relational Databases on the Semantic Web". 1998. https://www.w3.org/DesignIssues/RDB-RDF.html (Retrieved Aug 01, 2017)

these systems have been well studied and overcome. As a result, many relational databases today are equipped with sophisticated query optimizers.

The relational approach to storing RDF data has three major ways to model RDF data, these are known as: Vertical storage, N-ary storage and Binary storage. The first approach is somewhat of a naïve approach to storing RDF data whereas the latter two approaches are more complex, but have some very strong advantages.

Vertical storage is an approach that uses a single table for storing all the RDF triples and is the most commonly used approach for relational RDF data storage applications because of the simplicity of its implementation and its ability to express RDF graph queries [1,17,42,43,44,45,59]. SPARQL RDF query translation to SQL for vertical storage is straightforward. Vertical storage methods however, suffer from scalability issues because as the RDF dataset gets larger, the vertical approach has difficulty handling queries as they require many self-joins to be made on this large table. This leads to a polynomial time complexity for every query. In order to improve performance with this approach you must make extensive use of indexes.

N-ary storage combines the RDF data into relevant tables based on type of thing so that there is no need to use self joins (subject-subject joins). This method creates a table for every RDFS class with its corresponding RDFS properties being assigned as table attributes. When an RDFS property is multi-valued, another table is created to store triples for this property. SPARQL queries translated into SQL for this approach, result in translations with much simpler syntax (often less joins and less where clause conditions) than an equivalent SQL translation for the vertical storage method. N-ary storage has the advantage of offering better query performance (worst case polynomial but best case log)

for RDF queries that specify a type of thing (person, company, etc..), but has a disadvantage when queries do not specify a type of thing or predicate to be queried. This is because all tables must be analyzed to perform the query. It can also be a disadvantage when the tables are sparse because sparse tables induce overhead in query computation [60]. A major drawback for the N-ary method and the reason it is so rarely implemented is because this storage method cannot easily capture multi-valued properties.

The binary storage method takes a slightly different approach, where a table is created for every RDF property. When translating SPARQL to SQL for binary table storage, the resulting query translation has slightly more complex query syntax than both N-ary and vertical storage methods (as many different tables must be called using joins). This method offers better query performance than vertical or N-ary storage for general queries (not specified by type of thing) because binary tables are always non-sparse and thus do not incur large computational overhead. The best case time complexity for the binary method is polynomial, but to a much smaller degree than the vertical storage method. Both the binary storage method and vertical storage method can easily capture multi-valued properties.

A detailed review of the literature suggests that the use of object-oriented and object-relational databases for storing RDF is quite rare. Although no reason is formally stated in any of the cited papers of this thesis, the reason for this may be due to the complexity and cumbersome nature of reference pointers in these systems. Of the documented implementations, the object-oriented approach represents RDF as a graph and does not represent RDFS as database schema. In contrast, there are two object-relational approaches which attempt to use object-relational database schema to represent

RDFS. Both of these object-relational systems implement a variation of the relational N-ary approach [53,54]. With these two approaches, classes which can have property inheritance are used to represent RDFS classes and multi-valued attributes are supported. Furthermore, object pointers (references) are used to represent subject-object relationships. Despite not having been commercialized, these two are the best attempts so far to represent RDFS using database schema.

1.2 Problem with Existing Solutions

A detailed study of the state of the art storage approaches which have been implemented or proposed to date identified a common problem amongst all of them. These storage approaches do not fully represent RDFS as the schema of the underlying storage system.

The native approaches and the relational vertical and binary table approach make no attempt to represent the RDFS with their database schemas or storage architecture. The relational N-ary approach is significantly better at representing RDFS using database schema than these approaches; however, it is not able to represent several key features of RDFS such as inheritance, sub-properties or multi-valued attributes using a relational database schema.

As of this writing, the N-ary object-relational approaches are the best attempts to represent RDF schema as database schema by offering multi-valued attribute support, property inheritance and subject-object representations using object references [53,54]. Limitations of the object-relational databases used for these approaches do not allow for the representation of RDFS sub-properties. Furthermore, references in these systems do

not adequately represent RDF subject-object relationships because they use meaningless ID values.

1.3 Addressing the Problem with INMDB

INMDB is a semantic object-oriented database management system based off the Information Network Model (INM) [61]. INMDB provides classes which support property inheritance, sub-attributes and multi-valued support. As well, it provides a novel feature for modeling relationships between objects, referred to as linking, which is different from object-oriented and object-relational references. These links can model the roles that objects can play in relation to one-another (e.g., Person can be a director for a university). Role relations can be queried directly. Furthermore, INMDB automatically generates inverse links for every relationship.

The problem of representing RDFS as database schema is addressed in this thesis by using a novel mapping of RDFS onto the INMDB database schema. The representation for RDFS in INMDB is named the INMDB RDFS storage approach.

In this representation each RDFS class is represented as an INMDB class. RDFS subclasses are represented by INMDB property inheritance. Each RDFS literal property is represented as a multi-valued string attribute, in the class defined by its RDFS domain definition. Literal RDFS properties whose RDFS range data types are specified as a string, int, Date or url are cast as INMDB datatypes. RDFS object-mapping properties are represented as INMDB normal and role relationships (links), which target the class defined by RDFS property's domain. The value of the subject-object inserted into these links is simply the URI value of the triple's object. Furthermore, RDFS sub-properties are

captured by INMDB sub-attributes and sub-relationships. A complete illustration f this mapping with examples is presented in section 4.1.

1.4 Methodology

The nature of the work presented in this thesis is experimental. It's main objective is to investigate INMDB's ability to represent, store and query RDFS and lay out the ground work for future RDFS storage application projects to be built on top of INMDB. To determine INMDBs capacity, the English version of the DBPedia dataset is stored in INMDB using its novel storage approach. In order to draw comparisons with existing work, DBPedia is also stored in the relational database MySQL using the relational N-ary approach.

In order to store the dataset in INMDB, an algorithm is presented for loading static RDFS datasets (snapshots) into INMDB (section 5.3.1). This algorithm can be applied to any logically consistent RDFS dataset. In addition an algorithm for loading RDFS data into MySQL is also presented (section 5.3.1.1).

In order to verify the query functionality of INMDB, a systematic way of translating SPARQL graph queries to INMDB's query language is presented (section 4.2.1). Then using several SPARQL queries that are representative of its features, we verify whether or not these queries are translatable to INMDB query language (IQL) and whether they are supported by INMDB (section 4.2.2). The performance of the SPARQL queries translated to IQL is compared to their equivalent SQL translation by executing them on the DBPedia dataset (section 5.5).

1.5 Summary of Work

The investigation determined that INMDB can better represent RDFS than existing solutions (represents more RDFS features than existing systems) and is capable of storing large RDFS datasets. Most RDF queries were found to translate to INMDB (section 0). It was found that there is enough coverage of RDF query features to support typical user querying requirements; however, some queries could not be formulated in INMDBs query language, such as the BIND operator. The results of the query performance tests indicate that most queries are faster in INMDB (section 5.6).

1.6 Novel Contributions

The novel contributions made by the thesis are summarized as follows:

1. A mapping from RDFS to database schema that is able to capture more features of RDFS then existing mappings.
2. An experimental assessment determining if INMDB is able to represent, store and query RDFS. This includes an evaluation of supported SPARQL queries on INMDB and their relative query performance compared to a relational database system.
3. An algorithm for loading static RDFS datasets into INMDB.
4. A systematic way of translating SPARQL queries to the query language of INMDB.

1.7 Scope and Limitations of the Thesis

The scope of the thesis is limited to loading static datasets into INMDB (snapshots) without specific requirements in time complexity. Life cycle maintenance of the data loaded into INMDB and dynamic datasets support (Changing RDFS and database schema) are not addressed in this investigation. While the investigation of life cycle management of data and how well dynamic datasets are supported in INMDB via update, delete, insert and data integrity operations, is important, this thesis focuses on the preliminary ground work of querying and representing RDFS. This work though preliminary will facilitate future investigations of life cycle management and dynamic datasets.

Under the open world assumption, any RDF data (truthful or not) can be added to an RDFS dataset. This poses a number of problems for the quality of the data being inserted into the RDF datastores, such as logical inconsistencies (e.g., A person being both alive and dead) and misnaming of entities (e.g., Two different predicates that mean the same thing “book writer” and “author”).

The approach taken in the thesis for dealing with the truthfulness of data is to apply the closed world assumption and to assume the data is truthful until a conflict arises. In the closed world approach data that is logically inconsistent or does not conform to the RDFS is rejected (see section 5.2 for how this was handled). For several real world applications such as the medical ontologies, it is preferred to use the closed world assumption, where a predefined vocabulary is enforced. It should be noted that the DBPedia dataset had very few such conflicts. Other datasets, such as raw data

scraped from the web may require a more thorough cleaning and verification for truthfulness.

The focus of the thesis was to determine a working set of algorithms for loading data and translating queries. Since static datasets were used, these algorithms did not have to meet a particular time complexity constraint. Future investigations surrounding dynamic datasets would require more emphasis on competitive timeliness of updating, inserting, deleting and translating data.

In this thesis, only relational databases will be compared to the INMDB solution. MySQL is used as a relational backend in the system Jena 2[41], which implements N-ary representation. The relational N-ary storage representation was selected for the comparison because our investigation found that this approach was the most well-known of the disk-based RDFS representations approaches (N-ary relational vs. object-relational). Object-relational approaches are obscure, not commercially implemented and lacking development for SPARQL querying, and would therefore not provide as meaningful a basis to perform the comparison.

Chapter 2

Background

In this chapter a detailed summary of background information is presented. The chapter is outlined as follows. Section 2.1 describes what RDF, RDFS and OWL are, as well as their query language SPARQL and its algebra. A running is example is also presented in this section which will be used throughout the remainder of the thesis. Section 2.2 introduces INMDB and its three languages: IDL, IML and IQL.

2.1 Semantic Web Technologies

This section describes the basic background of semantic web data models and query languages. It is outlined as follows: section 2.1.1 introduces the basics of the RDF data model, section 2.1.2 introduces RDF schema and section 2.1.3 introduces OWL (Web Ontology Language). Then in section 2.1.4, querying RDF is described as well as the RDF graph query language SPARQL. The section 2.1.5 introduces SPARQL algebra and how to translate a SPARQL query into SPARQL algebra. The section 2.1.6 describes some application where RDF is used in the real world. Finally, section 2.1.7 summarizes a working RDF with RDFS example which will be used to illustrate the different RDF storage approaches discussed in the thesis.

2.1.1 Resource Description Framework

Resource Description Framework (RDF) is a data model used for describing information about web resources and knowledge management applications [1]. This data

model facilitates data interchange on the internet by allowing data from many different data schemas to be merged together easily[3] by using atomic statements known as triples to describe data resources. The atomic nature of these statements allows different datasets to easily be merged together, simply by adding any new triples to an existing set of triples. At the abstract level, a set of RDF triples can be seen as a directed graph. In this form, subject and objects are represented as nodes while predicates are represented as edges on the graph.

A triple is composed of three components: subject, predicate and object. The subject is used to identify the resource that is being described, the predicate is used to identify a trait of the resource or a relationship that the resource has with another resource, and the object is a value for the predicate. For example, on a web page you could encounter the following sentence: “Wall-E like Eva”. This sentence can be broken down into a triple (subject-predicate-object), where the subject would be “Wall-E”, the predicate would be “likes” and the object would be “Eva”.

The subject of an RDF triple in the semantic web is either what is called a uniform resource identifier (URI) or a blank node, both of which signify the resource being defined. A uniform resource identifier is a string of characters used to uniquely identify a resource. For the semantic web, the most common type of uniform resource identifier is the uniform resource locator (URL), which is also known as the web address. Other examples of possible URIs in real world applications would be unique identifiers such a global unique identifier, a social insurance number, email address, ISBN number

[3]Although merging datasets together is easy, guaranteeing the integrity of the data during merging is a complex endeavor and requires careful checks to ensure that consistency is maintained. E.g. When predicates from different datasets are meant to be the same, but are not.

or telephone numbers. A blank node is considered an anonymous resource, akin to a null value or free variable.

The predicate of an RDF triple is a URI which indicates a relationship but is not a blank node. The predicate URI is generally an agreed upon value, defined by a vocabulary that is standardized such as those proposed by W3C or other metadata initiatives. This allows data to be integrated from many different resources because everyone has agreed to define properties of real objects with a common vocabulary.

The object is either a URI, a blank node or a literal such as: string, integer, Date, Boolean, distance, etc.

The following is a small example text which will be expressed in RDF. It will be used throughout the thesis to illustrate key principles, to explain differences between models and query syntax.

Example Text

A movie called "Wall-E" has a director "Bob Smith", two writers, "Andrew Stanton" and "Pete Docter", and a year that it was released "2008". "Wall-E" also has a sequel called "Wall-E 2" which is also directed by "Bob Smith". Both movies are produced by a company called "Disney". "Disney" has a chief writer "Andrew Stanton" and was founded in 1940.

The triples for this example are shown below. "Disney", "Wall-E" and "Wall-E 2" are subjects expressed as URIs "<ex.ca/r/Disney>", "<ex.ca/r/Wall-E>" and "<ex.ca/r/Wall-E 2>". Director, Sequel, produced by, writer and year released are

predicates expressed as URIs “<ex.ca/p/Director>”, “<ex.ca/p/Sequel>”, “<ex.ca/p/producedby>”, “<ex.ca/p/writer>” and “<ex.ca/p/yearReleased>”. The objects are “Andrew Stanton”, “Bob Smith”, “Pete Docter”, 1940 and 2008 expressed as literals (strings or integers). The objects “<ex.ca/r/BobSmith>” and “<ex.ca/r/AndrewStanton>” are object mappings.

Wall-E Triples Example

(<ex.ca/r/Wall-E>, <ex.ca/p/Director>, <ex.ca/r/BobSmith>)
(<ex.ca/r/Wall-E>, <ex.ca/p/Writer>, <ex.ca/r/AndrewStanton>)
(<ex.ca/r/Wall-E>, <ex.ca/p/Writer>, <ex.ca/r/PeteDocter>)
(<ex.ca/r/Wall-E>, <ex.ca/p/YearReleased>, 2008)
(<ex.ca/r/Wall-E>, <ex.ca/p/Sequel>, <ex.ca/r/Wall-E 2>)
(<ex.ca/r/Wall-E 2>, <ex.ca/p/Director>, <ex.ca/r/BobSmith>)
(<ex.ca/r/Wall-E>, <ex.ca/p/Madeby>, <ex.ca/r/Disney>)
(<ex.ca/r/Wall-E 2>, <ex.ca/p/Madeby>, <ex.ca/r/Disney>)
(<ex.ca/r/Disney>, <ex.ca/p/Yearfounded>, 1940)
(<ex.ca/r/Disney>, <ex.ca/p/ChiefWriter>, <ex.ca/r/AndrewStanton>)
(<ex.ca/r/AndrewStanton>, <ex.ca/p/Name>, “Andrew Stanton”)
(<ex.ca/r/PeteDocter>, <ex.ca/p/Name>, “Pete Docter”)
(<ex.ca/r/BobSmith>, <ex.ca/p/Name>, “Bob Smith”)

The RDF graph representation for this example is shown in Figure 1. At any time, new edges can be created on the graph using semantic association discovery, also known as inference. For example if “Wall-E 2” was made by “Disney” and “Disney” was founded in “1940”, we can infer that “Wall-E 2” was made sometime after “1940” even though it is not explicitly stated.

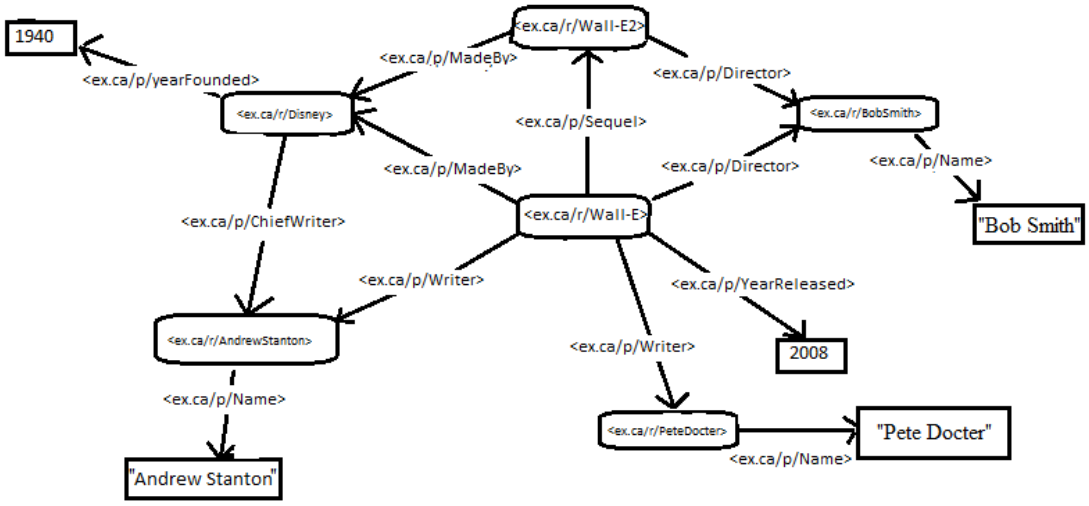


Figure 1: RDF Graph Example

Often when expressing RDF statements a shortened version of the URI is shown which use what are known as prefixes[4]. A Prefix is an alias for the namespace part of the URI string. For example, the URI "<ex.ca/r/Wall-E>" has the namespace "<ex.ca/r/>". This URI can be expressed as <ex:Wall-E> by replacing the original URI's namespace with the shortened prefix "ex". Below are some common prefixes that will be used throughout the thesis to make some URIs more readable.

PREFIX	**URI**
rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs	http://www.w3.org/2000/01/rdf-schema#
owl	http://www.w3.org/2002/07/owl#
xsd	http://www.w3.org/2001/XMLSchema#

[4] DBPedia offers a list of many different prefixes: http://eo.dbpedia.org/sparql?nsdecl (Retrieved July 19, 2017)

2.1.2 Resource Description Framework Schema

RDF schema (RDFS) is a language which expresses rules for RDF in order to give it a structure [62]. RDF schema allows for a global definition of vocabularies which define objects in the real world, and specifies the kind of objects for which these vocabularies can be applied [40]. Essentially, RDF schema provides a mechanism for RDF data to be recorded in a way that is globally interchangeable.

In the globally interchangeable setting (linked open data), RDF and RDF schema are considered to operate under the open world assumption. This means that any RDF triple can be added to an RDF dataset regardless of the RDF schema or existing RDF triples in the dataset. In practice however, when storing RDF data in specialized storage software, different implementations can operate under either the open or closed world assumption. If the RDF storage system operates under the open world assumption then any RDF data can be added to the system whether it conflicts with the RDF schema or not. This approach is useful when integration between many different RDF datasets is required. If the RDF storage system operates under the closed world assumption then only RDF data that conforms to the RDF schema is permitted to be stored. This approach can be useful when the user has a strict set of definitions which must be followed.

The features of RDFS are listed in Table 1. The RDF schema has two main features: classes and properties. Several other features like domain, range and label are used to further define classes and properties. Each RDFS feature is expressed by a triple statement. These RDFS triple statements are stored within an RDF dataset, thus becoming an extension to its RDF graph.

Table 1: RDFS List of Features and Their Descriptions

RDFS feature	Description
rdfs:Class	declares a type of thing
rdf:Property	declares an attribute
rdfs:domain	assigns a property to a class
rdfs:range	declares the datatype for a property
rdfs:subClassOf	declares a subClass of a class which inherits all of the super classes attributes.
rdfs:subPropertyOf	used to define that property is also related by another property
rdfs:label	used to give a human-readable version to resources name
rdfs:comment	used to provide a description of a resource

The example RDF dataset presented in the previous section (2.1.1) is continued here by adding to it an RDF schema. This continued example will illustrate how the RDFS triple statements are used.

Similar to object oriented programming, an RDF class is used to define a type of entity or category of object. In the RDF schema, a class is defined using the triple statement shown below. In the corresponding example there exists an RDF class called "Movie" which will be used to describe all the data about movies being stored in the RDF dataset.

Class Definition Statement

("URI reference", rdf:type, rdfs:Class)

Example

(Movie, rdf:type, rdfs:Class)
(Company, rdf:type, rdfs:Class)
(Person, rdf:type, rdfs:Class)
(MovieWriter, rdf:type, rdfs:Class)
(MovieDirector, rdf:type, rdfs:Class)

Under the open world assumption, users can define objects that belong to more than one class. The open world assumption can introduce several problems for the dataset, including: logical inconsistencies, duplicate data, misnaming of entities, inconsistent descriptions for objects (e.g., "author" vs. "book writer" vs. "article writer"). An example of a logical inconsistency would be if you have a class called "alive" and a class called "dead", in RDFS there is nothing stopping an object from being assigned to both classes. In RDFS, there is no way to define that two classes are disjoint, but in Web Ontology Language (OWL) this is possible. Some RDF applications, depending on how they are programmed, will not be able to handle such a definition because it is logically inconsistent. Ultimately, it is up to the user to ensure that their RDFS is logically consistent.

INMDB handles several logical inconsistencies in the data automatically by enforcing some close world constraints. Duplicate triple nodes (e.g., Cat with the same URI as a dog), duplicate classes and circular class inheritance are not permitted by INMDB and are reject. Other more subtle logical inconsistencies such as misnaming of entities or inconsistent descriptions (e.g., predicates that have different names, but same meaning) cannot be corrected by INMDB and will be accepted in the database. By operating under the closed world assumption INMDB eliminates the majority of data inconsistencies. Ultimately, it is up to the user to ensure RDFS data loaded into INM is logically consistent.[5]

The "rdf:type" statement shown in the example above is also used to define that a resource is an instance of a class, as shown below. In the corresponding example the URI

[5] Note that the DBPedia dataset was largely free of data inconsistencies as discussed in section 5.2.

“Wall-E” is assigned as an instance of the “Movie” class. This means that any RDF triple with “Wall-E” as the subject is now an instance of the “Movie” class.

Type Statement

(“URI”, rdf:type, “URI class”)

Example

(“ex.ca/r/Wall-E” rdf:type, Movie)
(“ex.ca/r/Wall-E 2”, rdf:type,Movie)
(“ex.ca/r/Disney”, rdf:type,Company)
(“ex.ca/r/AndrewStanton”, rdf:type, movieWriter)
(“ex.ca/r/PeteDocter”, rdf:type, movieWriter)
(“ex.ca/r/BobSmith”, rdf:type, movieDirector)

In RDFS classes have property inheritance just like in object oriented programming, where sub-classes inherit all the attributes of the super-class. The open world assumption of RDF permits RDFS classes to inherit from multiple super-classes and have multiple sub-classes, simply by adding new definitions to the RDFS dataset. As well, it permits for circular inheritance where the parent class can inherit from the child. Circular inheritance is not normally implemented in programming languages because it creates paradoxes (e.g., where one part of a code depends on another to finish, but the other part of code depends on the original code finishing).

An illustration of a scenario where multiple super-classes and sub-classes exist for a class as well as circular inheritance is illustrated in Figure 2. In this example, the class “C” inherits from two parent classes “A” and “B” and has two subclasses “X” and “Y”. Furthermore, “X” is the parent of “A” and “Y” is the parent of “B”, which introduces a circular reference.

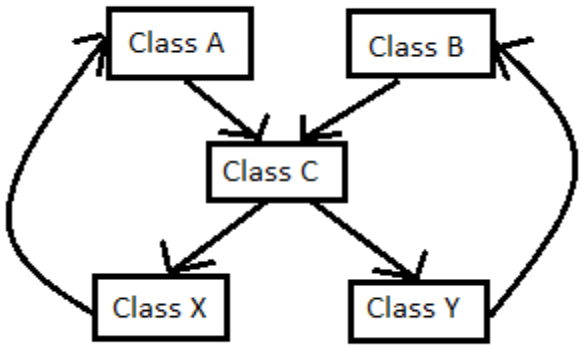


Figure 2: Multi Sub-Super-Circular Inheritance

To define sub-classes, “rdfs:SubClassOf” statements are used as shown below. In the corresponding example the class “MovieWriter” and “MovieDirector” are defined as a subclass of the Person class.

SubClassOf Statement

(“URI class reference”, rdfs:SubClassOf, “URI class reference”)

Example

(MovieWriter, rdfs:SubClassOf, Person)
(MovieDirector, rdfs:SubClassOf, Person)

RDFS properties are used to describe a relation between subjects and objects. Shown below is how to define a property in RDFS.

Property Statement

(“Predicate URI reference”, rdf:type, rdf:Property)

Example

(ex.ca/p/Director, rdf:type, rdf:Property)
(ex.ca/p/Writer, rdf:type, rdf:Property)
(ex.ca/p/YearReleased, rdf:type, rdf:Property)
(ex.ca/p/Madeby, rdf:type, rdf:Property)
(ex.ca/p/Sequel, rdf:type, rdf:Property)
(ex.ca/p/YearFounded, rdf:type, rdf:Property)
(ex.ca/p/ChiefWriter, rdf:type, rdf:Property)
(ex.ca/p/Name, rdf:type, rdf:Property)

RDF properties can have sub-properties. A sub-property indicates that a particular property is also another type of property. The sub-property statement is shown below. For example, the property “ChiefWriter” has the super property “writer”, meaning that all “chiefWriters” are also “writers”.

SubPropertyOf Statement

(“Predicate URI reference”, rdfs:subPropertyOf, “Predicate URI reference”)

Example

(ex.ca/p/ChiefWriter, rdfs:subPropertyOf, ex.ca/p/Writer)

RDFS properties can be assigned to one or more classes using a domain statement. The domain statement is shown below. In the corresponding example the properties are assigned to the classes defined above.

Domain Statement

("Predicate URI reference", rdfs:domain, "URI reference")

Example

(ex.ca/p/Director, rdfs:domain, Movie)
(ex.ca/p/Writer, rdfs:domain, Movie)
(ex.ca/p/YearReleased, rdfs:domain, Movie)
(ex.ca/p/Sequel, rdfs:domain, Movie)
(ex.ca/p/Madeby, rdfs:domain, Movie)
(ex.ca/p/YearFounded, rdfs:domain, Company)
(ex.ca/p/ChiefWriter, rdfs:domain, Company)
(ex.ca/p/Name, rdfs:domain, Person)

RDFS properties' datatype is defined by the range statement shown below. A range for a property can be one of several different datatypes such as a string, integer or Date. In the example, the property "YearReleased" is defined as an integer. The range can also be a class. As shown in the example, the property "madeBy" refers to objects of class "company". These types of relationships are also known as subject-object relationships or object-mappings, because we have a triple's object which is referring to another triple's subject.

Range Statement

("Predicate URI reference", rdfs:range, "RDF class URI reference")

Example

(ex.ca/p/Director, rdfs:range, movieDirector)
(ex.ca/p/Writer, rdfs:range, movieWriter)
(ex.ca/p/YearReleased, rdfs:range, xsd:integer)
(ex.ca/p/Sequel, rdfs:range, Movie)
(ex.ca/p/Madeby, rdfs:range, Company)
(ex.ca/p/YearFounded, rdfs:range, xsd:integer)
(ex.ca/p/ChiefWriter, rdfs:range, movieWriter)
(ex.ca/p/Name, rdfs:range, string)

2.1.3 Web Ontology Language

Web ontology language (OWL) is semantic markup language for publishing and sharing ontologies on the World Wide Web [63]. OWL has similar syntax to RDFS. It provides all the features of RDFS such as classes, properties, sub-properties and sub-classes; but it also provides an extension on these features which allow for more rigid definitions of ontologies. OWL incorporates RDFS into its vocabulary, that is to say RDFS can be used in conjunction with OWL.

OWL has many features, some of which are shown in Table 2. The full OWL vocabulary is available online[6]. OWL and RDFS have some features that are semantically equivalent, which are shown in Table 3. In practice, OWL features are expressed as RDF triple statements as shown in Table 4. Overall, the features of OWL allow for stronger ontological definitions than what only using RDFS permits.

Table 2: List of OWL Features and Their Descriptions

OWL feature	Description
owl:Class	declares a type of thing
owl:Property	declares an attribute
rdfs:Domain	Assigns a property to a class
rdfs:Range	declares the datatype for a property
rdfs:subClassOf	declares a subClass of a class which inherits all of the super classes attributes.
rdfs:subPropertyOf	used to define that property is also related by another property
owl:equivalentClasses	Means that two classes are semantically equivalent.
owl:equivalentProperty	Means that two properties are semantically equivalent
owl:disjointClasses	Used to indicate that one class specifically excludes membership from another class.
Owl:PropertyAssertion	Used to define that an instance of a class has a particular property.

[6] OWL Structural Specification: https://www.w3.org/TR/owl2-syntax/ (retrieved Aug 01, 2017)

Owl:NegativePropertyAssertion	Used to define that an instance of a class does not have a particular property.
Owl:DifferentIndividuals	Used to define that an instance is not the same as another instance.
Owl:SameIndividual	Used to define that an instance is the same as another instance.

Table 3: RDFS-OWL Equivalencies

Equivalent RDFS feature	Equivalent OWL feature
rdfs:Class	owl:Class
rdf:Property	owl:Property
rdfs:Resource	owl:Thing

Table 4: OWL Feature Triple Statements

OWL feature	RDF expression
Class	classA rdf:type owl:Class
Property	propertyA rdf:type owl:Property
Domain	propertyA rdfs:domain classA
Range	propertyA rdfs:range rangeValue
subClassOf	classB rdfs:subClassOf classA
subPropertyOf	propertyB rdfs:subPropertyOf propertyA
EquivalentClasses	classA owl:equivalentClass classC
EquivalentProperty	propertyA owl:equivalentProperty propertyC
disjointClasses	classA owl:disjointWith classD
PropertyAssertion	instance1 propertyA instance2
NegativePropertyAssertion	blankNode1 rdf:type owl:NegativePropertyAssertion . blankNode1 owl:sourceIndividual instance1 . blankNode1 owl:assertionProperty propertyA . blankNode1 owl:targetIndividual instance3 .
DifferentIndividuals	instance1 owl:differentFrom instance2
SameIndividual	instance1 owl:sameAs instance4

To further illustrate how OWL can be used, the RDFS given in the previous section is expressed in OWL shown below.

Classes
(Movie, rdf:type, owl:Class)
(Company, rdf:type, owl:Class)
(Person, rdf:type, owl:Class)
(MovieWriter, rdf:type, owl:Class)
(MovieDirector, rdf:type, owl:Class)

Disjoint Classes

(Movie owl:disjoint Company)
(Movie owl:disjoint Person)
(Company owl:disjoint Person)

Sub-classes
(MovieWriter, rdfs:subClassOf, Person)
(MovieDirector, rdfs:subClassOf, Person)

Properties
(ex.ca/p/Director, rdf:type, owl:Property)
(ex.ca/p/Writer, rdf:type, owl:Property)
(ex.ca/p/YearReleased, rdf:type, owl:Property)
(ex.ca/p/MadeBy, rdf:type, owl:Property)
(ex.ca/p/Sequel, rdf:type, owl:Property)
(ex.ca/p/Yearfounded, rdf:type, owl:Property)
(ex.ca/p/ChiefWriter, rdf:type, owl:Property)
(ex.ca/p/Name, rdf:type, owl:Property)

(ex.com/p/Director, rdf:type, owl:Property)
(ex.com/p/Writer, rdf:type, owl:Property)
(ex.com/p/YearReleased, rdf:type, owl:Property)
(ex.com/p/MadeBy, rdf:type, owl:Property)
(ex.com/p/Sequel, rdf:type, owl:Property)

Equivalent Properties
(ex.com/p/Director, owl:equivalentProperty, ex.ca/p/Director)
(ex.com/p/Writer, owl:equivalentProperty, ex.ca/p/Writer)
(ex.com/p/YearReleased, owl:equivalentProperty, ex.ca/p/YearReleased)
(ex.com/p/MadeBy, owl:equivalentProperty, ex.ca/p/MadyBy)
(ex.com/p/Sequel, owl:equivalentProperty, ex.ca/p/Sequel)

Sub-Properties
(ex.ca/p/ChiefWriter, rdfs:subPropertyOf, ex.ca/p/Writer)

Domain
(ex.ca/p/Director, rdfs:domain, Movie)
(ex.ca/p/Writer, rdfs:domain, Movie)
(ex.ca/p/YearReleased, rdfs:domain, Movie)
(ex.ca/p/Sequel, rdfs:domain, Movie)
(ex.ca/p/MadeBy, rdfs:domain, Movie)
(ex.ca/p/Yearfounded, rdfs:domain, Company)
(ex.ca/p/ChiefWriter, rdfs:domain, Company)
(ex.ca/p/Name, rdfs:domain, Person)

Range
(ex.ca/p/Director, rdfs:range, MovieDirector)
(ex.ca/p/Writer, rdfs:range, MovieWriter)
(ex.ca/p/YearReleased, rdfs:range, xsd:integer)
(ex.ca/p/Sequel, rdfs:range, Movie)
(ex.ca/p/MadeBy, rdfs:range, Company)

(ex.ca/p/Yearfounded, rdfs:range, xsd:integer)
(ex.ca/p/ChiefWriter, rdfs:range, MovieWriter)
(ex.ca/p/Name, rdfs:range, xsd:string)

2.1.4 Querying RDF and SPARQL

RDF datasets can be considered to have three different levels of abstraction [40]: the syntactic level, the structural level and the semantic level. At each of these levels the RDF data can be queried.

At the syntactic level RDF data takes the form of embedded RDF which can reside in website HTML, XML, and text documents. Web crawlers can be deployed by search engines to search for the embedded RDF tags. Once the RDF data is retrieved it can then be parsed. Additionally, algorithms can be used to sift through these embedded RDF tags and convert this data into triples. The triples can then be stored as a structured format in RDF datastores for future access. An example of embedded RDF is shown below.

RDF/a snippet for website "http://www.wall-e.example.ca/"

```
<span vocab="http://schema.org/" typeof="Article">
  <a property="url" href="http://www.wall-e.example.ca/">
     <span property="name">Blog about Wall-E</span></a>.
</span>
```

Equivalent set of triples

Subject	Predicate	Object
http://www.wall-e.example.ca/	Type	Article
http://www.wall-e.example.ca/	url	http://www.wall-e.example.ca/
http://www.wall-e.example.ca/	name	Blog about Wall-E

At the structural level RDF is represented as structured data in an RDF datastore. This is queried directly using the RDF datastore's structural query language. For example if the RDF triples are stored in a relational database, we would use SQL to query these triples directly. RDF applications can query the RDF triples at the structural level in order to interpret the RDF data and generate its graphical representation.

At the semantic level, RDF and its schema are represented as a graph. Several languages have been specially developed to query the RDF graph, such as a SPARQL [10,26,27,64], SquishQL [65], SeRQL [66] and RQL [67]. Efforts have been made to compare the differences between these RDF query languages [68]. The syntax of these languages is used to describe the semantic graph patterns that the user would like to query. The languages are designed to be written in a way where the user describes what graph pattern they would like to be retrieved, leaving the RDF database system to figure out how to retrieve the results.

At the backend of these graph query languages, the RDF storage system will translate the graph query into an equivalent structural query, which is then optimized and executed on the RDF datastore. For example with relational-backed RDF datastores, a SPARQL query is translated into SPARQL algebra, then Datalog and then into SQL which is then optimized by the relational database engine, executed and the results are returned.

SPARQL is the standard RDF query language first published by W3C in 2009 and has two major versions: SPARQL 1.0 [26] and SPARQL 1.1 [64]. Version 1.1 has more features which allows for more graph query expressions. The examples given for

SPARQL in this thesis and the exploration of SPARQL compatibility with INMDB will use SPARQL Version 1.1.

There are four types of SPARQL query: SELECT, CONSTRUCT, ASK and DESCRIBE. The SELECT query is the most common type of query, it used for extracting RDF data from the RDF storage system. The CONSTRUCT query returns an RDF graph, which is used for generating graph results. The ASK query is similar to a SELECT query, however it only returns either true or false, whether the query pattern matches or not. The DESCRIBE query returns an RDF graph describing some features of a specified resource.

SPARQL queries have their own syntax, which has four main components: literal values, query variables, blank nodes, triple patterns. The simplest component of a triple pattern is what is known as a basic graph pattern (BGP). BGPs are used to match triples in the RDF graph, whereas triple patterns are used to match graph patterns in the RDF graph. These components are described in Table 5.

Table 5: SPARQL Syntax Components

SPARQL syntax	Explanation and Example
Literal Value	A string, e.g., "<ex.ca/r/Wall-E>"
Query Variable	A variable is prefixed with "?" or "$" and has global scope, e.g., ?movie
Blank Node	Blank nodes are anonymous variables, similar to the ones in Domain Calculus (the "_" variable). Blank nodes in SPARQL begin with an underscore "_", e.g., _:abc
Basic Graph Pattern (BGP)	A basic graph pattern is used to match one or more triples. A basic graph pattern has three parts, they are: subject, predicate and object. These parts are separated by white space " " and terminated by a period "." e.g., {"<ex.ca/r/Wall-E>" _:abc ?x .}

Triple Pattern	A triple pattern is composed of one or more BGPs. It is used to match parts of the RDF graph patterns. e.g., {"<ex.ca/r/Wall-E>" ?p1 ?o . ?o ?p2 ?o2 . ?o2 ?p3 ?o3 }

SPARQL 1.1 has many different query operators which are shown in Table 6.

Table 6: SPARQL Query Features

SPARQL operator	Description / Function
BGP belonging to a Class and it's subclasses	Retrieve a set of triples belonging to an RDFS class
UNION of two BGP	Union two sets of triples.
JOIN	Perform a join on two sets of triples.
OPTIONAL JOIN	Perform an optional join (left outer join) on two sets of triples.
FILTER	Filter a set of triples. This is equivalent to an SQL "where" statement.
BIND	Used to set a value for a variable. Similar to SQL's select statement operator "as". e.g., Select a + b as c From table1.
MINUS	Negate one set of triples from another, equivalent to SQL's negation operator
SUBQUERY	Query a subset of triples. (i.e., A query within a query)
GROUP BY	Same as GROUP BY in SQL, supporting aggregate functions such as sum, count, average, etc.
ORDER BY	Same as ORDER BY in SQL, orders a set of triples.
OFFSET and LIMIT	Same as Offset and Limit found in SQL. This sets an offset and limit for the number of triples to be returned.

A SPARQL SELECT query has two main clauses which are required to write a query: SELECT clause and WHERE clause. The SELECT clause is similar to the SQL select clause, where variables that you wish to project are selected. The WHERE clause is used to specify the graph pattern to be matched. A sample SELECT query is shown in below. The query shown below asks a simple subject-subject join style query. The query

asks what the subject and year released for movies that have a sequel called “Wall-E 2”. The query joins together triples where “?movie” share the same value.

SPARQL SELECT Query

```
SELECT ?movie
       ?yearReleased
WHERE
  {
    ?movie   rdf:Type     Movie .
    ?movie   <ex.ca/p/sequel>       <ex.ca/r/Wall-E2> .
    ?movie   <ex.ca/p/yearReleased> ?yearReleased .
  }
```

Results of SELECT Query

?movie	?yearReleased
<ex.ca/r/Wall-E>	2008

A SPARQL CONSTRUCT query is composed of two clauses: CONSTRUCT clause and WHERE clause. The CONSTRUCT clauses describes a graph pattern that we wish to construct and the WHERE clause specifies a graph pattern to fetch the data which will be used to construct the new graph with. An example CONSTRUCT query is shown below. In this example we construct a new RDF graph of movies with a new “movie writer” predicate, which is output as set of triples shown in the results.

SPARQL CONSTRUCT Query

```
CONSTRUCT { ?movie <new.ex.ca/p/MovieWriter> ?writer }
WHERE
  {
    ?movie   rdf:Type     Movie .
    ?movie   <ex.ca/p/Writer> ?writer .
  }
```

Results of CONSTRUCT Query

(<ex.ca/r/Wall-E>, <new.ex.ca/p/MovieWriter>, <ex.ca/r/AndrewStanton>)
(<ex.ca/r/Wall-E>, <new.ex.ca/p/MovieWriter>, <ex.ca/r/PeteDocter>)

A SPARQL ASK query is composed only of the ASK clause. An example is shown below. This query asks whether there is a movie that was released in 2008. The answer to this query is Boolean value true.

SPARQL ASK Query

```
ASK
 {
   ?movie rdf:Type Movie .
   ?movie <ex.ca/p/yearReleased> "2008".
 }
```

Result of ASK Query

True

A SPARQL DESCRIBE query is composed of two clauses: DESCRIBE clause and WHERE clause. The DESCRIBE clause is used to specify what should be described by the RDF storage system. The WHERE clause specifies a graph pattern which filters the results from which we can describe the data. Depending on the implementation of the SPARQL engine for an RDF store, the DESCRIBE query will return different results. An example DESCRIBE query is shown below along with its output, which describes movies made by Disney.

SPARQL DESCRIBE Query

```
DESCRIBE ?movie
WHERE
  {
    ?movie   rdf:Type     Movie .
    ?movie   <ex.ca/p/Madeby> "<ex.ca/r/Disney>" .
  }
```

Result of DESCRIBE Query

```
<ex.ca/r/Wall-E> a Movie ;
     <ex.ca/p/Director> <ex.ca/r/BobSmith>;
     <ex.ca/p/Sequel> <ex.ca/r/Wall-E2> .
<ex.ca/r/Wall-E2> a Movie ;
```

<ex.ca/p/Director> <ex.ca/r/BobSmith> .

2.1.5 SPARQL Algebra

SPARQL algebra defines the semantics for a SPARQL queries execution [64]. This expression is generated through a transformation process which takes as input a SPARQL query and RDF graph, of which the steps are described in below. SPARQL algebra has several operators which are abstract commands for operations to be performed when processing the query. SPARQL algebra does not determine how the query will be executed on the RDF datastore, but provides a semantic query execution plan. A list of SPARQL algebra operators and their semantics are shown in Table 7. [7]

Table 7: SPARQL Algebra Operator Semantics

SPARQL algebra operator	Example Syntax	Semantics of operator
BGP	BGP (?a ?b ?c)	BGP is acronym for Basic Graph Pattern. A basic graph pattern is the simplest graph pattern that you can generate in RDF. BGP returns a set of triples from a graph G which matches the given triple pattern.
FILTER	FILTER(?a > 0, BGP(?a ?b ?c))	Returns a set of triples restricted by a given expression. This is the same as a where clause in SQL.
JOIN	JOIN(BGP(?a ?b ?c), BGP(?a ?x ?y))	Merges two sets of triples.
Diff / MINUS	MINUS (BGP(?a ?b ?c), BGP(?a ?x ?y))	Substracts one set of triples from another set. Equivalent to safe negation found in DataLog or SQL.
LEFTJOIN	LEFTJOIN(BGP(?a ?b ?c), BGP(?a ?x ?y), true)	Left outer join of two sets of triples.

[7] Note: The SPARQL algebra operators shown in Table 7 are defined by the W3C recommendations for SPARQL 1.0 and 1.1 [38, 116].

UNION	UNION(BGP(?a ?b ?c) , BGP(?a ?x ?y))	Union two sets of triples.
TOLIST	TOLIST (BGP(?a ?b ?c))	Turns a set of triples into a sequence with the same elements and cardinality.
ORDER BY	ORDERBY(BGP(?a ?b ?c), ?a Descending)	Orders a sequence triples.
PROJECT	PROJECT(BGP(?a ?b ?c), ?a)	Restricts the variables in a set of triples. Same as projection in relational algebra.
DISTINCT	DISTINCT(BGP(?a ?b ?c))	Returns a list of triples with only unique values. Similar to Distinct found in SQL.
SLICE	SLICE(BGP(?a ?b ?c), Start, Length)	Returns a subset of triples from position "start" to "start + length" from a set of triples.

Several steps are used to translate SPARQL into SPARQL algebra. Each step is applied recursively to each syntactic element in the SPARQL graph pattern. The result of this step is a SPARQL algebra expression which is composed of SPARQL algebra operators. The steps to translate a SPARQL query into SPARQL algebra are as follows:

1) **Expand syntax forms for IRIs literals and triple patterns**

   Expand prefixes for IRIs.

   e.g.,"<ex:Wall-E>" → "<ex.ca/r/Wall-E>"

2) **Translate property path expressions**

   This converts path expressions such as "*", "+", "/", "a" into a SPARQL algrebra statement .

   e.g., P / Q → seq(P, Q)

3) **Convert some property path patterns to triples**

   This converts property path patterns into their expanded triple form

   e.g., X seq(P, Q) Y → X P V1 . V1 Q Y .

4) **Collect the FILTERS in the group**

   Remove all filter expressions from groups and save them for later

5) **Translate basic graph patterns**

Adjacent triple patterns are collected together to form a basic graph pattern.

e.g., {X P V1 .} → BGP(X P V1)

6) **Translate remaining graph patterns in the group**

This translates each remaining graph pattern with join or union recursively: GroupOrUnionGraphPattern, GraphGraphPattern, GroupGraphPattern, Subselect.

7) **Add filters**

Applies the filter expressions here so they apply to whole expression

8) **Simplify the algebra expression**

This step replaces graph patterns that are empty.

Some examples of SPARQL graph patterns translated into SPARQL algebra are shown in Table 8. At an abstract level a SPARQL algebra expression is a tree, as illustrated in Figure 3.

Table 8: SPARQL to Algebra Examples

SPARQL Graph Pattern	SPARQL algebra
{?s ?p ?o}	BGP(?s ?p ?p)
{ ?s :p1 ?v1 ; :p2 ?v2 }	JOIN(BGP(?s :p1 ?v1) , BGP(?s :p2 ?v2))
{ { ?s :p1 ?v1 } UNION {?s :p2 ?v2 } }	UNION(BGP(?s :p1 ?v1) , BGP(?s :p2 ?v2))
{ { ?s :p1 ?v1 } UNION {?s :p2 ?v2 } UNION {?s :p3 ?v3 } }	UNION(UNION(BGP(?s :p1 ?v1) , BGP(?s :p2 ?v2), BGP(?s :p3 ?v3))
{ ?s :p1 ?v1 OPTIONAL {?s :p2 ?v2 } }	LEFTJOIN(BGP(?s :p1 ?v1), BGP(?s :p2 ?v2), true)
{ ?s :p1 ?v1 OPTIONAL {?s :p2 ?v2 FILTER(?v1<3) } }	LEFTJOIN(BGP(?s :p1 ?v1) , BGP(?s :p2 ?v2) , (?v1<3))
{ ?s :p ?v . MINUS {?s :p1 ?v2 } }	MINUS(BGP(?s :p ?v) BGP(?s :p1 ?v2))
{ ?s :p ?o . {SELECT DISTINCT ?o {?o ?p ?z} } }	JOIN(BGP(?s :p ?o) , ToMultiSet(DISTINCT(PROJECT(BGP(?o ?p ?z),

	{?o}))))

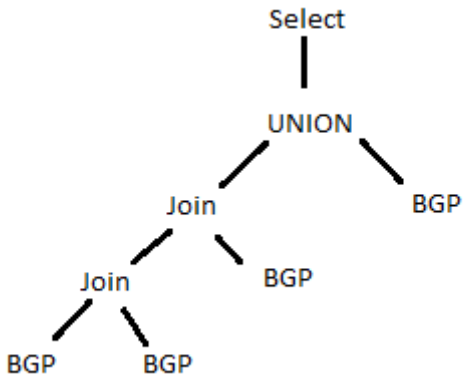


Figure 3: SPARQL Algebra Tree Example

2.1.6 Applications of Semantic Web

Many applications use RDF data in the real world. A quick search on internet yields numerous results of different products which incorporate RDF into their works. Here are a few that are well known in the semantic web community: Dublin Core Metadata Initiative [9] is a metadata framework which aims to provide a minimal set of descriptive elements for describing and indexing web documents found on the web. PRISM (Publishing Requirements for Industry Standard Metadata) [2] is a metadata framework used by the publishing industry which enables publishers to publish their creative content in many different ways (web, book, email, etc) more easily. A very popular application using RDF data that many people in the field of computer science have used is RSS 2.0[8] [69]. RSS is an acronym for "RDF Site Summary", it is a web feed

[8] RSS 2.0 - http://cyber.harvard.edu/rss/rss.html
(retrieved Aug 01, 2017)

service which is used to publish frequently updated information such as blog entries, news headlines, audio documents and video documents. The structured metadata frameworks SNOMED RT (Systemized Nomenclature of Medicine Reference Terminology) [3] and MeSH (Medical Subject Headings) [4] have important roles in the medical field by aiding in the distribution of medical knowledge. Google's web crawlers read and understand an HTML version of RDF data called RDFa [70], which is essentially RDF triples embedded into HTML code that renders an HTML page machine readable.

Many user interface applications exist for visualizing and navigating RDF data. Some examples are: LODWheel [71], Tabulator [72], Haystack [73], Swoop [74], Piggy-Bank [75], Longwell[9], IsaViz[10] or Noadster[11]. These browsers enable users to query and visualize the data in an RDF datastore. For example, if we have an RDF datastore which contains metadata about movies (writer, year released, publisher, title, etc.), users can visualize such questions as "How many movies has a particular author written?", "What are all the movies that a certain publisher has released?", "What year was the last movie written by a particular author?", "What is the sequel to a particular movie?", among many others. Additionally some of these RDF browsers allow users to visualize links between RDF nodes and to generate graphs or charts to better understand the RDF data.

[9] Simile Longwell - https://www.w3.org/2001/sw/wiki/Longwell
(retrieved Aug 01, 2017)
[10] IsaViz - https://www.w3.org/2001/11/IsaViz/
(retrieved Aug 01, 2017)
[11] Noadster - http://homepages.cwi.nl/~media/demo/noadster/
(retrieved Aug 01, 2017)

2.1.7 Working Example RDF Dataset

For the remainder of the thesis all illustrations of RDF storage techniques and query answering will use the RDF dataset shown below. The corresponding RDFS for this dataset is also shown below along with the class instance definitions. This sample RDF dataset and RDF schema is the "Wall-E" example that was introduced in sections 2.1.1 and 2.1.2.

The working example is summarized to provide the reader with an easy reference. It includes the different features found in RDFS and several RDF triples, making it representative of real world RDF datasets and ideal for illustrative purposes.

RDF triples
(<ex.ca/r/Wall-E>, <ex.ca/p/Director>, <ex.ca/r/BobSmith>)
(<ex.ca/r/Wall-E>, <ex.ca/p/Writer>, <ex.ca/r/AndrewStanton>)
(<ex.ca/r/Wall-E>, <ex.ca/p/Writer>, <ex.ca/r/PeteDocter>)
(<ex.ca/r/Wall-E>, <ex.ca/p/YearReleased>, 2008)
(<ex.ca/r/Wall-E>, <ex.ca/p/Sequel>, <ex.ca/r/Wall-E 2>)
(<ex.ca/r/Wall-E 2>, <ex.ca/p/Director>, <ex.ca/r/AndrewStanton>)
(<ex.ca/r/Wall-E>, <ex.ca/p/Madeby>, <ex.ca/r/Disney>)
(<ex.ca/r/Wall-E 2>, <ex.ca/p/Madeby>, <ex.ca/r/Disney>)
(<ex.ca/r/Disney>, <ex.ca/p/Yearfounded>, 1940)
(<ex.ca/r/Disney>, <ex.ca/p/ChiefWriter>, <ex.ca/r/AndrewStanton>)
(<ex.ca/r/AndrewStanton>, <ex.ca/p/Name>, "Andrew Stanton")
(<ex.ca/r/PeteDocter>, <ex.ca/p/Name>, "Pete Docter")
(<ex.ca/r/BobSmith>, <ex.ca/p/Name>, "Bob Smith")

classes
(Movie, rdf:type, rdfs:Class)
(Company, rdf:type, rdfs:Class)
(Person, rdf:type, rdfs:Class)
(MovieWriter, rdf:type, rdfs:Class)
(MovieDirector, rdf:type, rdfs:Class)

subClasses
(MovieWriter, rdfs:subClassOf, Person)
(MovieDirector, rdfs:subClassOf, Person)

Properties
(ex.ca/p/Director, rdf:type, rdf:Property)
(ex.ca/p/Writer, rdf:type, rdf:Property)

(ex.ca/p/YearReleased, rdf:type, rdf:Property)
(ex.ca/p/MadeBy, rdf:type, rdf:Property)
(ex.ca/p/Sequel, rdf:type, rdf:Property)
(ex.ca/p/Yearfounded, rdf:type, rdf:Property)
(ex.ca/p/ChiefWriter, rdf:type, rdf:Property)
(ex.ca/p/Name, rdf:type, rdf:Property)

subProperties
(ex.ca/p/ChiefWriter, rdfs:subPropertyOf, ex.ca/p/Writer)

domain
(ex.ca/p/Director, rdfs:domain, Movie)
(ex.ca/p/Writer, rdfs:domain, Movie)
(ex.ca/p/YearReleased, rdfs:domain, Movie)
(ex.ca/p/Sequel, rdfs:domain, Movie)
(ex.ca/p/MadeBy, rdfs:domain, Movie)
(ex.ca/p/Yearfounded, rdfs:domain, Company)
(ex.ca/p/ChiefWriter, rdfs:domain, Company)
(ex.ca/p/Name, rdfs:domain, Person)

range
(ex.ca/p/Director, rdfs:range, MovieDirector)
(ex.ca/p/Writer, rdfs:range, MovieWriter)
(ex.ca/p/YearReleased, rdfs:range, xsd:integer)
(ex.ca/p/Sequel, rdfs:range, Movie)
(ex.ca/p/MadeBy, rdfs:range, Company)
(ex.ca/p/Yearfounded, rdfs:range, xsd:integer)
(ex.ca/p/ChiefWriter, rdfs:range, MovieWriter)
(ex.ca/p/Name, rdfs:range, xsd:string)

class instance definitions
(“ex.ca/r/Wall-E” rdf:type, Movie)
(“ex.ca/r/Wall-E 2”, rdf:type,Movie)
(“ex.ca/r/Disney”, rdf:type,Company)
(“ex.ca/r/AndrewStanton”, rdf:type, movieWriter)
(“ex.ca/r/PeteDocter”, rdf:type, movieWriter)
(“ex.ca/r/BobSmith”, rdf:type, movieDirector)

2.2 Information Network Model Database

The problem with existing RDF storage approaches is that they do not fully represent RDFS as database schema, partially due the limitations of their data models.

With native approaches and the relational vertical and binary table approach, there is no representation of the RDFS as database schema. The relational N-ary approach is significantly better at representing RDFS as database schema than these approaches; however, the relational storage model is not able to represent property inheritance, sub-attributes, references, or multi-valued attributes. As of this writing, the N-ary object-relational and expert system approaches are the best attempts to represent RDF schema database schema by offering multi-valued attribute support, property inheritance and subject-object representations using object references. However, existing object-relational and object-oriented data models have focused mainly on the representation of the object itself, such as property inheritance and attribute type, but they do not adequately model relationships that exist in the real world as well as those that are found in the RDF data model.

Information Network Model (INM) is a data model invented to better model the various simple and complex relationships among real-world entities [61]. The core representation of objects in INM is similar to what is found in other object-oriented and object-relational models. INM supports classes with property inheritance (multiple super-classes and multiple sub-classes). Furthermore, it supports multi-valued attributes which can have sub-attributes. Attributes are represented with basic data types such as: Dates, text, urls, strings, floats or integers. The main novelty of INM that distinguishes it is that it has eight kinds of semantic relationships that each have different behaviours: normal, contain, role, composite, complex, context-based, role-based, and nary. Furthermore, some of these relationships can have sub-relationships and can be combined together to represent real-world organizational structures. In this thesis we only introduce the

relationships that are needed for representing RDFS. Figure 4 shows how classes can have attributes with sub-attributes and relationships with sub-relationships in INM.

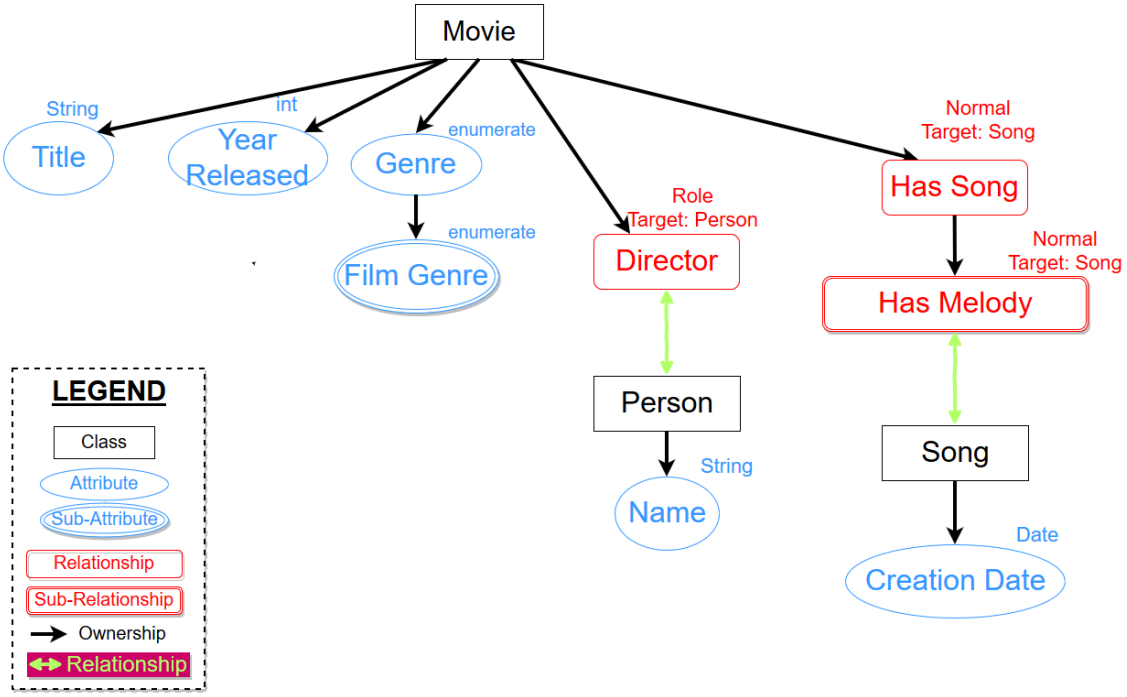


Figure 4: INM Model Illustration

The RDF/RDFS data model is relationship rich and has several features such as classes, property inheritance and properties that can have sub-properties. INM is therefore a suitable database model for representing the various features of RDFS and the relationships between RDF triples. Sub-attributes, sub-relationships and property inheritance in INM enables the semantic representation of RDFS:subPropertyOf and RDFS:subClassOf definitions found in the RDFS data model. INM relationships enable the semantic representation of RDFS property object-mappings (subject-object relationships). Finally, multi-valued attributes/relationships enable the semantic representation of the multi-valued nature of RDFS properties.

INMDB is a database management system which implements the Information Network Model. It therefore supports all of the INM features such property inheritance, sub-attributes, multi-valued attributes and multi-valued relationships, and sub-relationships.

In INMDB relationships to other objects are referred to as “linking”. There are four main differences between linking in INMDB and references found in OODBs and OORDBs. The first difference is that the value stored for a link is not a system generated ID, it is the name of the object specified by the user. This gives superior semantic representation for RDFS relationships when using triple subjects as object IDs. Second, the link is able to point to more than one different class. Third, INM supports many different types of links such as role, context and normal. Fourth, links support sub-links allowing us to model RDFS:SubPropertyOf definitions found in the RDFS data model for RDF object mappings.

References found in OODB and OORDB systems are cumbersome and difficult to work with. This is largely due to the fact that a lookup in the database must be carried out prior to setting the references value for an object because there is no direct way to access the reference value. This process of referencing must be implemented with a mechanism outside the database management system and is often inefficient. In contrast, INMDB requires no look up when inserting RDF relationships because the triple’s subject serves as the reference ID.

Linking is a semantic way of representing relationships. Storing the relationships between data in a semantic way can improve the performance of queries. For example, almost all RDF queries involve making joins on subject-subject and subject-object

relationships. In the relational N-ary model, for subject-object relationships there is no way to know where the related triple is stored, because they could potentially be stored in one of several different tables (for example in an RDFS class table or any one of its subclass tables). Therefore, in order to complete a query requesting a subject-object join it requires that extraneous joins be carried out with all possible tables to identify where the data resides. In INM however, the link contains information which can be used to locate the related object, eliminating the need for extraneous join computations.

INM uses three different languages: a data definition language IDL, a data manipulation language IML and a query language IQL. IDL is used to define the class structures of an INMDB instance; it is introduced in section 2.2.1. IML is used to insert, delete or change data for INMDB instance; it is introduced in section 2.2.2. IQL is INMDB's query language used for exploring the graph structure of objects in an INMDB instance, returning query results in a concise and compact way; IQL is introduced in section 2.2.3.

2.2.1 IDL – Data Definition Language

IDL is the data definition language used for defining the class structure for an INMDB instance. To introduce this, an example of a database storing movies similar to the running RDFS example is described below and the IDL statement used to create INMDB classes is shown below.

The Movie class has some attributes Title, Genre, and "Year Released". The Movie class has a relationship "Made By" which is a link to the Company class, it has a relationship called "Has Song" which links to the Song class and it has a "Uses Movie

Prop for Filming" relationship which links to the Item class (Up to five items can be used when filming a movie, each appearing in only one movie). The movie class can have "Director" roles played by one or more people, which links to the "Person" class. The Director can either be a regular Director or an "award winning Director". The Movie can also have a rating given to it by several different Reviewers. The corresponding IDL statement is shown below.

IDL Statement

```
create class Movie [
  @Title:string,
  @Genre*:{"funny", "scifi", "Romance"}|[@ "Film Genre": {"funny", "scifi",
"Romance"}],
  @"Year Released":int default 1900,
  Normal "Made By" (N:1): Company (inverse Makes),
  Normal "Has Song" (1:N) -> {"Has Melody"(1:N)}: Song (inverse "featured in"),
  Normal "Uses Movie Prop for Filming" (1:5): Item (inverse "appears in")
  role Director[role_based "Won Award"*:Award(inverse Winner)] (inverse "as") (N:N):
                                                    Person (inverse Directs),
  Rating *: [@Rating:int, normal Reviewer(N:1):Person]
];
```

Table 9: IDL syntax definitions

Syntax	Defintion
create class "Class Name" […];	Class Creation Statement. Inside the square brackets "[…]"we define the attributes and relationships
@"attribute name": "type"	Attribute Statement Defines an attribute with "attribute name" as its name and "type" as its type. The type can be int, string, real or date.
@"attribute name"*: "type"	Multi-valued Attribute Statement Defines a multivalued attribute. The "*" indicates (1:N) cardinality.
@"attribute name": "type" | [@"sub-attribute name": "type"]	Sub-Attribute Statement Defines an attribute with name "attribute name" and a sub-attribute with name "sub-

	attribute name".
Normal "relationship name" ("Cardinality") : "Target Class1"(inverse "inverse relation name") | "Target Class2"(inverse "inverse relation name2")	Normal Relationship Statement Defines a normal relationship with name "attribute name" and cardinality, whose target class is "Target Class1" or "Target Class2". INMDB relationships can target an infinite number of classes. The "Normal" keyword is optional and can be omitted.
role "relationship name" ("Cardinality") : "Target Class1"(inverse "inverse relation name") | "Target Class2"(inverse "inverse relation name2")	Role Relationship Statement Defines a role relationship with name "attribute name" and cardinality, whose target class is "Target Class1" or "Target Class2". INMDB relationships can target an infinite number of classes.
role "relationship name"[…] ("Cardinality") : "Target Class2"(inverse "inverse relation name")	Role Relationship Statement with Attributes Defines a role relationship with name "attribute name" targeting "Target Class2". Within the square brackets attributes and relationships can be added.
Normal "relationship name" ("cardinality") -> {"sub-attribute name"("cardinality")}: "Target Class"	Sub-Relationship Statement The sub-relationship is defined using the symbol "->". Sub-relationships must target the same the class as the parent relationship.
"composite name": [attribute or relationship statements]	Composite Relationship Statement Defines a composite relationship composed of one or more attributes or relationships. Optionally you can use the "Normal" keyword infront of this statement.

The above IDL statement shows several features of the IDL language. Clauses starting with "@" are attribute statements. Both attribute and link names can have spaces in them by putting the name in quotations, for example "Year Released". Attributes can be one of several predefined INM data types such as int, string, url, Date, text or real. Attributes that have an asterix "*" after their name are multi-valued. Attributes can also

be of enumerate type, as shown in the example above for attribute genre. The genre can have as string value of “funny”, “scifi”, “Romance”. Attributes can also have a default value, as shown in example above for attribute “Year Released”. The attribute genre has a sub-attribute “Film Genre” which is also the same enumerate type as its parent attribute “Genre”. The relationship “Has Song” has a sub-relationship “Has Melody”, both of which target the “Song” class.

There are six kinds of links in INM: normal, contain, role, composite, complex, context and nary. Above, the link “Made By” is a normal link to the Company class. In INM, cardinality constraints can be represented either generally using the “*” which signifies a “(1:N)” cardinality or specifically using “(N:M)”. For example the “Made By” link is specified as a “(N:1)” cardinality, meaning that a Movie can be “Made By” only one Company, but a Company can make more than one movie. The “Has Song” link is a “(1:N)” cardinality, meaning that a movie can have any number of songs but each song can only appear in one movie. The “Uses Movie Prop for Filming” link is specified as having a “(1:5)” cardinality, meaning that each Item can only appear in one movie and a movie is limited to having only 5 Items used as props. The specified “N:N” of the “Director” role means that this is a multi-valued link, where a Movie can have more than one director and a Person can direct more than one Movie.

INMDB automatically generates classes that are targeted by links if they have not already been declared. For example, in the above IDL statement, the “Made By” link targets the Company class which has not been explicitly declared. INMDB automatically generates the “Company” class. As well, the Person, Item, Song and Award class will be

generated automatically. Additionally, if a class has already been declared but new attributes are introduced, INMDB will merge the classes together into one single class.

The “Director” link above is a role link that targets the “Person” class. A role relationship in INM is used to represent the dynamic role facet of an object. Here, a Person object can play different roles in Movie. For example “Georges” can be either a director or an actor. When playing a particular role he can have different “role_based” attributes or relationships, such as an award that he has won as director, or a “character name” when he is an actor. Hence, a Person object can be instances of the class “Person”, “Director” or “Actor”. Using role relationships, only one Person object is needed to capture the three different facts of the “Georges” object. When a role is defined, INM generates automatically a role class based on the definition which inherits from the link target class.

In the example above, it assigns the “role_based” link “Won Award” that targets the role class “Director”, which inherits all other attribute/relationships from the “Person” class. When the “Won Award” relationship exists in the linked Person instance, he is an “award winning director” otherwise he is just a regular director.

The link “Rating” is a composite link which links to a self-generated class that has two properties: a regular attribute and a normal link. Composite links can be very useful when a list of objects is required, for example, a list of ratings given by users.

Every link in INMDB has a unique inverse which can either be user-defined or system generated. A user-defined inverse link is defined for the Director role, where a Person object has an inverse link called “Directs” to a Movie object. The inverse is stored in the INMDB instance to permit reverse lookups of objects. For example, if the user

would like to know what movies are directed by a Person, they can directly query the inverse link instead of having to access the value through the Movie objects.

Similarly, when a role link is defined, INMDB will automatically generate an inverse context link within the role class which is derived from the corresponding role link. For example, the corresponding role class “Director” for the “Director” role link will have the inverse relationship “context directs:Movie[as:Director]” generated.

The above IDL statement shown will generate the classes in INMDB shown below.

Resulting INMDB classes

```
class Movie [
  @Title:string,
  @Genre*:{“funny”, “scifi”, “Romance”}|[@ “Film Genre”: {“funny”, “scifi”, “Romance”}],
  @”Year Released”:int default 1900,
  Normal “Made By” (N:1): Company (inverse Makes),
  Normal “Has Song” (1:N) -> {“Has Melody”(1:N)}: Song (inverse “featured in”),
  Normal “Uses Movie Prop for Filming” (1:5): Item (inverse “appears in”)
  role Director[role_based “Won Award”*:Award(inverse Winner)] (inverse “as”) (N:N):
                                                    Person (inverse Directs),
  Rating *: [@rating:int, normal reviewer(N:1):Person]
];

class Company [
  normal “Makes” (1:N) : Movie (inverse “Made By”)
];

class Song [
  normal “featured in” (N:1) : Movie (inverse “Has Song”)
];

class Item [
  normal “appears in” (5:1) : Movie (inverse “Uses Movie Prop for Filming”)
];

class Person [
  Inverse link of “Rating->reviewer” (1:N): Movie (inverse Rating->reviewer),
];

role_class Director SUBCLASS of {Person}[
  Inverse link of “Rating -> reviewer” (1:N): Movie (inverse Rating->reviewer),
```

```
  context “Directs” (N:N) : Movie (inverse “Director”) with relation “as” {
            normal “winAward”(1:N):Award (inverse “Winner)
            }
];

class Award [
  normal Winner (N:1) : Director (inverse “Won Award”)
];
```

2.2.2 IML – Data Manipulation Language

IML is the data manipulation language for INMDB used for inserting, modifying or deleting data from an INMDB instance. In this section the example of a database storing movies presented in section 2.2.1 will be used to show how to insert objects into INMDB.

Below we insert the “Person” object instance with the object id “James Cameron” as well as the “Movie” object instance with the object id “Titanic”.

IML Statements

```
insert “Person”  “James Cameron” [ ];

insert Movie Titanic[
  @"Title": “Titanic”,
  @Genre:"Romance",
  @”Year Released”:1998,
  Normal “Made By” : CompanyA,
  role Director :"James Cameron"[role_based “Won Award”: Oscar],
  Rating: {[@rating:9,normal reviewer:PersonA],
           [@rating:8,normal reviewer:PersonB]}
];
```

Table 10: IML Syntax Definitions

Syntax	Definition
insert “Class” “Object Name” […];	Insert Statement This statement is used to insert objects into the INMDB database. It inserts an object with values for attributes defined in the square

	brackets "[…]".
@ "attribute name": "Value"	Single-Valued Attribute Statement This statement inserts a single-valued attribute of type defined by the schema (e.g., int, string, state, real,…)
@ "attribute name": {"Value1", "Value2",…}	Multi-Valued Attribute Statement This statement inserts multiple values into an attribute.
Normal "relationship name": "object id"	Relationship Statement This statement inserts a relationship for "relationship name" linking to the "object id" class.
role "relationship name"[…]: "object id"	Role Relationship Statement This statement inserts a relationship for "relationship name" linking to the "object id" class. Within the square brackets we insert additional attributes for the role relationship.

Normal "composite name": {
[tuple of attributes 1], [tuple of attributes 2], …
}
	Composite Relationship Statement
	
	This statement inserts into the composite relationship tuples composed of attributes or relationships.

The attributes' values for each instance are restricted by the INMDB schema definition. The "Titanic" object instance has some basic attributes such as name, genre and "Year Released". The "Titanic" instance has a normal link "Made By" which will generate the Company object instance "CompanyA" if not already defined. The role link "Director" references the "James Cameron" Person object instance and assigns him the role "Director" is assigned to him. As "Director" he wins an "Oscar" award.

After inserting the two statements above, the INMDB system will have the object instances shown below:

INMDB Object Instances

```
Company “CompanyA” [
  Normal Makes: “Titanic”
];

Award “Oscar” [
  context “Winner”: “James Cameron”
];

Person “PersonA” [
  Inverse relationship “Rating->reviewer”: Titanic
];

Person “PersonB” [
  Inverse relationship “Rating->reviewer”: Titanic
];

Director(Movie) “James Cameron” [
  context directs:Titanic[as:Director[role_based “Won Award”:Oscar]]
];

Movie “Titanic” [
  @Genre:”Romance”,
  @Title:”Titanic”,
  @”Year Released”:1998,
  role Director : “James Cameron”,
  Rating : {[@rating:9,normal reviewer:PersonA],
            [@rating:8,normal reviewer:PersonB]}
];
```

2.2.3 IQL – Query Language

IQL is the query language that is unique to the INMDB software. It includes three types of queries: schema query, instance of query and instance query. Schema query is used for querying the schema of the database. For example, a schema query will return a list of all the classes that are stored in the database. An instance of query will query the specific information about a particular class in the database, and the instance query is used for querying object instances stored in the database. Instance queries have format shown below:

```
Query instanceQuery [constructExpression]
```

IQL instanceQuery statements use path expressions to define the conditions; this is similar to the SQL where clause of a query. Paths, which are represented in the query by "/", "//" and "[ ]" are used for filtering the data. IQL also supports paging queries for queries that are very large. A paged query allows the user to specify a range of results to be displayed. Finally, IQL supports fuzzy queries, which is equivalent to the keyword "LIKE" used in SQL. Fuzzy queries use wildcards "?" and "%" to replace strings of characters in a variable.

IQL constructExpression statements are used to display the query results. Results are displayed in a tree structure where "/" represents node parent-child relationships, "," means the nodes are juxtaposed, and "[ ]" is used to indicate that there is more than one child. Aggregates are performed in the construct expression. The aggregate functions in IQL are very similar to SQL aggregate functions and include the following descriptive statistics: min, max, average, distinct and count.

Table 11: IQL Syntax Descriptions

IQL Syntax	Defintion
query 'graph pattern' construct 'construct expression'	Query Syntax The basic query expression consists of 2 parts, the query part and the construct part. The query part is formed with the keyword "query" with a "graph pattern" expressing what operation to perform. The construct part is formed with the keyword construct and the "construct expression" which is used to define which variables to project as output.
query "graph pattern"	Yes/No Query Syntax When omitting the "construct expression" the query becomes a yes or no style query, returning either true or false if the graph pattern exists.

query ‘graph pattern’ construct $x[…]	Construct attributes of object This construct expression will list attribute and relationship values for object $x. If the square brackets are left empty (e.g., $x[ ]) then all attributes will be listed. Otherwise, we specify inside the brackets which attributes to list using their graph pattern variable (e.g., $x[$a,$b,…,$z]).
query ‘graph pattern’ construct sum({$y})	Construct aggregates There are several different aggregate functions that can be used in the construct statement. e.g., sum({$y}), count({$y}), avg({$y}) These can be nested within objects in order to “group by” the object. e.g., $x[sum({$y}), count({$y})]
query ‘graph pattern’ construct $x order by $x	Construct Order by This statement will order all objects. It can be specified as either ASC or DESC order. Order by operators can nested within other objects, when ordering attribute values for an outputted object.
query ‘graph pattern’ construct $x top(“start”,”length”)	Construct offset/limit The top statement will only output “length” number of results starting at position “start”.
query “Class name” $x construct $x[]	Query instances of classes This will return the attributes and relationships of the “Class name” class.
query “object id”	Query object instance This type of query will return attributes and relationships belonging to an object instance with “object id” object identifier.
Query “Class name” $x […]	Query attributes of instances of classes This query will query instances of class “class name” and will try to match the attributes and relationships specified in the square brackets “[…]”.
Query “Class name” $x [“attribute1”:$a = “some value”]	Query class instances with attribute comparison This query will query instances of class “class name” and will try to match attribute

	"attribute1" that have the value "some value". Note that the comparison operator can be >, <, = or !=.
Query "Class name" $x ["relationship1":$a ["attribute1":$b]]	Query class instances and relationship This query will query instances of class "class name" and will try to match the relationship "relationship1" whose instance is linked to an object that has attribute "attribute1".

Because IQL is structured differently from commonly used query languages a simple set of queries which describe some of the basic syntax for IQL are presented in the following examples:

1) Query if an object named "Steven Spielberg" is in the database (returns true or false):

```
query "Steven Spielberg";
```

2) Query all information of instances of the Movie class:

```
query Movie $x
construct $x[ ];
```

3) Query all information of object named "Titanic" belonging to class Movie:

```
query Movie $x = "Titanic"
construct $x[ ];
```

4) Query all information of instances belonging to the Movie class, who have an attribute named title with the value "Titanic":

```
query Movie $x//title : "Titanic"
construct $x[ ];
```

5) Query all information of instances belonging to the Movie class, who have an attribute named title with the value "Titanic" or "Titanic 2":

```
query Movie $x[title: $a = "Titanic" |  "Titanic 2"]
construct $x[ ];
```

6) Query all information of instances belonging to the Movie class, who have an attribute named title with the value "Titanic" and an attribute named year with a value greater than 1950:

```
query Movie $x [title : $a = "Titanic", year:$b > 1950)
construct $x[ ];
```

7) Query all information of instances belonging to the Movie class, who have an attribute named title with a value containing the string "tan":

```
query Movie $x[title : $a = "%tan%"]
construct $x[ ];
```

8) Query all distinct instances of the Movie class:

```
query Movie $x
construct distinct $x;
```

9) Calculate the number of instances of Movie class

```
query Movie $x
construct count({$x});
```

10) Query the the lowest year value for all instances of the class Movie.

```
query Movie $x//year : $y
construct min({$y});
```

11) Query the names of Directors of every Movie and the title of the Movie. (JOIN)

```
query Movie $x [Title:$a, Director:$b[Name:$c]
construct $x[$a,$b[$c]];
```

12) UNION the names of Persons and title of Companies:

```
query {Person|Company} $x[Name:$n | Title:$na]
```

```
construct $x[$n,$na];
```

13) Select movies that have or don't have a Director. (LEFT JOIN)

```
query Movie $x [! Director is null | Director:$b[Name:$c]
construct $x[$a,$b[$c]];
```

14) Query all instances of Person and its subclasses.

```
query Person* $s
construct $s[ ];
```

15) Negation. Query all Persons and Person sub-class instances who are not Directors.

```
query Person* $x, not Director $x
construct $x [ ];
```

16) GROUP BY and aggregate SUM the year released for each Movie.

```
query Movie $x/"YearReleased":$y
construct $x[sum({$y})];
```

17) List all movies and ORDER BY their title in descending order.

```
query Movie $m/Title:$t
construct $m/$t order by $t DESC;
```

18) Query all movies and return only the results 10 to 20.

```
query Movie $m
construct $m top(10, 20);
```

2.2.4 INMDB Requirements for RDFS Storage

RDF storage applications can have several requirements from an underlying database storage system. A short list of features that are important for a database system to have as an RDFS storage backend, from RDFS application designers perspective would bes:

- Multi-user access - Required when an RDF application can be used by more than one person at once.
- Operating statistics - For monitoring workload on the database system
- APIs – Support for popular programming languages can ensure allow more developers to make applications.

INMDB supports multi-user access, which makes it a good backend for websites and medium-large businesses which require concurrent access to the database. INMDB also has some API capability for interfacing with applications. Currently INMDB does not support work load monitoring.

Chapter 3

Existing RDF Storage Systems and Query Processing

RDF storage systems are used for storing raw RDF triples in order to facilitate query answering and dataset analysis. Recent surveys highlight the effort that has been applied towards the development of different storage infrastructures [17,18,19,19,21,22,23,24,25]. There are five common RDF data storage approaches: Native [28,29,30,31,32,33,34,35,36], Relational [28,37,38,39,40,41,42,43,44,45,46,47,48,49,50], Object-Oriented [51,52], Object-Relational [53,54] and Expert Systems [55,56,57]. Of these, native and relational-backed are the most common approach [17,19,58]. This chapter describes how these methods store and query RDF data, their advantages and their disadvantages.

3.1 Native Storage

Native storage is the approach of building a storage system designed specifically for storing and querying RDF data. The strength of each of these systems is founded on customized data structures, indexes and query processing. The underlying storage representation of the RDF data is designed to easily map semantic web query languages (such as SPARQL) and thus optimize query answering [17]. Data can be stored either persistently, on top of file systems or transiently in main memory. There are three types of native storage: triplestores, graph-based and in-memory.

3.1.1 Triplestores

The triplestore approach represents RDF data persistently on disk in one or more large tables. At a low level, these tables resemble a comma separated file format where each row represents a single triple. Implementations differ in their underlying representation of the triple within the table and the indexes that are implemented. The extensive use of customized indexes distinguishes the triplestore's approach from other storage approaches. While the use of indexes can significantly improve query performance, this is associated with an increase in storage space [30,34].

From our investigation, the most common table structure for the triplestore approach is the single large table where rows of triples containing their subject, predicate and object values are stored directly in one single table [28,30,33,34]. An example of this structure is shown in Figure 5. A fourth element called the graph identifier is sometimes included along with the subject, predicate and objects values which allow the system to store information about the origin or provenance of the data [29,76,77]. Tables containing a graph identifier are referred to as quadstores. An example of this structure is shown in Figure 6. In this example all the triples have a graph identifier value of 1 which means they all originate from the same graph. Another implementation stores RDF data in three tables [78]: Subject-Predicate table, Predicate-Object table and a Subject-Object table. By keeping three separate triple tables, it speeds up join computations by allowing subject-object and object-object joins to be implemented as sorted-merge joins instead of regular joins.

Several implementations keep dictionary encodings which replace the subject, predicate and object values with id values that can be looked up in the dictionary table

[30,34,76]. The use of dictionaries reduce long literal values (for example triple subjects that are quite long) into shorter integers, which can then be joined together much faster (because it is faster to match an integer then a long string).

Subject	Predicate	Object
<ex.ca/r/Wall-E>	<ex.ca/p/Director>	<ex.ca/r/BobSmith>
<ex.ca/r/Wall-E>	<ex.ca/p/Writer>	<ex.ca/r/AndrewStanton>
<ex.ca/r/Wall-E>	<ex.ca/p/Writer>	<ex.ca/r/PeteDocter>
<ex.ca/r/Wall-E>	<ex.ca/p/YearReleased>	2008
<ex.ca/r/Wall-E>	<ex.ca/p/Sequel>	<ex.ca/r/Wall-E 2>
<ex.ca/r/Wall-E 2>	<ex.ca/p/Director>	<ex.ca/r/BobSmith>
<ex.ca/r/Wall-E>	<ex.ca/p/Madeby>	<ex.ca/r/Disney>
<ex.ca/r/Wall-E 2>	<ex.ca/p/Madeby>	<ex.ca/r/Disney>
<ex.ca/r/Disney>	<ex.ca/p/Yearfounded>	1940
<ex.ca/r/Disney>	<ex.ca/p/ChiefWriter>	<ex.ca/r/AndrewStanton>
<ex.ca/r/AndrewStanton>	<ex.ca/p/Name>	"Andrew Stanton"
<ex.ca/r/BobSmith>	<ex.ca/p/Name>	"Bob Smith"
<ex.ca/r/PeteDocter>	<ex.ca/p/Name>	"Pete Docter"

Figure 5: Example of Triplestore Table Structure.

GraphID	Subject	Predicate	Object
1	<ex.ca/r/Wall-E>	<ex.ca/p/Director>	<ex.ca/r/BobSmith>
1	<ex.ca/r/Wall-E>	<ex.ca/p/Writer>	<ex.ca/r/AndrewStanton>
1	<ex.ca/r/Wall-E>	<ex.ca/p/Writer>	<ex.ca/r/PeteDocter>
1	<ex.ca/r/Wall-E>	<ex.ca/p/YearReleased>	2008
1	<ex.ca/r/Wall-E>	<ex.ca/p/Sequel>	<ex.ca/r/Wall-E 2>
1	<ex.ca/r/Wall-E 2>	<ex.ca/p/Director>	<ex.ca/r/BobSmith>
1	<ex.ca/r/Wall-E>	<ex.ca/p/Madeby>	<ex.ca/r/Disney>
1	<ex.ca/r/Wall-E 2>	<ex.ca/p/Madeby>	<ex.ca/r/Disney>
1	<ex.ca/r/Disney>	<ex.ca/p/Yearfounded>	1940
1	<ex.ca/r/Disney>	<ex.ca/p/ChiefWriter>	<ex.ca/r/AndrewStanton>
1	<ex.ca/r/AndrewStanton>	<ex.ca/p/Name>	"Andrew Stanton"
1	<ex.ca/r/BobSmith>	<ex.ca/p/Name>	"Bob Smith"
1	<ex.ca/r/PeteDocter>	<ex.ca/p/Name>	"Pete Docter"

Figure 6: Example of Quadstore Table Structure.

Triplestores implement different forms of indexing. One of the most common approaches for indexing is to apply six indexes (sextuple indexing) on the single large triple table [30,34, 77,78,79]. Sextuple indexing makes use of the fact that an RDF triple is a three-dimensional entity that is fixed (i.e., The order of the subject, predicate and object do not change) meaning that it has a finite number of permutations. These six indexes represent all possible permutations of the subject (S), predicate (P) and object (O) that appear in the single large table. These permutations are: SPO, SOP, OSP, OPS, PSO, POS. By creating an index for each permutation we can optimize our query for each possible query sub-graph pattern ordering. For example, different indexes would be applied when processing the query for a subject-object join versus an object-subject join.

Numerous other indexing methods exist; these include the use of B+ trees [76], a combination of AVL and B trees [77] or hash indexes [80].

The process by which querying is performed on the RDF data in a triplestore is sometimes based on relational algebra or relational calculus [29,30,34]. From a detailed investigation, triple store systems have different query engines which process queries differently due to the unique indexes or table structures of the underlying table storage [28,76,77,78,80]. For example, one system stores access patterns which are an index that allows the lookup of any combination of s,p,o rather than having to perform the join from multiple indexes [76]. Systems which implement sextuple indexing will order their joins based of the different index permutations. Triplestores can often limit themselves to a specialization of asking certain types queries because of their storage architecture (e.g., indexing or table structure) [17].

One approach for processing RDF queries which can be applied to any single large table triplestore is described as follows [29]. The first step in processing a query for a triplestore is to transform the graph query (for example a SPARQL query) into an algebra representation (e.g., SPARQL algebra) which is suitable for optimizations. In this step the graph query is parsed and expanded into several simple triple patterns called basic graph patterns. When dictionary encodings are used in the triple store the parser looks up the values of triple patterns and replaces them with their corresponding dictionary IDs. When only one triple pattern is present in the algebra representation the indexes are used to perform a scan of the single large table to fetch the results. When more than one triple pattern are present in the algebra representation, the results of each individual pattern must be joined together. The order in which joins are performed is computed by an algorithm which calculates the optimal join ordering (e.g., by table size). Some systems support a form of dynamic optimization which further optimizes the join plan as it proceeds through its execution [77].

Below, an example SPARQL query being translated into relational algebra for a triplestore, is shown. Once the relational algebra has been generated, the triplestore system will execute the query and return results to user.

Step 1 – SPARQL query

```
SELECT *
WHERE
  {
    ?movie   a     Movie .
    ?movie   <ex.ca/p/sequel>  "<ex.ca/r/Wall-E 2>" .
    ?movie   <ex.ca/p/yearReleased> ?yearReleased .
    OPTIONAL { ?movie <ex.ca/p/Writer> ?x .
               ?x <ex.ca/p/Name> ?name }
  }
```

Step 2 – Translate to algebra

```
LEFTJOIN(
  JOIN(BGP(?movie :a :Movie),
        JOIN(BGP(?movie <ex.ca/p/sequel> "<ex.ca/r/Wall-E 2>"),
              BGP(?movie <ex.ca/p/yearReleased> ?yearReleased))),
  JOIN(BGP(?movie <ex.ca/p/Writer> ?x),
        BGP(?x <ex.ca/p/Name> ?name)),
TRUE
)
```

Step 3 – Translate to relational algebra

$$\Pi_{movie,yearReleased,x,name} \bowtie ($$
$$\Pi_{movie,yearReleased} \bowtie (\Pi_{movie}\, \sigma_{predicate = rdf:type}\, (triple),$$
$$\Pi_{movie,yearReleased} \bowtie (\Pi_{movie,yearReleased}\, \sigma_{predicate = <ex.ca/p/yearReleased>}\, (triple),$$
$$\Pi_{movie}\, \sigma_{predicate = <ex.ca/p/sequel>}\, (triple))),$$

$$\Pi_{movie,x,name} \bowtie (\Pi_{x,name}\, \sigma_{predicate = <ex.ca/p/Name>}\, (triple),$$
$$\Pi_{movie,x}\, \sigma_{predicate = <ex.ca/p/Writer>}\, (triple)))$$

Step 4 – Execute query on RDF store and return results

?movie	?yearReleased	□x	?name
<ex.ca/r/Wall-E>	2008	<ex.ca/r/AndrewStanton>	"Andrew Stanton"
<ex.ca/r/Wall-E>	2008	<ex.ca/r/PeteDocter>	"Pete Doctor"

Some well-known triplestore systems are: Jena TDB [28], 4store [29], RDF-3X [30,34], Hexastore [79], RQ- RDF-3X [33], YARS [76], Kowari [77], RDFJoin [78] and RDFCube [80]. The pros and cons of triples stores are shown below:

Pros

- Custom indexing allows for fast merge-joins for any pairs of triple patterns.
- In general it is much faster to retrieve information from a triplestore than a vertical relational-backed RDF store because of indexing.
- Adding new RDF data does not require restructuring the underlying storage
- Easy to import or export RDF data using standardized formats (N-triples or N-quads)
- Easy to move RDF data from one triplestore to another, meaning the user is not locked into a single vendor.
- Good compliance to standards (eg. SPARQL, import-export formats, etc.)

Cons

- Extensive use of indexing increases the space required for storing RDF data (for example Hexastore requires 5 times the space as a relational store)
- Query optimizers are not as well developed as those found in RDBMS.
- Dependent on middleware-level logic to enforce constraints on data (i.e., Duplicate values, etc.) whereas relational database management systems handle this at a much lower level.
- Most implementations have limitations either on their scalability and/or specialization of their architecture for particular kinds of queries [17]
- Does not represent the RDFS as database schema in storage.

3.1.2 Graph-Based

The graph-based approach uses a graph store to represent RDF with nodes and pointers. The key concept of a graph database is that edges (or relationships) are represented by directly relating nodes together using pointers. The graph-based approach is ideally suited for representing RDF data because at a high semantic level RDF is a graph [81,82].

In the graph-based approach for storing RDF, data is stored such that the nodes in the graph are RDF triple subjects or objects, and the directed edges are predicates. At a low level, nodes are data structures which contain pointers (representing edges) to the location of other nodes on disk. In this way, direct access is gained to other nodes eliminating the need for expensive search or match computations when querying the graph. The use of pointers enables implementations which are often much more scalable then triplestores. In addition, indexes can be stored for every node and predicate, further enhancing query speeds when searching the graph. This architecture enables for fast query answer speeds even on very large datasets.

Figure 7 is a high level representation of the RDF dataset example as represented by a graph-based RDF store. Each node is a data structure containing an attribute for either a subject or object and a set of pointers representing the predicates. It is important to note that in RDF, subjects and object mappings are unique URIs (uniform resource identifiers). This means that in the graph database if two or more triples share the same URI value, they will be represented by the same node. For example, in the graph shown, "Wall-E" is a single node that represents the subject of 5 different triples. When representing object literals such as non-unique string or integer values, for example "1940", graph systems can either reference the same node, store different redundant nodes or store the literal as an attribute within the node (such as a string or integer) instead of as a pointer to a node.

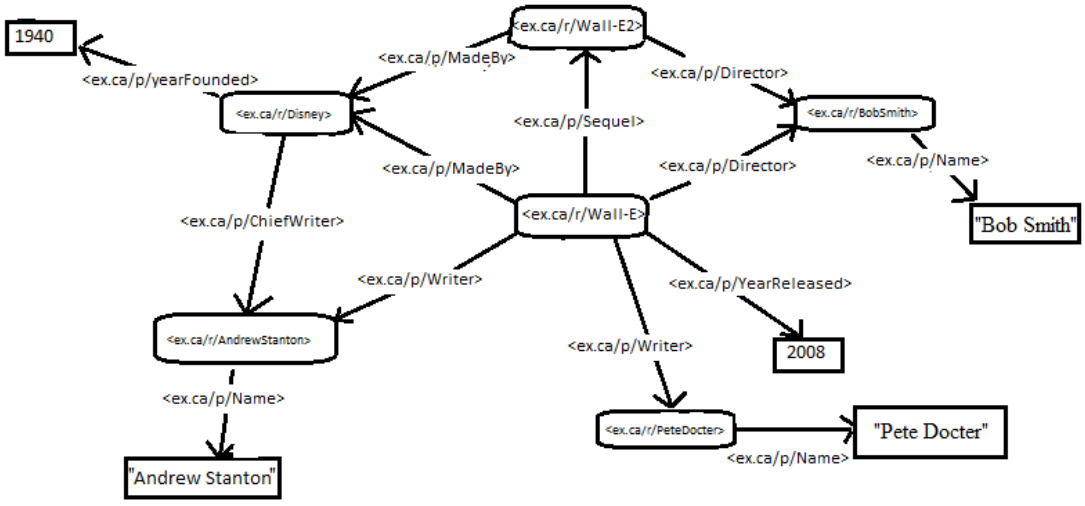


Figure 7: Graph-based Representation of RDF Data

To query the RDF graph using SPARQL, native graph-based database systems implement sub-graph pattern matching [35,36,83]. Sub-graph matching uses homomorphisms to map the SPARQL graph pattern to the underlying RDF graph in

storage. If a match is found, the database system will then retrieve the data requested by the query. An example SPARQL query and its homomorphism mapping to the graph are shown in the steps below.

Step 1 – SPARQL query

```
SELECT *
WHERE
  {
    ?x   ?y  ?z .
    ?z   <ex.ca/p/Name> "Pete Docter" .
  }
```

Step 2 – Generate Homomorphism

H = {?x → <ex.ca/r/Wall-E>,
?y → <ex.ca/p/Writer>,
?z → <ex.ca/r/PeteDocter>,
<ex.ca/p/Name> → <ex.ca/p/Name>,
"Pete Docter" → "Pete Docter" }

Step 3 – Return results

?x	?y	?z
<ex.ca/r/Wall-E>	<ex.ca/p/Writer>	<ex.ca/r/PeteDocter>

An illustration is shown in Figure 8 where the sub-graph pattern (shown in orange) is mapped onto the graph (shown in black) via a homomorphism (shown in purple).

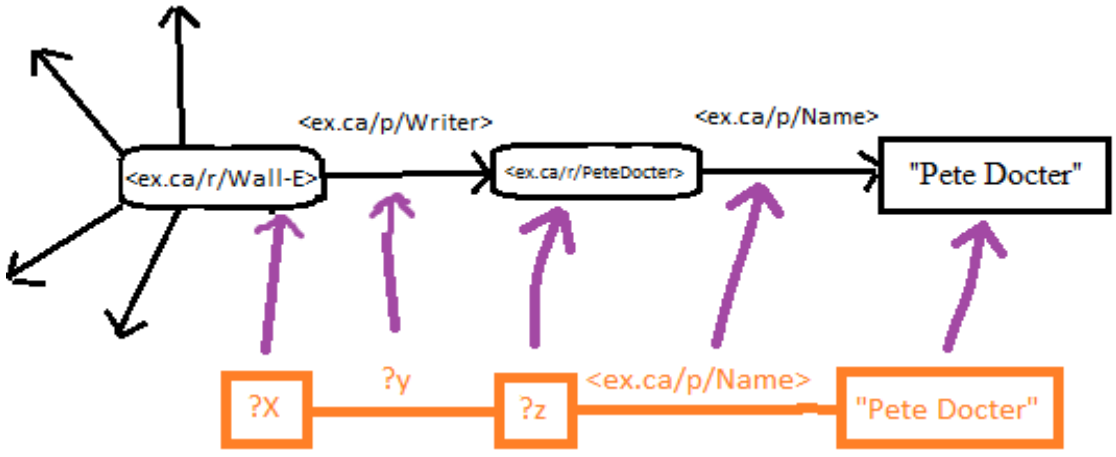


Figure 8: Subgraph Homomorphism

Some graph-based RDF storage systems are: gStore [35,36], Ontotext GraphDB [31], Allegrograph[12], Dydra[13] and BrightstarDB[14]. The pros and cons of graph-based RDF storage systems are shown below.

Pros

- In general it is much faster to query information because of sub-graph matching and pointer look-ups.
- Adding new data doesn't require the underlying data structures change.
- The most natural way of storing RDF data as it is represented as a graph on disk.
- High scalability (ability to handle tens of billions of triples. For example, allegrograph [40] is a distributed graph database which can support 1.009 trillion triples)
- These systems often have the ability to infer information from the graph (i.e., Create new information based on what is found in the graph) [31]
- Good compliance to standards (eg. SPARQL, import-export formats, etc.)

Cons

- Query optimizers are not as well developed as those found in RDBMS.
- Dependent on middleware-level logic to enforce constraints on data (i.e., Duplicate values, ontology rules, etc) whereas relational database management systems handle this at a much lower level.
- Does not represent the RDFS as database schema in storage.

3.1.3 In-memory

In-memory stores attempt to keep all RDF data in memory and are specifically designed to support fast access. In-memory native RDF databases store data the same way as the triplestore or graph-based approach, the difference is that instead of persistent disk storage these systems store data in system memory.

[12] Allegrograph - http://www.franz.com/products/allegrograph/
(retrieved Aug 01, 2017)
[13] Dyhdra - http://dydra.com/
(retrieved Aug 01, 2017)
[14] BrightStarDB - http://brightstardb.com/
(retrieved Aug 01, 2017)

The systems BRAHMS [84], BitMat [85], TripleT [86], RAP [87] and Profium Sense[15] structure their data as single large tables, the same way as triplestores. Similarly they query the RDF data using the same process that triplestores use to query RDF data.

An example of a graph-based in-memory database system is OntoDB [32]. This system represents RDF data as a graph with an object-oriented language. At the structural level, objects represent RDF subjects and objects as nodes on the graph and pointers represent the predicates as edges on the graph. The same sub-graph pattern matching techniques used in graph-based systems are also used for in-memory implementations.

The pros and cons of using an in-memory RDF storage system are shown below.

Pros

- Fast access and fast inference because data is stored in memory.
- Adding new RDF data does not require restructuring the underlying storage
- Easy to import or export RDF data using standardized formats (N-triples or N-quads)
- Easy to move RDF data from one store to another.
- Good compliance to standards (eg. SPARQL, import-export formats, etc.)

Cons

- Difficulty handling large RDF datasets because system memory is the limit (Often much smaller then storage on disk)
- Use of indexing increases the memory space required for storing RDF data
- Query optimizers are not as well developed as those found in RDBMS.
- Dependent on middleware-level logic to enforce constraints on data (i.e., Duplicate values, etc) whereas relational database management systems handle this at a much lower level.
- Most implementations have limitations due to specialization of their architecture for particular kinds of queries [17]
- Does not represent the RDFS as database schema in storage.

[15] Profium Sense - http://www.profium.com/en/profium-sense (retrieved Aug 01, 2017)

3.2 Relational Storage

Relational-backed storage of RDF data uses a relational database management system as the primary method for storing and retrieving data. Relational databases have been around for a long time (over 40 years) and are well established so they are often chosen as back ends for RDF and semantic web data stores. In general they are found to be very reliable for storing and querying data because they implement ACID (atomicity, consistency, isolation and durability) transactions and have excellent query optimizers. RDF is also easily mapped to the relational model[16].

Some examples of relational-backed RDF storage applications are: C-Store [37], RDFPROV [39], rdf4j (Sesame) [40], Jena1 [28], Jena2 [41], Oracle [42], 3Store [43], Redland [44], RDFstore [45], rdfDB[17], Virtuoso [46], DB2RDF [38], ohStore [47], TripleFCA [48], roStore [49] and RStar [50]. Many of these systems enable the use of different relational databases systems as their backend and sometimes object-relational databases are used in place of relational database [40,41,44].

There are three main approaches for relational RDF data storage: Vertical, binary and N-ary (property tables, property class tables and direct primary hash tables).

3.2.1 Vertical Table Storage

The easiest and most common relationally backed approach is to use the vertical approach. The vertical approach is similar to the native triplestore approach which stores

[16] Tim Berners- Lee. Relational Databases on the Semantic Web. 1998. https://www.w3.org/DesignIssues/RDB-RDF.html (retrieved Aug 01, 2017)

[17] R. V. GUHA. rdfDB: An RDF Database. 2000. http://www.cs.cmu.edu/afs/cs/usr/niu/rdf/ (retrieved Aug 01, 2017)

all triples in a single table. In this approach the single table has one column for each of the components of the triple: subject, predicate and object. There are several variations on this single table approach which include adding row identifiers to each triple, implementing dictionary encodings, as well as normalizing the table by separating subjects, predicates and objects into their own tables and performing star joins when querying.

Continuing with the example of the "Wall-E" set of triples and RDF schema shown previously in section 2.1.7, an example vertical RDF store stored in a relational would have the following schema shown below and database instance shown in Figure 9.

Vertical Table Schema

```
Create Table VerticalStore {
Subject String,
Predicate String,
Object String
};
```

Subject	Predicate	Object
<ex.ca/r/Wall-E>	<ex.ca/p/Director>	<ex.ca/r/BobSmith>
<ex.ca/r/Wall-E>	<ex.ca/p/Writer>	<ex.ca/r/AndrewStanton>
<ex.ca/r/Wall-E>	<ex.ca/p/Writer>	<ex.ca/r/PeteDocter>
<ex.ca/r/Wall-E>	<ex.ca/p/YearReleased>	2008
<ex.ca/r/Wall-E>	<ex.ca/p/Sequel>	<ex.ca/r/Wall-E 2>
<ex.ca/r/Wall-E 2>	<ex.ca/p/Director>	<ex.ca/r/BobSmith>
<ex.ca/r/Wall-E>	<ex.ca/p/Madeby>	<ex.ca/r/Disney>
<ex.ca/r/Wall-E 2>	<ex.ca/p/Madeby>	<ex.ca/r/Disney>
<ex.ca/r/Disney>	<ex.ca/p/Yearfounded>	1940
<ex.ca/r/Disney>	<ex.ca/p/ChiefWriter>	<ex.ca/r/AndrewStanton>
<ex.ca/r/AndrewStanton>	<ex.ca/p/Name>	"Andrew Stanton"
<ex.ca/r/BobSmith>	<ex.ca/p/Name>	"Bob Smith"
<ex.ca/r/PeteDocter>	<ex.ca/p/Name>	"Pete Docter"

Figure 9: Relational Vertical Store Example

Some popular RDF storage solutions which implement a vertical storage approach are: Jena1 [1], Sesame [40], Oracle [42], 3Store [43], Redland [44], RDFstore [45], rdfDB, among many others. Pros and cons are described below.

Pros

- Easiest method to implement.
- No knowledge of RDF schema required
- In general graph queries are easy to translate into SQL for a vertical RDF datastore
- We do not need to worry about changes to the database schema (i.e., The single table remains fixed).
- Does not have null values present in the data, unless elements of the triple are unknown.
- Can handle multi-valued data

Cons

- This method can produce a very large table when storing many RDF triples
- Not the most efficient approach
- To query this data efficiently can require some specialized techniques such as managing the join ordering, which requires some metadata knowledge of the RDF dataset [30].
- If you must formulate your query into SQL by first navigating the graph, this can overall slowdown performance of the RDF datastore [90].
- This type of storage is efficient when a large amount of data is retrieved from the table, however it is not appropriate when retrieving very small amounts of data [89]
- Complex queries require many self-joins over a single large table [79,97]
- Does not represent the RDFS as database schema in storage.

3.2.2 Binary Table Storage

The binary table storage method was first proposed by Abadi [37,88]. The binary table storage method separates all RDF data into separate tables, where each table is used to represent a predicate of the RDF triples having a subject column and an object column. As you add more data to the store and new predicates are encountered, we add new tables which can introduce the problem of schema evolution. The tables in binary storage

method are preferably stored in column-oriented database managements systems to increase query speed [17].

The schema for the binary table method is shown below and an instance database using the working example is shown in Figure 10.

Binary Table Schema

```
Create Table <ex.ca/p/Director> {
  Subject String,
  Object String
};

Create Table <ex.ca/p/Writer> {
  Subject String,
  Object String
};

Create Table <ex.ca/p/YearReleased> {
  Subject String,
  Object String
};

Create Table <ex.ca/p/Sequel> {
  Subject String,
  Object String
};

Create Table <ex.ca/p/MadeBy> {
  Subject String,
  Object String
};

Create Table <ex.ca/p/YearFounded> {
  Subject String,
  Object String
};

Create Table <ex.ca/p/ChiefWriter>{
  Subject String,
  Object String
};

Create Table <ex.ca/p/Name>{
  Subject String,
  Object String
};
```

<ex.ca/p/Director>

Subject	Object
<ex.ca/r/Wall-E>	<ex.ca/r/BobSmith>
<ex.ca/r/Wall-E 2>	<ex.ca/r/BobSmith>

<ex.ca/p/Writer>

Subject	Object
<ex.ca/r/Wall-E	<ex.ca/r/AndrewStanton>
<ex.ca/r/Wall-E	<ex.ca/r/PeteDocter>

<ex.ca/p/YearReleased?

Subject	Object
<ex.ca/r/Wall-E>	2008

<ex.ca/p/Sequel>

Subject	Object
<ex.ca/r/Wall-E>	<ex.ca/r/Wall-E 2>

<ex.ca/p/Madeby>

Subject	Object
<ex.ca/r/Wall-E>	<ex.ca/r/Disney>
<ex.ca/r/Wall-E 2>	<ex.ca/r/Disney>

<ex.ca/p/YearFounded>

Subject	Object
<ex.ca/r/Disney>	1940

<ex.ca/p/ChiefWriter>

Subject	Object
<ex.ca/r/Disney>	<ex.ca/r/AndrewStanton>

<ex.ca/p/Name>

Subject	Object
<ex.ca/r/AndrewStanton>	"Andrew Stanton"
<ex.ca/r/BobSmith>	"Bob Smith"
<ex.ca/r/PeteDocter>	"Pete Docter"

Figure 10: Relational Binary Store Instance Example

Some popular binary relational RDF stores are C-Store [37] and roStore [49].

Pros and cons of the binary table method are shown below.

Pros

- Faster query execution than vertical storage
- Does not experience many null values and is therefore faster than N-Ary storage for many types of queries because it does not have additional overhead caused by null values
- can handle multi-valued data
- Fewer unions and faster joins [89]
- Rapid subject-subject joins because you can cluster index [17]

Cons

- Queries are more complex to write if not specifying the predicate.
- Having many tables in the binary method can generate some very large queries which are otherwise very simple in a vertical store or N-ary store
- For very large RDF data sets, the number of tables used to store data can quickly get out of hand.
- Every time a new predicate or type of thing is added to the database you must change the schema
- Because the predicates are elevated as column names and subject/objects are normal values stored inside the table, there is a weakness when asking queries that do not specify the predicate value in the query [90]. For queries that don't specify the predicate value all tables must be analyzed to return the result of the query, which can reduce performance especially when the dataset has many different predicates values.
- With a large number of properties, vertical triple stores can outperform binary storage solutions [91].
- Does not represent the RDFS as database schema in storage.

3.2.3 N-ary Table Storage

3.2.3.1 Property Table and Class Table N-ary

The first mention of N-ary storage was given by a paper for Jena 2 [41]. The N-ary method separates all RDF data based on the RDFS class (i.e., People, Companies, Animals, etc..) or by triples having similar properties into their own tables [17]. RDF data stored based on class is sometimes referred to as property-class table storage and RDF data stored based on clusters of similar properties is sometimes referred to as property-table storage [17]. In the property-class approach each ontology class is represented as a table, with predicates belonging to these classes as table columns. Multi-valued properties are separated into their own property table.

The following example would be an N-ary property-class table representation of the vertical RDF store given previously:

N-ary Property Class Table Schema

```
//property-class tables
Create Table Movie {
  Subject String,
  Director String,
  YearReleased int,
  Sequel String,
  Madeby String
};
Create Table Company {
  Subject String,
  Yearfounded int,
  ChiefWriter String
};
Create Table Person{
  Subject String,
  name string
};
Create Table MovieWriter{
  Subject String,
  name string
```

```
};
Create Table MovieDirector{
  Subject String,
  name string
};

//multi-valued property table
Create Table Writer{
  Subject string,
  Writer String
}
```

Movie table

Subject	Director	YearReleased	Sequel	MadeBy
<ex.ca/r/Wall-E>	<ex.ca/r/BobSmith>	2008	<ex.ca/r/Wall-E2>	<ex.ca/r/Disney>
<ex.ca/r/Wall-E2>	<ex.ca/r/BobSmith>			<ex.ca/r/Disney>

multivalued Writer property table

Subject	Writer
<ex.ca/r/Wall-E>	<ex.ca/r/AndrewStanton>
<ex.ca/r/Wall-E>	<ex.ca/r/PeteDocter>

Person table

Subject	name

movie writer table

Subject	name
<ex.ca/r/AndrewStanton>	"Andrew Stanton"
<ex.ca/r/PeteDocter>	"Pete Docter"

Company table

Subject	Yearfounded	ChiefWriter
<ex.ca/r/Disney>	1940	<ex.ca/r/AndrewStanton>

movie Director table

Subject	name
<ex.ca/r/BobSmith>	"Bob Smith"

Figure 11: Relational N-ary Store Instance Example

This method can make some types of queries much faster and simpler, because if we are interested in particular predicates or types of objects (people, movies, cars, etc.), the system knows where to look for the data and can avoid performing subject-subject joins. When you add more triples to the store, the underlying tables change (new tables are added or new properties are added to existing tables), this can introduce the problem of schema evolution. Additionally, multi-valued properties are not easily captured with this approach.

Of the relational approaches, the N-ary class based method has the best representation of RDFS as relational database schema.

Some popular RDF storage solutions which implement a vertical storage approach are: Jena2 [41] and ohStore [47].

Pros

- Faster query execution than vertical storage
- Queries where the class is known are much faster and simpler [42,92].
- Makes best use of the relational database schema for representing RDFS.
- Self-joins on subject column can be avoided [17].

Cons

- For very large RDF data stores, the number of tables used to store data can quickly get out of hand [38].
- every time a new predicate or type of thing is added to the database you must change the schema [17]
- It can generate a lot of null values in the tables [17,93], having a non-dense (lots of null values) relational table adds considerable overhead when processing queries [60]. (Some database softwares have null-value compression such as PostgreSQL, which can counteract the downsides of querying on sparse tables.)
- Because the predicates are elevated as column names and subject/objects are normal values stored inside the table, there is a weakness when asking queries that do not specify the predicate value in the query [90]. For these queries the whole table must be analyzed to return the result of the query, which can reduce performance especially when the dataset has many different predicates values (columns in the N-ary table).
- Not easily able to capture multi-valued data [17,37], rarely used in implementations because of this [17].

3.2.3.2 Direct Primary Hash

A variation on the N-ary method is called the direct primary hash (DPH), used in the system IBM DB2RDF [38]. This approach uses a single table called the DPH to store subjects and k number of predicate-object pairs. When the number of predicates overflow, and become greater than k, a new row is created in the DPH and new

predicates are added there, as shown in the example below for triples with the subject "Wall-E". Multi-valued properties are stored in a secondary table called the direct secondary hash (DSH), which stores a unique id linked to the object column in the DPH.

An example schema of a k = 4 DPH/DSH approach for the working example is shown in below and an example instance is shown in Figure 12.

DPH/DSH Table Schema

```
Create Table DPH {
  Subject String,
  Pred1 String,
  Obj1 String,
  Pred2 String,
  Obj2 String,
  Pred3 String,
  Obj3 String,
  Pred4 String,
  Obj4 String
};

Create Table DSH {
  ID Int,
  Object string
};
```

DPH with k = 4

Subject	Pred1	Obj1	Pred2	Obj2	Pred3	Obj3	Pred4	Obj4
<ex.ca/r/Wall-E>	<ex.ca/p/Director>	<ex.ca/r/BobSmith>	<ex.ca/p/Writer>	1	<ex.ca/p/YearReleased>	2008	<ex.ca/p/Sequel>	<ex.ca/r/Wall-E2>
<ex.ca/r/Wall-E>	<ex.ca/p/MadeBy>	<ex.ca/r/Disney>						
<ex.ca/r/Wall-E2>	<ex.ca/p/Director>	<ex.ca/r/BobSmith>	<ex.ca/p/MadeBy/>	<ex.ca/r/Disney>				
<ex.ca/r/BobSmith>	<ex.ca/p/name>	"Bob Smith"						
<ex.ca/r/Disney>	<ex.ca/p/YearFounded>	1940	<ex.ca/p/ChiefWriter>	<ex.ca/r/AndrewStanton>				
<ex.ca/r/AndrewStanton>	<ex.ca/p/name>	"Andrew Stanton"						
<ex.ca/r/PeteDocter>	<ex.ca/p/name>	"Pete Docter"						

DSH

ID	object
1	<ex.ca/r/AndrewStanton>
1	<ex.ca/r/PeteDocter>

Figure 12: Relational DPH/DSH Example

The pros and cons of the DPH/DSH approach are shown below.

Pros

- Faster query execution than vertical storage
- Elimination of subject-subject joins [38]
- Does not require schema change [38]

Cons

- Is designed to have less Null values than N-ary class property table storage [38], however it can still have significant null values [19] as shown in the example above compared to the N-ary class property table approach.
- Dealing with multi-valued attributes is still difficult. [19]
- More complex SQL query syntax than basic N-ary storage method.
- Does not represent the RDFS as database schema in storage.

3.2.4 RDF Query Processing in Relational Storage

To query RDF data in a relational RDF store an RDF graph query (such as SPARQL) must be translated into the query language of the underlying relational database, most commonly this language is SQL. The SQL query is then executed on the relational database, the resulting table is then processed by middleware into one of several desired formats (csv file, XML, HTML, raw data, etc) and finally returned to the user. An illustration showing the general architecture for relational-backed systems is shown in Figure 13.

RDF Queries in relational databases are processed nearly identically for the three relational RDF storage approaches, however, the SQL generated for the three approaches depends on the table structures used for storage.

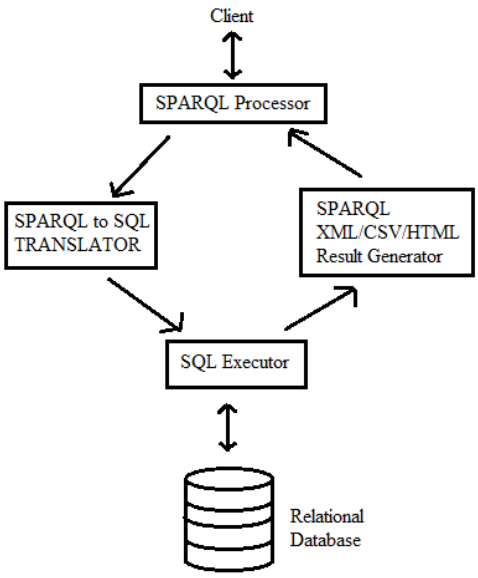


Figure 13: Relational RDF Query Processing Architecture

All SPARQL queries have an equivalent translation to SQL. SQL is a much more expressive language then SPARQL and supports many more types of queries then what is offered by SPARQL. All possible SPARQL queries are a subset of all possible SQL queries.

A well-known method for translating SPARQL to SQL described by Mariano Rodríguez-Muro and Martin Rezk has five general steps [94]:

1) Parse SPARQL graph pattern
2) Convert SPARQL to SPARQL algebra
3) Convert SPARQL algebra to Datalog
4) Convert Datalog to relational Algebra
5) Translate to SQL

The first step parses the SPARQL query breaking it into logical data types. The second step involves taking the parsed SPARQL and converting it into SPARQL algebra. The translation steps required to convert SPARQL into SPARQL algebra are described in section 2.1.5.

Once the SPARQL algebra expression is generated, the next step is to convert it into Datalog. This involves taking the SPARQL algebra tree (the expression) and generating a series of rules for each node in the tree, starting from the root. The rules for each node encode the meaning of each of the algebra expressions; where the head of the Datalog rule project the bindings that correspond to the SPARQL algebra expression and the body of the rule implements the expression itself. The Datalog program is generated by recursively translating each node into a set of rules, and then their children nodes also into sets of rules. The leaves of the SPARQL algebra tree are basic graph patterns, so they result in a special Datalog rule called a triple which symbolizes accessing the triple.

For the final step, we importantly note that Datalog with safe negation and without recursion is equivalent to relational algebra. Therefore, a well-designed SPARQL query can be translated to SQL. Throughout the steps algorithms can be applied which simplify the SPARQL alegrebra, datalog program, relational algebra or SQL.

Consider an example of a basic SPARQL query that finds triple subjects that are assigned to the “Movie” class which have the sequel “Wall-E 2”, the year released for this movie and the subject and name of all it’s writers. The following steps show how it is converted into SQL for a relational vertical triple store is shown below.

Step 1 – SPARQL query

```
SELECT *
WHERE
```

```
{
  ?movie   a     Movie .
  ?movie   <ex.ca/p/sequel>  <ex.ca/r/Wall-E 2> .
  ?movie   <ex.ca/p/yearReleased> ?yearReleased .
  OPTIONAL { ?movie <ex.ca/p/Writer> ?x .
             ?x <ex.ca/p/Name> ?name }
}
```

Step 2 – translate to SPARQL algebra

LEFTJOIN(
  JOIN(BGP(?movie :a :Movie),
        JOIN(BGP(?movie <ex.ca/p/sequel> "<ex.ca/r/Wall-E 2>"),
              BGP(?movie <ex.ca/p/yearReleased> ?yearReleased))),
  JOIN(BGP(?movie <ex.ca/p/Writer> ?x),
        BGP(?x <ex.ca/p/Name> ?name)),
TRUE
)

Step 3 – translate to Datalog program

ans9(movie, YearReleased, x, name)←LeftJoin(ans8(movie,yearReleased),
                                              ans3(movie,x,name),
                                              TRUE)
ans8(movie, yearReleased)← ans7(movie), ans6(movie, yearReleased)
ans7(movie)←triple(movie, "rdf:type", "Movie")
ans6(movie, yearReleased) ← ans5(movie), ans4(movie, yearReleased)
ans5(movie) ← triple(movie, "<ex.ca/p/sequel>", "<ex.ca/r/Wall-E 2">)
ans4(movie, yearReleased) ← triple(movie, "yearReleased", yearReleased)
ans3(movie, x, name) ← ans1(x, name), ans2(movie, x)
ans2(movie, x) ← triple(movie, "<ex.ca/p/Writer>", x)
ans1(x, name) ← triple(x, "<ex.ca/p/Name>", name)

Step 4 – translate to relational algebra

$\Pi_{movie,yearReleased,x,name} \bowtie($
  $\Pi_{movie,yearReleased} \bowtie(\Pi_{movie}\ \sigma_{predicate = rdf:type}$ (triple),
        $\Pi_{movie,yearReleased} \bowtie (\Pi_{movie,yearReleased}\ \sigma_{predicate = <ex.ca/p/yearReleased>}$ (triple),
                $\Pi_{movie}\ \sigma_{predicate = <ex.ca/p/sequel>}$ (triple))),

  $\Pi_{movie,x,name} \bowtie (\Pi_{x,name}\ \sigma_{predicate = <ex.ca/p/Name>}$ (triple),
              $\Pi_{movie,x}\ \sigma_{predicate = <ex.ca/p/Writer>}$ (triple)))

Step 5 – translate to SQL

```
SELECT *
FROM
(SELECT t5.subject, t3.object)
 FROM triplestore as t3 JOIN triplestore as t4 JOIN triplestore as t5
 WHERE t3.subject = t4.subject AND t3.subject = t4.subject
```

```
      AND t3.predicate = "<ex.ca/p/yearReleased>"
      AND t4.predicate = ""<ex.ca/p/sequel>"
      AND t4.object = ""<ex.ca/r/Wall-E 2>"
      AND t5.predicate = "rdf:type") as u2
 LEFT JOIN
(SELECT t1.subject, t1.object,t2.object
FROM triplestore as t1 JOIN triplestore as t2
WHERE t1.object = t2.subject AND t1.predicate = ""<ex.ca/p/writer>"
    AND t2.predicate = ""<ex.ca/p/name>") as u1
 WHERE t1.subject = t5.subject
```

Results

?movie	?yearReleased	?x	?name
<ex.ca/r/Wall-E>	2008	<ex.ca/r/AndrewStanton>	"Andrew Stanton"
<ex.ca/r/Wall-E>	2008	<ex.ca/r/PeteDocter>	"Pete Docter"

For a relational horizontal N-ary store a similar procedure can be applied, the steps for which are shown below. Steps 1 and 2 are identical between the vertical and N-ary translation procedure. The main difference is that the structural representation of RDF in the N-ary approach, properties are grouped together into their own table, so subject-subject joins between basic graph patterns are coalesced together. An additional difference is that in order to generate the Datalog step for N-ary knowledge of the relational database schema is required.

Step 3 – translate to Datalog program for N-ary implementation

```
ans9(movie, YearReleased, x, name)←LeftJoin(ans1(movie,yearReleased),
                                             ans4(movie,x,name),
                                             TRUE)
ans4(movie, x, name)←ans2(movie, x), ans3(x, name)
ans3(x, name) ← MovieWriter(x, name)
ans2(movie,x) ← Movie(movie, _, x, _, _, _)
ans1(movie, yearReleased) ← Movie(movie, _, _,
                                  yearReleased, "<ex.ca/r/Wall-E 2>", _)
```

Step 4 – translate to relational algebra

$\Pi_{movie,yearReleased,x,name} \bowtie ($
$\Pi_{subject,\ yearReleased}\ \sigma_{sequel = \text{"<ex.ca/r/Wall-E 2>"}}$(Movie),
$\Pi_{movie,x,name} \bowtie (\Pi_{x,name}\ \sigma$ (MovieWriter),

$\Pi_{movie,x}\ \sigma$ (Movie)))

Step 5 – translate to SQL

```
SELECT t3.subject, t3.yearReleased, t1.Writer, t2.name
FROM
  (SELECT t3.subject, t3.yearReleased
   FROM Movie as t3
   WHERE sequel = "<ex.ca/r/Wall-E 2>") as u1
  LEFT JOIN
  (SELECT t1.subject, t1.Writer, t2.name
   FROM Movie as t1 JOIN MovieWriter as t2
   WHERE Movie.Writer = MovieWriter.subject) as u2
WHERE t1.subject = t3.subject
```

Results

?movie	?yearReleased	?x	?name
<ex.ca/r/Wall-E>	2008	<ex.ca/r/AndrewStanton>	"Andrew Stanton"
<ex.ca/r/Wall-E>	2008	<ex.ca/r/PeteDocter>	"Pete Docter"

For the relational Binary table storage method the translation procedure is similar to the vertical storage method, which is shown below. Steps 1 and 2 are identical to the vertical storage method, but steps 3, 4 and 5 differ. They are different because in the Binary table method triples are each stored in a different table based on predicate. Therefore, in order to generate the Datalog step for Binary knowledge of the relational database schema is required.

Step 3 – translate to Datalog program for binary implementation

```
ans9(movie, YearReleased, x, name)←LeftJoin(ans8(movie,yearReleased),
                                             ans3(movie,x,name),
                                             TRUE)
ans8(movie, yearReleased)← ans7(movie), ans6(movie, yearReleased)
ans7(movie)←rdf:type(movie, Movie")
ans6(movie, yearReleased) ← ans5(movie), ans4(movie, yearReleased)
ans5(movie) ← <ex.ca/p/sequel>(movie, "<ex.ca/r/Wall-E 2>")
ans4(movie, yearReleased)←<ex.ca/p/yearReleased>(movie, yearReleased)
ans3(movie, x, name) ← ans1(x, name), ans2(movie, x)
ans2(movie, x) ← <ex.ca/p/Writer >(movie, x)
ans1(x, name) ← <ex.ca/p/Name>(x, name)
```

Step 4 – translate to relational algebra

$\Pi_{movie,yearReleased,x,name} \bowtie ($
$\Pi_{movie,yearReleased} \bowtie (\Pi_{subject\ as\ movie}\ \sigma$ (rdf:Type),
$\Pi_{movie,yearReleased} \bowtie (\Pi_{subject\ as movie,\ object\ as yearReleased}$
σ (<ex.ca/p/yearReleased>),
$\Pi_{subject\ as movie}\ \sigma_{object = \text{“<ex.ca/r/wall-e 2>”}}$
$\sigma_{(}$ <ex.ca/p/sequel>))),

$\Pi_{movie,x,name} \bowtie (\Pi_{subject\ as\ movie,object\ as\ x}\ \sigma$ (<ex.ca/p/Writer>),
$\Pi_{subject\ as\ x,object\ as\ name}\ \sigma$ (<ex.ca/p/Name>)))

Step 5 – translate to SQL

```
SELECT *
FROM
(SELECT t5.subject, t3.object)
 FROM rdf:type as t3 JOIN <ex.ca/p/yearReleased> as t4 JOIN <ex.ca/p/sequel> as t5
 WHERE t3.subject = t4.subject AND t3.subject = t4.subject
       AND t4.object = “<ex.ca/r/Wall-E 2>”) as u2
 LEFT JOIN
(SELECT t1.subject, t1.object, t2.object
FROM <ex.ca/p/Writer> as t1 JOIN <ex.ca/p/Name> as t2
WHERE t1.object = t2.subject) as u1
 WHERE t1.subject = t5.subject
```

Results

?movie	?yearReleased	?x	?name
<ex.ca/r/Wall-E>	2008	<ex.ca/r/AndrewStanton>	“Andrew Stanton”
<ex.ca/r/Wall-E>	2008	<ex.ca/r/PeteDocter>	“Pete Docter”

3.3 Object-Oriented Storage

Although the RDF data model has some object-oriented characteristics, the use of object-oriented database systems has been very rare for RDF storage [24]. Following an extensive search of the literature only two implementations making use of object-oriented database management systems where discovered [51,52]. Both implementations attempt to store RDF data directly as a graph similar to the graph-based approach discussed in section 3.1.2.

The class definitions for an object-oriented system is shown below [51]. In this system each element of the triple (Subject, Predicate and Object) is represented by a database object. Subjects and predicates are represented by the "Resources" class. Objects are either represented by the "Resources" or "Literals" class. Objects are "Resources" is if they reference other subjects and they are "Literals" if they represent literals (such as strings or integers). To create the triple, connections between the subjects and their predicate-object pairs are encoded using a key mappings attribute inherited by the "Values" class. This mapping indicates the predicate and the object for a given resource.

Object-Oriented Schema

```
CREATE CLASS Values {
  //stores list of reference to predicate & object
  LIST outedges [value *Predicate, value *Object];
}

CREATE CLASS Resources SUPERCLASS (Values){
  INT Id                          //Store an object ID
  STRING namespace                //Store namespace, e.g "ex.ca/r/"
  STRING localname                //Store local name, e.g "Wall-E"
}

CREATE CLASS Literals SUPERCLASS (Values){
  STRING Id                       //Store an object ID
  STRING language                 //Store language code, e.g., "@EN"
  STRING name                     //Store value of literal, e.g., "2008"
}
```

Consider the representation in this system for the running "Wall-E" example. Object-oriented database system objects of type "Literals" are shown in Figure 14, the objects of type "Resources" are shown in Figure 15, and the graph for this representation is shown in Figure 16. As shown, each node of the graph is an object-oriented database

object which represents a subject, predicate or object and where each edge on the graph is a database system reference pointing to another database object.

Id	Language	Name	outedges
1		“Pete Docter”	
2		“Andrew Stanton”	
3		“Bob Smith”	
4		“2008”	
5		“1940”	

Figure 14: OODB Literals Type Object Instances

Id	namespace	localname	outedges
6	“ex.ca/r/”	“Wall-E”	12 -> 7 13 -> 8 14 -> 9 15 -> 4 16 -> 10 17 -> 11
7	“ex.ca/r/”	“Wall-E 2”	18 -> 10 19 -> 11
8	“ex.ca/r/”	“PeteDocter”	21 -> 1
9	“ex.ca/r/”	“AndrewStanton”	22 -> 2
10	“ex.ca/r/”	“Disney”	20 -> 9 24 -> 5
11	“ex.ca/r/”	“BobSmith”	23 -> 3
12	“ex.ca/p/”	“Sequel”	
13	“ex.ca/p/”	“Writer”	
14	“ex.ca/p/”	“Writer”	
15	“ex.ca/p/”	“YearReleased”	
16	“ex.ca/p/”	“MadeBy”	
17	“ex.ca/p/”	“Director”	
18	“ex.ca/p/”	“MadeBy”	
19	“ex.ca/p/”	“Director”	
20	“ex.ca/p/”	“ChiefWriter”	
21	“ex.ca/p/”	“Name”	
22	“ex.ca/p/”	“Name”	
23	“ex.ca/p/”	“Name”	
24	“ex.ca/p/”	“YearFounded”	

Figure 15: OODB References Type Object Instances

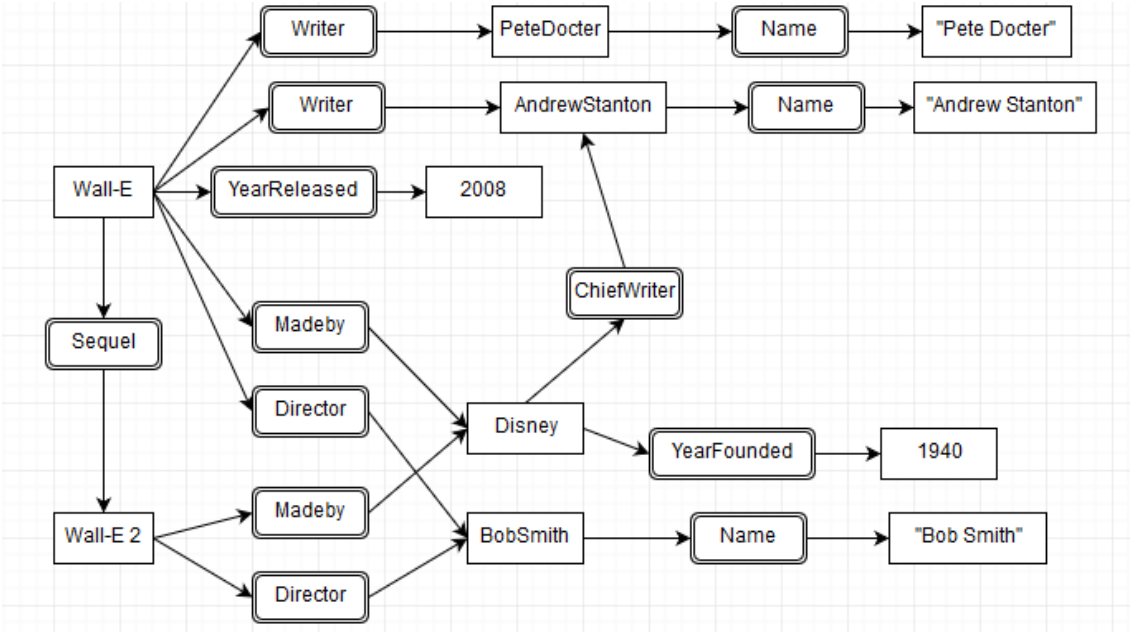


Figure 16: Example Instance of Object-Oriented

To query RDF data in the object-oriented approach, a graph query (such as SPARQL or RQL) is translated into object query language (OQL). Graph query languages such as RQL or SPARQL have a close relationship to OQL, which allows for simple translation and processing of queries written in RDF graph language into OQL [51]. In this approach, queries do not have to be expanded into basic graph patterns (or simple triple statements) as with relational-backed systems or triplestores. In the object-oriented approach querying can retrieve results with the same performance as SQL based approaches, but with much less optimization required [51]. An example of a SPARQL query written in OQL and the results returned is shown below.

SPARQL Query

```
SELECT *
WHERE
  {
    ?Movie   <ex.ca/p/Writer>  <ex.ca/r/PeteDocter> .
  }
```

Step 1 – Equivalent Pseudo OQL query

```
SELECT (r.namespace + r.localname) as movie
FROM Resources r
WHERE ((SELECT ID
        FROM Resources
        WHERE namespace = "ex.ca/p/" AND localname = "Writer"),
       (SELECT ID
        FROM Resources
        WHERE namespace = "ex.ca/r/" AND localname = "PeteDocter"))
            in r.outedge
```

Step 2 – Execute and return results

movie
<ex.ca/r/Wall-E>

The pros and cons for the object-oriented database storage method are shown below.

Pros

- Adding new data doesn't require the underlying data structures change.
- The most natural way of storing RDF data as it is represented as a graph on disk.
- Faster query times because when data is stored directly as a graph we can load it directly into memory without the need to parse. (For example a triplestore would require the data be parsed before the graph can be loaded into memory)

Cons

- No commercially available systems.
- Pointer references are difficult and cumbersome to manage.
- Despite having superior object-oriented modeling abilities, object-oriented approaches do not represent the RDFS as database schema in storage.

3.4 Object-Relational Storage

Object-relational databases have mostly been used in RDF storage as substitutes for relational databases. That is to say only their relational features are used. For example, RDF storage systems such as rdf4j (sesame) [40], ohStore [47], Jena1 [1] and Jena2 [41] are relational-backed and make use of relational database management systems, but are also able to use object-relational database systems by only utilizing the relational features of OORDB systems such as of postgreSQL [95] or Oracle.

Object-relational databases however have schema which enable much better RDFS representation thanks to such features as property inheritance for representing RDFS and OWL class hierarchy, support for multi-valued attributes and representing subject-object relationships with object pointer references. From our thorough investigation it was discovered that there is currently no commercial RDF storage software which make use of the schema of object-relational databases to represent RDFS; however, two non-commercial attempts to represent OWL using object-relational semantic features have been observed.

One proposal uses an object-relational database and suggests an approach that is a variation of the N-ary relational approach [53]. In this proposed approach property inheritance is used to represent OWL classes and multi-valued attributes are supported. Furthermore, object pointers (references) are used to represent subject-object relationships. Another paper proposed a similar N-ary style approach using an object-relational database which also makes use of property inheritance and multi-valued

attributes [54]. An example object-relational database schema for both these proposals using the working example is shown below.

Object-Relational Schema

```
//Define attributes
CREATE TYPE KeywordArray as VARRAY(20) OF REF VARCHAR;
CREATE TYPE movieDirectorArray as VARRAY(20) of REF MovieDirector_t;
CREATE TYPE movieWriterArray as VARRAY(20) of REF MovieWriter_t;
CREATE TYPE sequelArray as VARRAY(20) of REF Movie_t;
CREATE TYPE MadeByArray as VARRAY(20) of REF Company_t;

//Define class types
CREATE TYPE Movie_t as OBJECT (
  ID REF (Movie_t),
  Director movieDirectorArray,
  Writer movieWriterArray,
  YearReleased KeywordArray,
  Sequel sequelArray,
  MadeBy MadeByArray
);

CREATE TYPE Company_t as OBJECT(
  ID REF(Company_t),
  YearFounded KeyworkArray,
  ChiefWriter movieWriterArray
);

CREATE TYPE Person_t(
  ID REF(Person_t),
  Name KeywordArray
);

CREATE TYPE MovieWriter_t SUPERCLASS (Person_t)();
CREATE TYPE MovieDirector_t SUPERCLASS (Person_t)();

//Create tables for class types
CREATE TABLE Movie OF Movie_t;
CREATE TABLE Company OF Company_t;
CREATE TABLE Person OF Person_t;
CREATE TABLE MovieWriter OF MovieWriter_t;
CREATE TABLE MovieDirector OF MovieDirector_t;
```

A possible reason why reference pointers are so rarely used to represent RDF data in storage, despite superior semantic representation, is because with current commercially available systems working with them can be too difficult and cumbersome.

In both object-relational proposals how to query the RDF data was not discussed. Because there are many similarities between these proposals and the relational N-ary approach, it is likely that querying on these architectures would be very similar.

The pros and cons of the object-relational method are shown below.

Pros

- Makes best use of the database schema for representing RDF schema of all disk based systems.
    - RDFS/OWL classes represented as OORDB classes
    - Property inheritance
    - Able to capture multi-valued data
    - References to represent subject-object relationships
- Self-joins on subject column can be avoided [17].

Cons

- No commercial implementations available.
- For very large RDF data stores, the number of tables used to store data can quickly get out of hand [38].
- every time a new predicate or type of thing is added to the database you must change the schema [17]
- Because the predicates are elevated as column names and subject/objects are normal values stored inside the table, there is a weakness when asking queries that do not specify the predicate value in the query [90]. For queries that don't specify the predicate value the whole table must be analyzed to return the result of the query, which can reduce performance especially when the dataset has many different predicates values (columns in the N-ary table).

3.5 Expert System Storage

Expert Systems are in-memory database systems that are used in artificial intelligence and knowledge reasoning applications which support forward and backward chaining inference and truth maintenance. The feature of truth maintenance is especially

novel in this approach for enforcing constraints imposed by RDFS and OWL vocabularies. Expert system RDF storage solutions are not as common or as practical as relational storage solutions because they are limited by memory.

There are two expert system RDF stores that have identified from a thorough investigation, both of which are sister applications: R-DEVICE [55] for storing RDFS and O-DEVICE [56, 57] for storing OWL ontologies. Both of these systems have nearly identical storage architecture, the main difference being that each is specialized for storing either RDFS or OWL ontologies. Their architecture and query processing are shown in examples below.

Both systems represent RDF data similarly to the semantic object-relational N-ary method, except uses an expert system named CLIPS [96] to model data using objects. In their representation each RDFS or OWL class is represented as a CLIPS object class whose attributes are then assigned RDFS or OWL properties. The main difference between this approach and the object-relational approach is that subject-object relationships are represented slightly differently. For R-DEVICE and O- DEVICE to represent a subject-object relationship they store both the RDF's triples object value as well as an array of object pointers for which they reference.

R-DEVICE and O-DEVICE both have identical storage schemas. An example of what these object-oriented schema would look like is shown below.

R-DEVICE/O-DEVICE Schema

```
//Define our classes
CREATE CLASS Movie (
  Subject string,
  <ex.ca/p/Director> stringArray,
  <ex.ca/p/Writer> stringArray,
  <ex.ca/p/YearReleased> integer,
```

```
  <ex.ca/p/Sequel> stringArray,
  <ex.ca/p/MadeBy> stringArray,
  LinkedObjects MultiArray (objectName, objectPointer);
);

CREATE CLASS Company (
  Subject string,
  <ex.ca/p/YearFounded> integer,
  <ex.ca/p/ChiefWriter> stringArray,
  LinkedObjects MultiArray (objectName, objectPointer);
);

CREATE CLASS Person(
  Subject string,
  <ex.ca/p/Name> stringArray
);

CREATE CLASS MovieWriter SUPERCLASS (Person)();
CREATE CLASS MovieDirector SUPERCLASS (Person)();
```

The way in which subject-object relationships are stored in R-DEVICE and O-DEVICE is done using a CLIPS "MultiArray", shown in the example above. The "MultiArray" stores pairs of triples' object and the CLIPS object reference which it references. An example instance for the running "Wall-E" example is shown in the Figure 17. In this example the "ChiefWriter" for the company "Disney" has the MultiArray values of "<ex.ca/r/AndrewStanton>" (which is the triples object) and the reference id "4", which references the "<ex.ca/r/AndrewStanton>" object instance of the movieWriter class.

Movie Class

Object ID	Subject	Director	Writer	Year Released	Sequel	MadeBy
1	<ex.ca/r/Wall-E>	(<ex.ca/r/BobSmith>, 6)	(<ex.ca/r/AndrewStanton>, 4) (<ex.ca/r/PeteDocter>, 5)	2008	(<ex.ca/r/Wall-E 2>, 2)	(<ex.ca/r/Disney>, 3)
2	<ex.ca/r/Wall-E 2>	(<ex.ca/r/BobSmith>, 6)				(<ex.ca/r/Disney>, 3)

Company Class

Object ID	Subject	Yearfounded	ChiefWriter
3	<ex.ca/r/Disney>	1940	(<ex.ca/r/AndrewStanton>, 4)

Person Class

Object ID	Subject	Name

movieWriter Class

Object ID	Subject	Name
4	<ex.ca/r/AndrewStanton>	"Andrew Stanton"
5	<ex.ca/r/PeteDocter>	"Pete Docter"

movieDirector Class

Object ID	Subject	Name
6	<ex.ca/r/BobSmith>	"Bob Smith"

Figure 17: Expert System Storage Instance Example

Both R-Device and O-DEVICE use the same deductive rule language for querying over RDFS/OWL instances which is similar RDF graph languages. The syntax of the deductive rule is shown in step 1 of the example query below. Each deductive rule is translated to a CLIPS production rule and executed on the underlying CLIPS database. When querying, a mechanism outside of the CLIPS database system is required to perform the subject-object join computation.

A sample SPARQL query written for the RDFS running example using an R-DEVICE deductive rule and its query results are shown below. If our running example were to be constructed in OWL (as shown in section 2.1.3), the O-DEVICE deductive rule and results would be exactly the same as the one shown below.

SPARQL Query

```
SELECT *
WHERE
  {
    ?movie a Movie
    ?movie   <ex.ca/p/Director>  ?z .
    ?movie   <ex.ca/p/YearReleased> "2008" .
  }
```

Step 1 – Write Deductive Rule (R-Device Query)

```
(deductiverule q1
(Movie ?movie
  (<ex.ca/p/YearRealeased> ?x&:(str-index "2008" ?x))
  (<ex.ca/p/Director> ?z))
 =>
(result (movie ?movie)
        (z ?z)))
```

Step 2 – Execute and return results

movie	?z
<ex.ca/r/Wall-E>	<ex.ca/r/BobSmith>

The pros and cons for expert system RDF storage are shown below.

Pros

- Makes best use of the underlying storage architecture for representing RDF schema.
    - RDFS/OWL classes represented as CLIPS classes.
    - Property inheritance
    - Able to capture multi-valued data
    - References to represent subject-object relationships
- Self-joins on subject column can be avoided [17].

Cons

- Limited by system memory. Because it is memory based it cannot store as large volumes of data as object-relational approaches.
- Subject-object joins must be managed by middleware outside the expert system.
- For very large RDF data stores, the number of tables used to store data can quickly get out of hand [38].
- Every time a new predicate or type of thing is added to the database you must change the schema [17]
- Because the predicates are elevated as column names and subject/objects are normal values stored inside the table, there is a weakness when asking queries that do not specify the predicate value in the query [90]. For queries that don't

specify the predicate value the whole table must be analyzed to return the result of the query, which can reduce performance especially when the dataset has many different predicates values (columns in the N-ary table).

Chapter 4

Using INMDB for the Semantic Web

This chapter describes how INMDB is used for storing RDF data and is outlined as follows. Section 4.1 introduces how RDFS is stored in INMDB. Section 4.2 describes how to query RDF data in INMDB using IQL. Furthermore, it compares SQL to IQL and analyzes whether INMDB can support all RDF queries.

4.1 RDFS Storage in INMDB

RDFS storage in INMDB is similar to the relational N-ary (class property table) approach discussed in section 3.2.3 and object-relational approach discussed in section 3.3. This representation offers a better way to represent RDFS then existing approaches by making use of several features in INMDB such as property inheritance, sub-attributes, multi-valued attributes and linking. Most notably, the features of sub-attributes and sub-linking distinguish this representation from current relational and object-relational methods.

In this approach every RDFS class is represented as an INMDB class. In view of the fact that INMDB can support multiple super-classes and sub-classes; the RDFS:SubclassOf definitions are captured as INMDB subclasses using the property inheritance of INMDB. These subclasses automatically inherit all the attributes of the super-class in INMDB, facilitating schema management. Logical inconsistencies such as circular inheritance and duplicate classes are detected and rejected by INMDB.

The triples subject serves as the unique identifier for INMDB object instances of this representation because subjects in RDF are unique identifiers. The object identifier is used to represent the subject.

Each RDFS class is assigned a set of attributes/links determined by the rdf:Property, which are assigned to the class via the rdf:domain and whose range is determined by the rdf:range. The rdf:range has two forms: literals and object mappings.

RDF literals are stored as multi-valued attributes in INMDB. Some RDF datatypes can be cast to INMDB data types. The types of casting which are accepted in INMDB are shown in Table 12. Not all RDFS datatypes can be cast to INMDB datatypes (for example units of measurement) and in these cases the string value is stored for the literal.

Table 12: RDF to INMDB Datatypes

RDF datatype	INMDB datatype
xsd:string	STRING[18]
xsd:decimal xsd:double xsd:float	REAL[19]
xsd:byte xsd:short xsd:integer xsd:long	INT[20]
xsd:Date	DATE

RDF object mappings are represented using links. The target INMDB class with which we are linking to is specified by the rdf:Range which states the RDFS:class. INMDB linking can target multiple classes. Additionally, any sub-classes of the target class are included as targets. When storing a link within an INMDB object instance the

[18] STRING data type can store an unlimited size string.
[19] REAL data type can store an unlimited size floating point number.
[20] INT data type can store an unlimited size integer

subject URI of the target object being linked is stored and the class that the object belongs to is also specified.

Both literals and object mappings in RDF can have RDFS:SubPropertyOf. This is represented in INMDB using sub-attributes, which can be used on both attributes and links. The nature of RDFS allows for multiple parent attributes. The current version of INMDB is limited to only single parent attribute[21]; however, future versions of INMDB will likely support multi-parent attributes.

At an abstract level INMDB representation can be seen as a graph. INMDB object instance represents an RDF graph's subject nodes, INMDB literal attributes represent RDF graph edges to literal object nodes, and INMDB object mapping links represent RDF graph edges to other subject nodes.

An IDL schema for the working "Wall-E" example is shown below, along with several IML insert statements which insert the RDF triples into the database. The resulting object instances are shown in Figure 18. In this example five classes are generated: Movie, Company, Person, Writer (role based) and Director (role based). Property inheritance occurs for Writer and Director, which are both role based classes that inherit from the super-class Person. Sub-Attribute is shown in the Company class, where the relationship Writer is a super-attribute of ChiefWriter. Both Writer and ChiefWriter in this case target the class Writer and are multi-valued. The attributes YearReleased, Yearfounded and Name are multi-valued indicated by the asterix (*).

INMDB IDL Schema

[21] This limitation was not an issue when storing DBPedia as the dataset did not contain multiple parent properties.

```
create class Movie [
  role Director(inverse "as") (N:N): Person(inverse directs),
  role Writer(inverse "as") (N:N): Person(inverse writes),
  @"YearReleased"*:int,
  normal Sequel (N:N): Movie,
  normal MadeBy (N:N): Company
];

create class Company [
  @"YearFounded"*:int,
  role  Writer (N:N) ->{ ChiefWriter (N:N)}: Movie.Writer
];

create class Person[
  @"Name"*: string
]
```

INMDB IML statements

```
Insert Movie "<ex.ca/r/Wall-E>" [
  role Director: {"<ex.ca/r/BobSmith>"},
  role Writer: {"<ex.ca/r/AndrewStanton>", "<ex.ca/r/PeteDocter>"},
  @YearReleased:2008,
  normal Sequel:{"<ex.ca/r/Wall-E 2>"},
  normal MadeBy: {"<ex.ca/r/Disney>"}
];

Insert Movie "<ex.ca/r/Wall-E 2>" [
  role Director: {"<ex.ca/r/BobSmith>"},
  normal MadeBy: {"<ex.ca/r/Disney>"}
];

Insert Company "<ex.ca/r/Disney>" [
  @"YearFounded":1940 ,
  role ChiefWriter: {"<ex.ca/r/AndrewStanton>"}
];

Insert Person "<ex.ca/r/AndrewStanton >" [
  @"Name": "Andrew Stanton"
];

Insert Person "<ex.ca/r/PeteDocter>" [
  @"Name": "Pete Docter"
];

Insert Person "<ex.ca/r/BobSmith>" [
  @"Name": "Bob Smith"
];
```

Movie Class

Object ID	Director	Writer	Year Released	Sequel	MadeBy
<ex.ca/r/Wall-E>	<ex.ca/r/BobSmith>	<ex.ca/r/AndrewStanton>, <ex.ca/r/PeteDocter>	2008	<ex.ca/r/Wall-E 2>	<ex.ca/r/Disney>
<ex.ca/r/Wall-E 2>	<ex.ca/r/BobSmith>				<ex.ca/r/Disney>

Company Class

Object ID	Yearfounded	ChiefWriter
<ex.ca/r/Disney>	1940	<ex.ca/r/AndrewStanton>

Person Class

Object ID	Name	
<ex.ca/r/AndrewStanton>	"Andrew Stanton"	role: Writer
<ex.ca/r/PeteDocter>	"Pete Docter"	role: Writer
<ex.ca/r/BobSmith>	"Bob Smith"	role: Director

Figure 18: INMDB N-ary Object Instances Example

4.2 RDF Queries in INMDB

This section shows how to use INMDB for querying RDF data. An analysis of whether or not INMDB's query language IQL can support RDF graph queries for the storage approach proposed in section 4.1 is shown. In order to properly analyze whether IQL can support SPARQL graph queries, a comparison to SQL is made. The reason for this comparison is that SQL 3.0 is fully SPARQL 1.1 compatible and all SPARQL 1.1 queries can be translated into an equivalent SQL query [94]. The comparison is made between these two languages (IQL and SQL) because the translation process for SPARQL to IQL is very similar to that of SPARQL to SQL. Furthermore, by demonstrating SQL and IQL graph queries side by side the simplicity of IQL's query syntax can be demonstrated.

The SQL queries are written for the example "Wall-E" relational N-ary instance given in section 3.2.3.1, whereas the IQL queries are written for the example "Wall-E" INMDB instance given in section 4.1.

This section is outlined as a follows. Section 4.2.1 provides a systematic way to translate SPARQL queries into IQL. Section 4.2.2 illustrates with examples each of the different SPARQL language graph patterns translated into both SQL and IQL, the goal of which is to determine if all SPARQL graph patterns can be translated into IQL. Finally, the section 0 summarizes the findings for translating SPARQL into IQL.

4.2.1 Translating SPARQL to IQL

In this section we describe a generic algorithm for translating SPARQL queries into the IQL query language for INMDB. Because both SPARQL and IQL are declarative graph languages, the translation from SPARQL to IQL is straightforward.

4.2.1.1 Translating SPARQL to IQL Pseudo Code

As input to this algorithm:

- SPARQL expression

For INMDB, the following procedure is used to generate IQL from SPARQL:

1. Simplify the SPARQL.
2. Build a tree representation of the SPARQL
3. Generate IQL from the tree

Notes:

- Currently left join is not translated by the algorithm

  This is because the current solution for querying a left join in INMDB requires that we specify for a value to be both null and not null (see example optional join query in section 4.2.2.6). This workaround is not optimal and should not be

considered the final way to translate “left joins” into IQL. Future work will include translating “left joins” to IQL in INMDB when a “left join” operator is added. “left join” or “optional join” graph patterns will be manually translated using the workaround in this thesis.

- Some filters are not supported by INMDB at this time such as:

  Regex operators (^, \d, \D, $, …)

  Comparisons for datatypes which cannot be cast to int, string or Date in storage.

- Bind operator currently not supported by INM

The pseudo code for the steps is as follows:

INMDB SPARQL to IQL Pseudo Code

```
//Expand and simplify SPARQL statement
Expand syntax forms for IRIs literals and triple patterns
Convert property path patterns to triples
Convert each sub-query pattern into a sub-select statement.
Re-Nest each unioned or minus pattern as a tree pattern
Remove duplicate patterns

Store the resulting expression in the variable simplified_SPARQL


//Generate a SPARQL Tree function
def build_SPARQL_tree(node n, SPARQL s):
  Parse s by graph-pattern:
    If “select with a where clause” pattern:
      Store project, offset/limit and order by values in n
      Remove this select/where statement from s
      Add a child node ns to n,
      Call build_SPARQL_tree(ns, s)
    If “sub-select” pattern:
      Remove the select statement from s
      Add a child node ns to n
      Call build_SPARQL_tree(ns, s)
    If union or minus pattern:
      Remove the union/minus statement from s
```

```
        Put the left child SPARQL pattern in s1
        Put the right child SPARQL pattern in s2
        Add two children n1 and n2 to n
        Call build_SPARQL_tree(n1, s1)
        Call build_SPARQL_tree(n2, s2)
      If triple pattern:
         Group up triples by subject-subject
         Elevate the class definitions to parent of the subject groups
         Connect subject-object triples
         apply filters to each variable
         Store connected expression in n
         Return

//Generate a SPARQL Tree
Create an empty SPARQL node t
call build_SPARQL_tree(t, simplified_SPARQL)

//Generate IQL function
def generate_IQL(node t):
  create empty string s
  If t is “union” node:
    s = call generate_IQL(left child of t) + “| ” + call generate_IQL(right child of t)
  If t is “minus” node:
    s = call generate_IQL(left child of t) + “, not” + call generate_IQL(right child of t)
  If t is “sub-select” node:
    s = call generate_IQL(child of t)
  If “select with a where clause” node:
    s =  call generate_IQL(child of t) + “ construct”
    s += project, offset/limit and order by IQL expression from variables stored in t
  If “sub-select” node:
    s = call generate_IQL(child of t)
  If “triple pattern” node:
    s = connected pattern stored in t as IQL expression
  return s

//Generate IQL
return “query ” + call generate_IQL(t)
```

4.2.1.2 Various Translation Examples

4.2.1.2.1 Example 1

SPARQL query

```
SELECT *
```

```
WHERE
  {
    {Select *
     {
      ?m   a     Movie .
      ?m   <ex.ca/p/sequel>  <ex.ca/r/Wall-E 2> .
      ?m   <ex.ca/p/yearReleased> ?yearReleased .
      ?m   <ex.ca/p/Writer> ?x .
      ?x   <ex.ca/p/Name> ?name .
     }
   UNION
    {Select *
     {
      ?c   a     Company .
      ?c   <ex.ca/p/ChiefWriter> ?y .
      ?y   <ex.ca/p/Name> ?name .
     }
    }
   Filter (Regex(?name,"B*"))
  }
```

Step 1 - Simplified SPARQL

- Expand syntax forms for IRIs literals and triple patterns
- Convert property path patterns to triples
- Convert each sub-query pattern into a sub-select statement.
- Re-Nest each unioned or minus pattern as a tree pattern
- Remove duplicate patterns

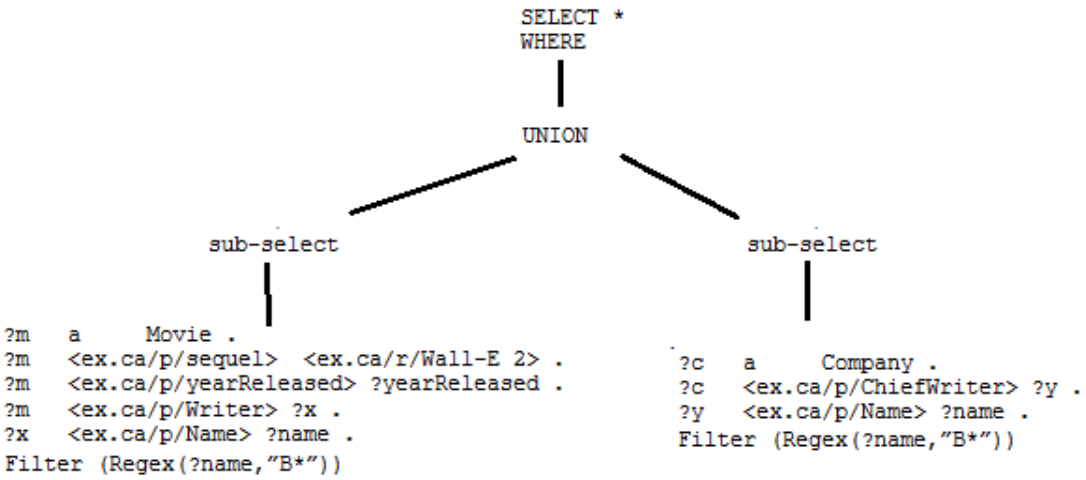
Results in simplified sparql:

```
SELECT *
WHERE
  {
    UNION (
    {sub-select *
     {
      ?m   a     Movie .
      ?m   <ex.ca/p/sequel>  <ex.ca/r/Wall-E 2> .
      ?m   <ex.ca/p/yearReleased> ?yearReleased .
      ?m   <ex.ca/p/Writer> ?x .
      ?x   <ex.ca/p/Name> ?name .
    }
   ,
    {sub-select *
     {
      ?c   a     Company .
```

```
    ?c    <ex.ca/p/ChiefWriter> ?y .
    ?y    <ex.ca/p/Name> ?name .
   }
  }
 )
 Filter (Regex(?name,"B*"))
}
```

Step 2 – Build Tree

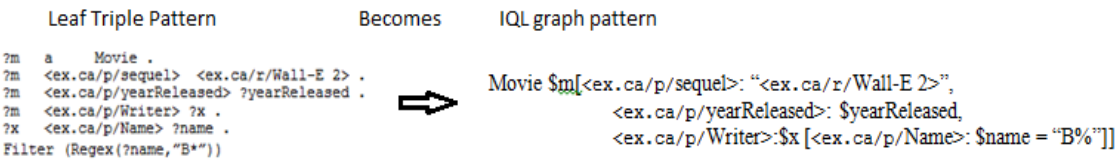
- Recursively build tree using function build_SPARQL_tree.



Step 3 – Build IQL

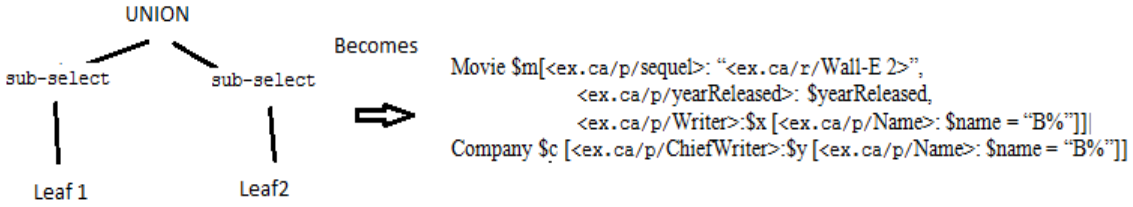
- Recursively build IQL statement using function generate_IQL.
- In this step, we recursively navigate through the tree down to each leaf which is a triple pattern.
- The triple patterns in the leaves are converted into an IQL graph pattern

e.g.,



- Tracing back through the recursion, leaves are joined together at their union, minus or sub-select node.

  e.g.,



- Finally, when the recursion returns to the "select where" statement, the construct argument is added which is the direct translation of project variables, aggregates and their ordering.

  Results in IQL:

  Query
  Movie $m[<ex.ca/p/sequel>: "<ex.ca/r/Wall-E 2>",
  <ex.ca/p/yearReleased>: $yearReleased,
  <ex.ca/p/Writer>:$x [<ex.ca/p/Name>: $name = "B%"]]|
  Company $c [<ex.ca/p/ChiefWriter>:$y [<ex.ca/p/Name>: $name = "B%"]]
  Construct $m[$x[$name]],$c[$y[$name]];

4.2.1.2.2 Example 2

SPARQL query

```
SELECT *
WHERE
  {
      ?c    a      Company .
      ?c    <ex.ca/p/ChiefWriter> ?y .
      ?y    <ex.ca/p/Name> ?name .
  }
```

Step 1 - Simplified SPARQL

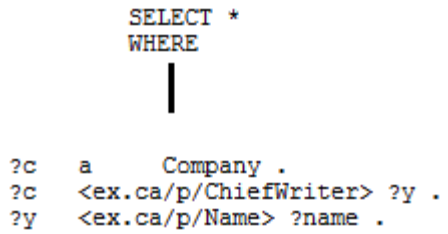
- Expand syntax forms for IRIs literals and triple patterns
- Convert property path patterns to triples
- Convert each sub-query pattern into a sub-select statement.
- Re-Nest each unioned or minus pattern as a tree pattern
- Remove duplicate patterns

  Results in simplified sparql:

```
SELECT *
WHERE
  {
      ?c   a     Company .
      ?c   <ex.ca/p/ChiefWriter> ?y .
      ?y   <ex.ca/p/Name> ?name .
  }
```

Step 2 – Build Tree

- Recursively build tree using function build_SPARQL_tree.



Step 3 – Build IQL

- Recursively build IQL statement using function generate_IQL.
- In this step, we recursively navigate through the tree down to each leaf which is a triple pattern.
- The triple patterns in the leaves are converted into an IQL graph pattern.

- Finally, when the recursion returns to the “select where” statement, the construct argument is added which is the direct translation of project variables, aggregates and their ordering.

  Results in IQL:

  Query
  Company $c [<ex.ca/p/ChiefWriter>:$y,[<ex.ca/p/Name>: $name]]
  Construct $c[$y[$name]]

4.2.1.2.3 Example 3

SPARQL query

```
SELECT *
WHERE
  {
    {Select *
     {
      ?c   a     Company .
      ?c   <ex.ca/p/ChiefWriter> ?y .
      ?y   <ex.ca/p/Name> ?name .
     }
    }
    MINUS
    {Select *
     {
      ?p   ?x   ?z .
      ?p   <ex.ca/p/Name> ?name .
     }
  }
```

Step 1 - Simplified SPARQL

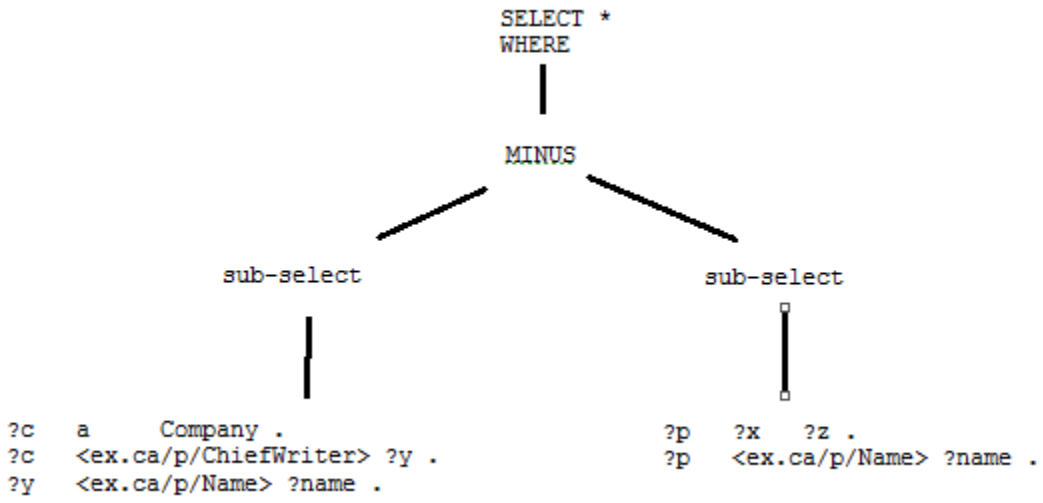
- Expand syntax forms for IRIs literals and triple patterns
- Convert property path patterns to triples
- Convert each sub-query pattern into a sub-select statement.
- Re-Nest each unioned or minus pattern as a tree pattern
- Remove duplicate patterns

Results in simplified sparql:

```
SELECT *
WHERE
  {
    MINUS(
   { sub-select *
     {
      ?c    a      Company .
      ?c    <ex.ca/p/ChiefWriter> ?y .
      ?y    <ex.ca/p/Name> ?name .
     }
    }
    ,
    { sub-select *
     {
      ?p    ?x    ?z .
      ?p    <ex.ca/p/Name> ?name .
     }
    )
  }
```

Step 2 – Build Tree

- Recursively build tree using function build_SPARQL_tree.

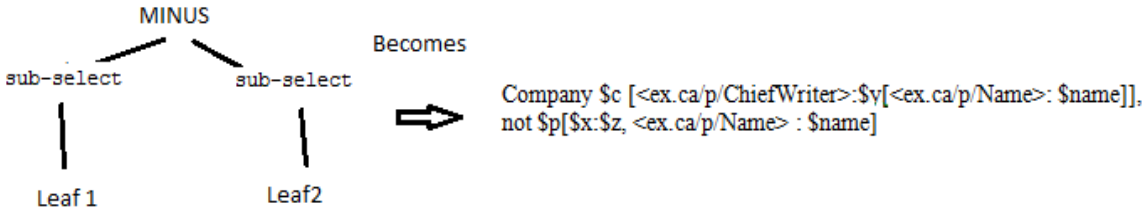


Step 3 – Build IQL

- Recursively build IQL statement using function generate_IQL.

- In this step, we recursively navigate through the tree down to each leaf which is a triple pattern.
- The triple patterns in the leaves are converted into an IQL graph pattern.
- Tracing back through the recursion, leaves are joined together at their union, minus or sub-select node.

  e.g.,



- Finally, when the recursion returns to the "select where" statement, the construct argument is added which is the direct translation of project variables, aggregates and their ordering.

  Query
  Company $c [<ex.ca/p/ChiefWriter>:$y[<ex.ca/p/Name>: $name]],
  not $p[$x:$z, <ex.ca/p/Name> : $name]
  Construct $c[$y[$name]], $p[$x\$z, $name]

4.2.1.2.4 Example 4

SPARQL query

```
SELECT *
WHERE
  {

      ?c   a     Company .
      ?c   ?x   ?y .
      ?y   <ex.ca/p/Name>  ?name .
      ?y   <ex.ca/p/Name>  ?name .
  }
```

Step 1 - Simplified SPARQL

- Expand syntax forms for IRIs literals and triple patterns
- Convert property path patterns to triples
- Convert each sub-query pattern into a sub-select statement.
- Re-Nest each unioned or minus pattern as a tree pattern
- Remove duplicate patterns

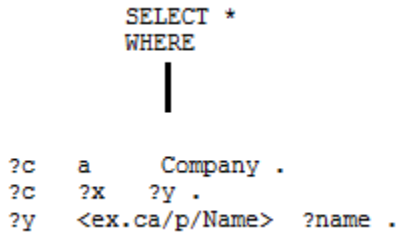
Results in simplified sparql:

```
SELECT *
WHERE
  {

      ?c    a      Company .
      ?c    ?x    ?y .
      ?y    <ex.ca/p/Name>   ?name .
  }
```

Step 2 – Build Tree

- Recursively build tree using function build_SPARQL_tree.



Step 3 – Build IQL

- Recursively build IQL statement using function generate_IQL.
- In this step, we recursively navigate through the tree down to each leaf which is a triple pattern.
- The triple patterns in the leaves are converted into an IQL graph pattern.

- Finally, when the recursion returns to the "select where" statement, the construct argument is added which is the direct translation of project variables, aggregates and their ordering.

```
Query
Company $c [?x:$y[<ex.ca/p/Name>: $name]]
Construct $c[?x\$y[$name]]
```

4.2.1.3 SPARQL to IQL Time Complexity

This section describes the time complexity for translating SPARQL to IQL.

Inputs:

- N – total number of elements in the SPARQL.
- T – number of patterns in the SPARQL
- S - select, sub-select, minus and union in the SPARQL
- k – maximum number of elements in a triple pattern

The time complexity for translating SPARQL to IQL is as follows:

- Step1- simplify SPARQL
  - $O(N^2)$
- Step 2 – Build SPARQL tree representation
  - O((S+T) * k * N)
- Step3 – Generate IQL
  - O(T * k)

Total time complexity is:

- $O(N^2 + T * k * N + (T * k))$

Simplified to:

- $O(N^3)$

4.2.2 INMDB Support for SPARQL 1.1 Queries

SPARQL queries are graph patterns composed of basic graph patterns. In order to translate a SPARQL query into an IQL query, we require that IQL be able to represent all the basic graph patterns. SPARQL is directly translated into IQL. Translating SPARQL to SQL however, first requires SPARQL be translated into SPARQL algebra.

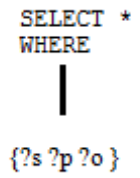
The following sections will illustrate each of the fifteen basic graph patterns and how they are directly translated into IQL. Furthermore, their SPARQL algebra translation is shown with its corresponding SQL translation. The queries selected are designed to be representative in that they make use of all features that SPARQL 1.1 provides.

4.2.2.1 Basic Graph Pattern

The basic graph pattern in Table 13 where no predicate is specified is supported by INMDB.

Table 13: SPARQL Pattern to Algebra/SQL/IQL – BGP 1

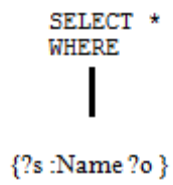
SPARQL Pattern	SELECT * WHERE {?s ?p ?o }
SPARQL Algebra	BGP(?s ?p ?o)
SQL Translation	(SELECT subject as s, “Name” as p, Name as o FROM Person) UNION

	(SELECT subject as s, “Name” as p, Name as o FROM movieWriter) UNION (SELECT subject as s, “Name” as p, Name as o FROM movieDirector) UNION (SELECT subject as s, “Writer” as p, Writer as o FROM Writer) UNION (SELECT subject as s, “yearFounded” as p, yearFounded as o FROM Company) UNION (SELECT subject as s, “ChiefWriter” as p, ChiefWriter as o FROM Company) UNION (SELECT subject as s, “Director” as p, Director as o FROM Movie) UNION (SELECT subject as s, “YearReleased” as p, YearReleased as o FROM Movie) UNION (SELECT subject as s, “Sequel” as p, Sequel as o FROM Movie) UNION (SELECT subject as s, “MadeBy” as p, MadeBy as o FROM Movie)
Tree	
IQL Translation	Query $s[$p:$o] construct $s/$p/$o

In Table 14 the graph pattern does not specify which class we are querying which requires us to query all tables that have the predicate value specified. In INMDB this is a very easy syntax compared to SQL.

Table 14: SPARQL Pattern to Algebra/SQL/IQL – BGP 2

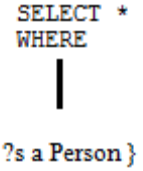
SPARQL Pattern	SELECT * WHERE {?s :Name ?o }
SPARQL Algebra	BGP(?s :Name ?o)
SQL Translation	(SELECT subject, Name

	FROM Person) UNION (SELECT subject, Name FROM movieWriter) UNION (SELECT subject, Name FROM movieDirector)
Tree	
IQL Translation	Query $s ["Name":$o] Construct $s[$o];

The graph pattern in Table 15 is well supported by INMDB. Querying a class and its subclasses is made very easy to using INMDBs ability to query a class and its subclasses.

4.2.2.2 Basic Graph Pattern Belonging to a Class and it's Subclasses

Table 15: SPARQL Pattern to Algebra/SQL/IQL – Class and Sub-Class

SPARQL Pattern	SELECT * WHERE { ?s a Person }
SPARQL Algebra	BGP(?s a Person)
SQL Translation	SELECT subject FROM Person UNION SELECT subject FROM movieDirector UNION SELECT subject FROM movieWriter
Tree	
IQL Translation	Query Person* $s

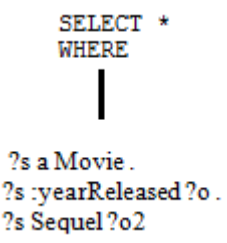
	construct $s;

The graph pattern shown in Table 15 is well supported by INMDB. INMDB's query language is specifically designed for querying classes and their subclasses, and therefore offers a much more compact query expression then in SQL. On larger more complex queries with many sub-classes this is more apparent.

4.2.2.3 Pattern with JOIN (subject-subject)

Table 16: SPARQL Pattern to Algebra/SQL/IQL – JOIN s/s

SPARQL Pattern	SELECT * WHERE
	{
	?s a Movie .
	?s :yearReleased ?o .
	?s Sequel ?o2
	}

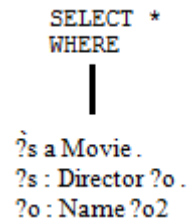
SPARQL Algebra	JOIN (BGP(?s :a Movie), JOIN (BGP(?s :yearReleased ?o), BGP(?s Sequel ?o2)))
SQL Translation	SELECT subject, yearReleased, Sequel FROM Movie
Tree	
IQL Translation	Query Movie* $s["YearReleased":$o, Sequel:$o2] construct $s[$o,$o2];

4.2.2.4 Pattern with JOIN (subject-object)

As shown in Table 17, the linking feature in INMDB simplifies the syntax of the query expression compared to what is done using SQL. Using the linking query syntax, the class upon which the subject-object join is being performed does not need to be specified. This is handled by the INMDB database management system. In SQL, the RDFS class upon which the join is being performed must be specified in the join argument. For subject-object relations that have many different class relations a query expression can easily get out of hand in SQL, but in INMDB the query syntax remains small and compact because it is not necessary to specify which class is being linked.

Table 17: SPARQL Pattern to Algebra/SQL/IQL – JOIN s/o

SPARQL Pattern	SELECT * WHERE
	{
	?s a Movie .
	?s : Director ?o .
	?o : Name ?o2
	}

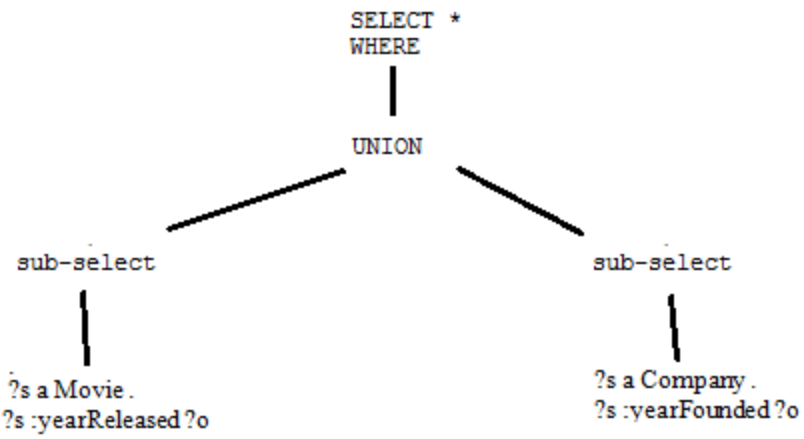
SPARQL Algebra	JOIN (BGP(?s :a Movie), JOIN (BGP(?s :Director ?o), BGP(?o :Name ?o2)))
SQL Translation	SELECT t1.subject, t1.Director, t2.Name FROM Movie as t1 JOIN movieDirector as t2 WHERE t1.Director = t2.subject
Tree	
IQL Translation	Query Movie* $s [Director:$o ["Name":$o2]] construct $s[$o[$o2]];

4.2.2.5 UNION of Two BGPs

The graph pattern with UNION shown in Table 18 is currently supported in INMDB.

Table 18: SPARQL Pattern to Algebra/SQL/IQL - UNION

SPARQL Pattern	SELECT * WHERE
	{
	{
	?s a Movie .
	?s :yearReleased ?o
	}
	UNION
	{
	?s a Company .
	?s :yearFounded ?o
	} }

SPARQL Algebra	UNION (JOIN (BGP(?s :a Movie), BGP(?s :yearReleased ?o)), JOIN (BGP(?s :a Company), BGP(?s :yearFounded ?o2)))
SQL Translation	(SELECT subject, yearReleased as o FROM Movie) UNION (SELECT subject, yearFounded as o FROM Company)
tree	
IQL Translation	Query Movie $s["YearReleased":$o] | Company $s2 ["YearFounded":$o2] Construct $s[$o],$s2[$o2]

4.2.2.6 Pattern with OPTIONAL JOIN

The graph pattern with an OPTIONAL JOIN shown in Table 19 is currently supported in INMDB, by using a workaround that involves checking for null values. The OPTIONAL JOIN feature is not supported in INMDB using a dedicated operator yet. This is because INMDB is not designed for doing relational joins; it was designed for doing relationship lookups. Future versions of INMDB will likely include a dedicated operator for performing left joins.

Table 19: SPARQL Pattern to Algebra/SQL/IQL – OPTIONAL JOIN

SPARQL Pattern	SELECT * WHERE { ?s a Movie . ?s :Director ?o . OPTIONAL {?o : Name ?o2}}
SPARQL Algebra	LEFTJOIN(JOIN (BGP(?s :a Movie), BGP(?s :Director ?o)), BGP(?o :Name ?o2), TRUE)
SQL Translation	SELECT t1.subject, t1.Director, t2.Name FROM Movie as t1 LEFT JOIN movieDirector as t2 WHERE t1.Director = t2.subject
Tree	No tree available for Optional Joins (see translation notes in section 4.2.1.1)
IQL Translation	Query Movie* $s [! Director is null \| Director:$d/ "Name":$n] construct $s[$d/$n];

4.2.2.7 Pattern with more than One OPTIONAL JOIN

The graph pattern with more than one OPTIONAL JOIN shown in Table 20 is currently supported in INMDB. However, it is currently supported using a work around which requires null checks. Future versions of INMDB will likely include a left join operator.

Table 20: SPARQL Pattern to Algebra/SQL/IQL – OPTIONAL JOIN ++

SPARQL Pattern	SELECT * WHERE { ?s a Movie . ?s :Director ?o . ?s :MadeBy ?o3 OPTIONAL {?o :Name ?o2} OPTIONAL {?o3 :YearFounded ?o4}}
SPARQL Algebra	LEFTJOIN(LEFTJOIN(JOIN(JOIN (BGP(?s :a Movie), BGP(?s :Director ?o)), BGP(?s :MadeBy ?o3)), BGP(?o :Name ?o2)), TRUE), BGP(?o3 :YearFounded ?o4), TRUE)
SQL Translation	SELECT t1.subject, t1.Director, t1.MadeBy, t2.Name, t3.yearFounded FROM (Movie as t1 LEFT JOIN movieDirector as t2) LEFT JOIN Company as t3) WHERE t1.Director = t2.subject AND t1.MadeBy = t3.subject
Tree	No tree available for Optional Joins (see translation notes in section 4.2.1.1)
IQL Translation	Query Movie* $s [! Director is null , ! MadeBy is null \| Director:$a/"Name":$o\| MadeBy:$b/"YearFounded":$oa] construct $s[$a,$b,$a/$o,$b/$oa];

4.2.2.8 Pattern with a FILTER

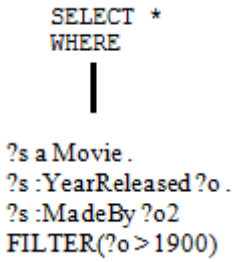
The graph pattern shown in Table 21 is able to be translated into IQL, however, there is limited application for the FILTER operation in IQL. Essentially, in order to use the built in comparators in INMDB, RDF literal data must be stored as native data types offered by INMDB. This is limited because the SPARQL FILTER clause is much more powerful than simply comparing native data type. It allows users to compare different units of measure (e.g., km and cm), perform regex on values, among many other functions.

There are as well several FILTER features which are not supported by INMDB such as filter on regex, filter EXISTS, filter NOT EXISTS, filter if a value is bound or not, filter where value equals another value and filtering where value does not equal. In

the future, INMDB will likely support these features. Some of these features are also very relevant to other fields of research (e.g., regex matching is useful for natural language processing).

Table 21: SPARQL Pattern to Algebra/SQL/IQL - FILTER

SPARQL Pattern	SELECT * WHERE
	{
	?s a Movie .
	?s :YearReleased ?o .
	?s :MadeBy ?o2
	FILTER(?o > 1900)
	}

SPARQL Algebra	JOIN(JOIN (BGP(?s :a Movie), BGP(?s :YearReleased ?o)), BGP(?o :MadeBy ?o2), (?o > 1900))
SQL Translation	SELECT t1.subject, t1.yearReleased, t1.MadeBy, FROM Movie as t1 WHERE t1.yearReleased > 1900
Tree	
IQL Translation	Query Movie* $t["YearReleased":$b > 1900, MadeBy:$c] Construct $t[$b, $c];

The SPARQL FILTER clause lets you compare literal values. For example, data stored as a string such as *"123.123"^^xsd:integer* cannot be compared in INMDB because we must first parse the string to acquire a datatype. This requires a functionality that is similar a regex match/extraction for a particular datatype and the ability to cast this value as a native datatype.

Ideally for situations like this where extraction of values from literals is required for comparison an internal mechanism would be needed in INMDB. A possible

mechanism would be a database system function similar to the gsub function found in the programming language R. This functionality is described in Figure 19.

```
//gsub function
/* gsub() function replaces all matches of a string, if the parameter is a string vector, returns a
string vector of the same length and with the same attributes (after possible coercion to
character). Elements of string vectors which are not substituted will be returned unchanged
(including any declared encoding)[22] */
gsub(regex_pattern_to_match, replacement_pattern, original_string)

//example use
gsub(“.*\”([0-9]+\.?[0-9]*)\”^^xsd:integer.*”,”\1”,“\“123.123\”^^xsd:integer”)

//here we replace the original string `“123.123”^^xsd:integer` with the integer value
`123.123`

//This gsub function is then combined with a casting function to convert the string value of
the result to an integer value, for example
as.integer(gsub(“\”([0-9]+\.?[0-9]*)\”^^xsd:integer”,”\1”,“\“123.123\”^^xsd:integer”))

//This would then allow us to compare values in INMDB which are typed literals.
//For example, the following use:

Query Movie $t1[subject:$a, yearReleased:$b, MadeBy:$c],
as.integer(gsub(“\”([0-9]+\.?[0-9]*)\”^^xsd:integer”,”\1”,$b)) > 1900
Construct $t1[$a, $b, $c]
```

Figure 19: gsub Mechanism

4.2.2.9 Pattern with UNION and OPTIONAL

The graph pattern with UNION and OPTIONAL shown in Table 22 is currently supported in INMDB.

Table 22: SPARQL Pattern to Algebra/SQL/IQL – UNION and OPTIONAL

SPARQL Pattern	SELECT * WHERE {{ { ?s a Movie . ?s :yearReleased ?o } UNION {?s a Movie . ?s :Sequel ?o2 }} OPTIONAL {?s :Writer ?o3}}

[22] http://www.endmemo.com/program/R/gsub.php
(Retrieved Feb 11, 2017)

SPARQL Algebra	LEFTJOIN(UNION (JOIN (BGP(?s :a Movie), BGP(?s :yearReleased ?o)), JOIN (BGP(?s :a Movie), BGP(?s :Sequel ?o2))), BGP(?s :Writer ?o3), TRUE)
SQL Translation	SELECT t1.subject, t1.o, t2.Writer FROM ((SELECT subject, yearReleased as o FROM Movie) UNION (SELECT subject, sequel as o FROM Movie)) as t1 LEFT JOIN Movie as t2 WHERE t1.subject = t2.subject
Tree	No tree available for Optional Joins (see translation notes in section 4.2.1.1)
IQL Translation	Query Movie $s ["YearReleased":$o | ! Sequel is null | Sequel:$ob/Writer:$oc] Construct $s[$o, $ob/$oc];

4.2.2.10 Pattern Involving BIND

The graph pattern involving BIND shown in Table 23 is not currently supported in INMDB. See "Table 6: SPARQL Query Features" for definition of BIND.

Table 23: SPARQL Pattern to Algebra/SQL/IQL - BIND

SPARQL Pattern	SELECT * WHERE { ?s a Movie . ?s :yearReleased ?o . BIND (2*?o AS ?o2)}
SPARQL Algebra	JOIN (BGP(?s :a Movie), EXTEND(BGP(?s :yearReleased ?o), ?o2, 2*?o))
SQL Translation	SELECT subject, yearReleased as o, (2*yearReleased) AS o2 FROM Movie
Tree	No tree available for Optional Joins (see translation notes in section 4.2.1.1)
IQL Translation	INMDB currently does not support binding like functionality, this will be added in future versions.

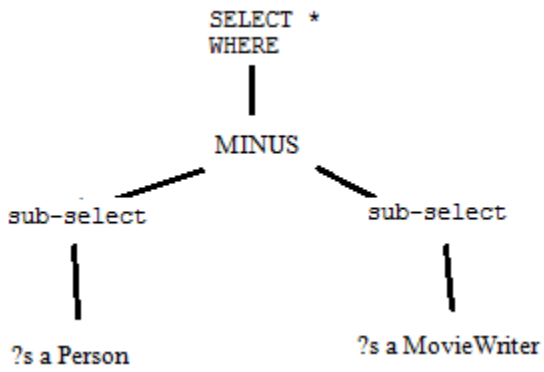
A possible IQL query syntax to represent this is shown below:

Query Movie* $x ["YearReleased":$o, $z as $o*2]
Construct $x[$o, $z];

4.2.2.11 Pattern Involving MINUS

The graph pattern involving MINUS shown in Table 24 is currently supported in INMDB. See "Table 6: SPARQL Query Features" for definition of MINUS.

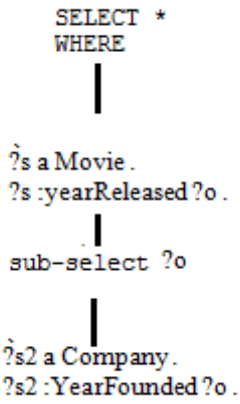
Table 24: SPARQL Pattern to Algebra/SQL/IQL - MINUS

SPARQL Pattern	SELECT * WHERE { ?s a Person . MINUS {?s a MovieWriter}}
SPARQL Algebra	MINUS (BGP(?s :a Person), BGP(?s :a MovieWriter))
SQL Translation	((SELECT subject FROM Person) UNION (SELECT subject FROM movieWriter) UNION (SELECT subject FROM movieDirector)) MINUS (SELECT subject FROM movieWriter)
Tree	
IQL Translation	Query Person* $s, not Writer $s construct $s;

4.2.2.12 Pattern Involving SUBQUERY

SUBQUERY graph patterns as shown in Table 25 are supported by INMDB. See "Table 6: SPARQL Query Features" for definition of SUBQUERY.

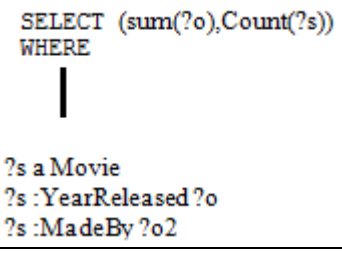
Table 25: SPARQL Pattern to Algebra/SQL/IQL - SUBQUERY

SPARQL Pattern	SELECT * WHERE { ?s a Movie . ?s :yearReleased ?o . {SELECT ?o { ?s2 a Company . ?s2 :YearFounded ?o . }}}
SPARQL Algebra	JOIN(JOIN(BGP(?s :a Movie), BGP(?s :yearReleased ?o)), ToMultiSet(PROJECT(JOIN(BGP(?s2 :a Company), BGP(?s2 :yearFounded ?o)), {?o}))
SQL Translation	SELECT t1.subject, t1.yearReleased FROM Movie as t1 JOIN (SELECT yearFounded FROM Company) as t2 WHERE t2.yearFounded = t1.yearReleased
Tree	
IQL Translation	Query Movie* $s["YearReleased":$o], Company $s2["YearFounded":$o] Construct $s[$o];

4.2.2.13 Pattern Involving GROUP BY with SUM and COUNT

GROUP BY and GROUP BY aggregates are supported by INMDB currently. However the operator HAVING, which is a filter applied in conjunction with a GROUP BY operator is not yet supported.

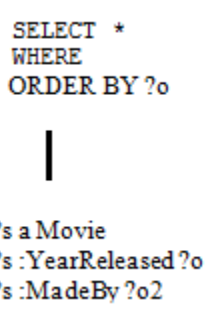
Table 26: SPARQL Pattern to Algebra/SQL/IQL – GROUP BY and Aggregate

SPARQL Pattern	SELECT (sum(?o),Count(?s)) WHERE { ?s a Movie ?s :YearReleased ?o ?s :MadeBy ?o2 } GROUP BY ?s ?o2
SPARQL Algebra	AggregateJoin(Aggregation((?o), Sum, {}, GROUP((?s, ?o2), JOIN(BGP(?s :a Movie), JOIN (BGP(?s :yearReleased ?o), BGP(?s :MadeBy ?o2))))), Aggregation((?s), Count, {}, GROUP((?s, ?o2), JOIN(BGP(?s :a Movie), JOIN (BGP(?s :yearReleased ?o), BGP(?s :MadeBy ?o2))))))
SQL Translation	SELECT Sum(YearReleased) as sum, Count(subject) as count FROM Movie GROUP BY subject, MadeBy
Tree	
IQL Translation	Query Movie* $s["YearReleased":$o, “Made By”:$o2] construct count({$s}), $s\sum({$o}) ;

4.2.2.14 Pattern Involving ORDER BY

The IQL query ORDER BY is functional in INMDB and shown in Table 27.

Table 27: SPARQL Pattern to Algebra/SQL/IQL – ORDER BY

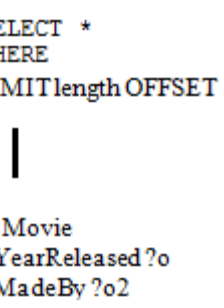
SPARQL Pattern	{ ?s a Movie . ?s :yearReleased ?o . ?s Sequel ?o2} ORDER BY ?o
SPARQL Algebra	ORDERBY(JOIN (BGP(?s :a Movie), JOIN (BGP(?s :yearReleased ?o), BGP(?s Sequel ?o2))), ?o Descending)
SQL Translation	SELECT subject, yearReleased, Sequel FROM Movie ORDER BY yearReleased desc
Tree	
IQL Translation	Query Movie* $s ["YearReleased":$o, Sequel:$o2] construct $s[$o,$o2 order by $o] ;

4.2.2.15 Pattern Involving OFFSET and LIMIT

The graph pattern shown in Table 28 is fully supported by INMDB. See "Table 6: SPARQL Query Features" for definition of OFFSET and LIMIT.

Table 28: SPARQL Pattern to Algebra/SQL/IQL – OFFSET and LIMT

SPARQL Pattern	{ ?s a Movie . ?s :yearReleased ?o . ?s Sequel ?o2} LIMIT length OFFSET start
SPARQL Algebra	SLICE(JOIN (BGP(?s :a Movie), JOIN (BGP(?s :yearReleased ?o), BGP(?s Sequel ?o2))), start, length)
SQL	SELECT subject, yearReleased, Sequel

Translation	FROM MOVIE LIMIT length OFFSET start
Tree	
IQL Translation	Query Movie* $s ["YearReleased":$o, Sequel:$o2] construct $s top(start, length) [$o,$2];

4.2.3 Summary of INMDB Support for SPARQL 1.1 queries

Table 29: Summary of Supported SPARQL Queries by INMDB

SPARQL query type	Supported by INMDB
Basic Graph Pattern	Supported
Basic Graph Pattern Belonging to a Class and it's Subclasses	Well Supported
Pattern with JOIN (subject-subject)	Well Supported
Pattern with JOIN (subject-object)	Well Supported
UNION of Two BGPs	Supported
Pattern with OPTIONAL JOIN	Supported
Pattern with more than One OPTIONAL JOIN	Supported
Pattern with a FILTER	Semi Supported
Pattern with UNION and OPTIONAL	Supported
Pattern Involving BIND	not Supported
Pattern Involving MINUS	Supported
Pattern Involving SUBQUERY	Supported
Pattern Involving GROUP BY with SUM and COUNT	Supported
Pattern Involving ORDER BY	Supported
Pattern Involving OFFSET and LIMIT	Supported

Table 30: Legend of Support Ratings

Level of Support	Definition
Well Supported	This rating means that INMDB is able to support the RDF query feature. In addition INMDB has features which enhance or facilitate this RDF query pattern's expression compared to the equivalent SQL.
Supported	This rating means that INMDB is able to support the RDF query feature.
Semi Supported	This rating means that INMDB is able to support only part of the RDF queryfeature.
not Supported	This rating means that INMDB does not have an operator to express the RDF query feature.

The results from the analysis indicate that INMDB is not fully SPARQL 1.1 compatible, however almost all graph patterns are supported as shown in Table 29. It can be stated that there is enough coverage of SPARQL features to support typical user

querying requirements. In order to be gain full compatibility operators for IQL must be incorporated into future versions of INMDB such as: BIND and FILTER.

INMDB's class/sub-class, subject-subject and subject-object query syntax features make expressing these RDF query graph patterns very simple. INMDB is specifically designed for querying classes and their sub-classes, elements of the class as well as relationships between classes (subject-object) and this design greatly facilitates RDF query expression compared to SQL.

Chapter 5

Storing DBPedia in INMDB

In this chapter the English version of the DBPedia data set is used to demonstrate how a real world RDF dataset can be stored in INMDB as well as the N-ary relational approach for MySQL. The chapter is outlined as follows. The tools and methods used for the preparation of the DBPedia dataset are described in section 5.1. The DBPedia dataset and how it was collected is described in section 5.2. The algorithms for generating the database schema and loading of the RDF data for both INMDB and MySQL are presented in section 5.3. Section 5.4 shows sample schema scripts as wells sample data insert scripts. Section 5.5 analyses various SPARQL queries on the DBPedia dataset stored in both INMDB and MySQL, offering a side-by-side comparison of each systems query language for querying RDF data. Finally, in section 5.6 a summary of the comparison between the two systems is discussed within the context of this database application, query performance results are presented and a summary of observed advantages and limitations of using INMDB are presented.

5.1 Tools

Several tools were used to complete the analysis done in this thesis. The database software used are shown in Table 31, the software tools used are shown in Table 32 and the computer system specifications used for implementation are shown in Table 33.

Table 31: Databases Used

Software	Source
MySQL version 5.7	https://www.mysql.com/
INMDB version 1.0	https://drive.google.com/file/d/0B4T7b5MI2y50bTVsQ20xUzhnYzg/view?usp=drive_web Provided by Mengchi Liu from Carleton University

MySQL was installed on a solid state drive on the main testing computer. INMDB came pre-installed on an Ubuntu virtual machine, which was loaded onto the same solid state drive as MySQL.

Table 32: Software Tools Used

Software	Source
gVim version 7.4	http://www.vim.org/download.php
Microsoft Excel 2010	https://en.wikipedia.org/wiki/Microsoft_Office_2010
Microsoft Notepad	https://en.wikipedia.org/wiki/Microsoft_Notepad
Processing 3.2.3	https://processing.org/download/?processing
Oracle VM VirtualBox 5.0	https://www.virtualbox.org/

Several software tools were used for testing, transforming, cleaning and manipulating the data. Microsoft notepad and Microsoft Excel have a limit on the file size that can be opened with them. The initial raw data that was downloaded was approximately 2.6 GB, which is much too large to be edited using either Microsoft Notepad or Microsoft Excel. gVim is a text editor that can handle very large files, so this was used to edit and view the very large dataset. VirtualBox was used to run the Linux virtual machine that INMDB came installed on. Processing 3.2.3 was used for cleaning

the data as well as building the insert scripts and schemas for both MySQL and INMDB, the algorithms for which will be discussed in detail in the section “Loading DBPedia Data into INMDB and ”.

Table 33: System Specifications

System Component	Description
Operating System	Windows 10 64-bit
Installed memory (RAM)	32.0 GB
Processor	Intel Core i5-2500k CPU @ 3.30GHz (Overclocked to 3.5GHz)
Storage	OCZ Agility 3 120GB SATA III Solid State Drive (SSD)

The virtual machine INMDB was installed on had a limit of 20GB of ram, whereas MySQL had a limit of 32GB on the host Windows 10 operating system. The program used to convert the RDF triples into insert scripts for MySQL and INMDB required 10GB of system memory.

5.2 Data Collection and Data Set

DBpedia is a crowd sourced effort to take information from the website Wikipedia and turn it into structured RDF format data so that machines and users can more easily query Wikipedia data. Its goal is to make it easier for people to access the huge amount of information available on Wikipedia in order for it to be used in new and interesting ways. The DBPedia dataset was chosen because it is easily accessible, open

source, has many different RDFS/OWL classes and is overall a very good candidate dataset to be used for investigating INMDB's ability for storing RDF datasets.

In this thesis the English version of DBpedia's April 2016 dataset is selected. It describes roughly 5 million real world things, out of which 4.85 million are classified in a consistent ontology. This includes:

- 1.4 million persons
- 735 thousand places
- 411 thousand creative works
- 241 thousand organizations
- 251 thousand species
- 6000 diseases

The DBPedia dataset used was collected from the DBpedia download page[23], totaling 4 downloaded files:

2. RDF schema containing 30,318 triples[24]
3. Instance types containing 4.85 million triples[25]
4. Literal mappings containing 16.9 million triples[26]
5. Object mappings containing 18.2 million triples[27]

[23] Download Page - http://wiki.dbpedia.org/dbpedia-version-2016-04
(retrieved Aug 01, 2017)
[24] Schema - downloads.dbpedia.org/2016-04/dbpedia_2016-04.nt
(retrieved Aug 01, 2017)
[25] Instance Types - downloads.dbpedia.org/2016-04/core-i18n/en/instance_types_en.ttl.bz2
(retrieved Aug 01, 2017)
[26] Literals - downloads.dbpedia.org/2016-04/core-i18n/en/mappingbased_literals_en.ttl.bz2
(retrieved Aug 01, 2017)
[27] Object Mappings - downloads.dbpedia.org/2016-04/core-i18n/en/mappingbased_objects_en.ttl.bz2
(retrieved Aug 01, 2017)

The schema file contains an RDFS/OWL schema for the DBPedia dataset. The instance types file contains triples which assign subjects to their respective classes. The literal mappings file contains triples which assign subject properties a literal value (e.g., Strings and integers). The object mappings file contains triples which link objects to other subjects (e.g., <ex.ca/r/Wall-E> <ex.ca/p/Sequel> <ex.ca/r/Wall-E 2>).

The entire dataset is composed of:

- Approximately 40 million triples.
- 740 different classes.
- 2694 different properties.
    - 1622 of which are literals.
    - 1072 of which are object-mappings.

These files needed to be cleaned using a custom program. Cleaning involved converting the triple files into comma separated value (csv) files so that they could be more easily processed. Cleaning also involved removing problem characters such as quotation marks and triangle brackets. Once this data was cleaned, each file was split into several sub files; the instance types file was split into 20 sub files, literal mapping file was split into 68 sub files and object mapping file was split into 74 sub files. The data files were split in order to help with memory management for the loading program which was used to generate insert scripts for MySQL and INMDB. It was found that loading the entire file was using too much system memory, so the sub files were used which made system memory use more manageable.

It is important to note that the extraction process by which DBPedia extracts RDF data from Wikipedia can affect the quality of the data that is being inserted into INMDB.

A case where this could arise for example is if the data provided by DBPedia is logically inconsistent. For instance, this can occur if a triple node is defined as both a cat and a dog. The problem that arises from this is that a user querying INMDB will receive a logically inconsistent result.

From an analysis on the dataset, very few contradictions where found in the DBPedia dataset. One example of which however were that some breweries had the same subject URI as abbys. This would be a logical inconsistency in RDF and it stems from improper labeling from the extraction process (Wikipedia to DBPedia). Ideally tags would be added to the subject URI to specify that one is a brewery (a company) and one is an abby (place). This exceptions were handled in INMDB by enforcing unique subject URIs.

5.3 Loading DBPedia Data into INMDB and MySQL

In this section the algorithms used to load the DBPedia data set into the INMDB approach and MySQL N-ary approach are described. In order to load the RDF data into MySQL and INMDB two algorithms where required in order to generate database insert scripts. The algorithm used for INMDB is described in section 5.3.1 and the algorithm used for MySQL is described in section 5.3.1.1. With each algorithm the triples are parsed, logically represented in memory and then reconstructed into insert scripts which could then be loaded into the databases.

The main difference between these two algorithms is the way in which multi-valued attributes are handled. For MySQL, multi-valued RDF properties need to be normalized (they need to have their own table) and without foreknowledge of whether a

property is multi-valued or not, we first need to load all data in order to check whether the property is multi-valued, and then build the MySQL database schema accordingly. It is impossible to build the MySQL schema without knowing if properties are multi-valued. For INMDB however, the problem is very straight forward. There is no need to check if properties are multi-valued, they are all simply stored as multi-valued properties. This permits the generation of the IDL schema regardless of whether data is multi-valued or not.

In general, for the relational N-ary approach, not knowing if properties are multi-valued is considered to be a serious difficulty when processing RDF data. This is because it requires the database schema to change when new multi-valued properties arise; in INMDB this is not a problem.

Once the insert scripts were created, they were executed and the data was loaded into each of the databases. For INMDB, the insert script was too large for the INMDB client to handle, so it was split into 46 subfiles, each of which were executed in the INMDB client using the multi-file execute command “import”.

MySQL required 3.6GB to store the entire DBPedia dataset where as INMDB required 41.6GB to store the entire DBPedia dataset. Approximately 10x storage in INMDB. The reason INMDB takes so much more room is because it generates indexes for every object, attribute and relationship (including inverse relationships).

5.3.1 INMDB Loader Pseudo Code

Given the properly formatted input, this algorithm is able to generate an equivalent INMDB database (schema generation and data insert). This algorithm will

work for any RDFS dataset, as well as DBPedia (which has some OWL). Logical inconsistencies in the class structure (circular inheritance), duplicate classes and duplicate class assignment are handled by INMDB.

As input to this algorithm:

- RDFS schema in csv format
- RDFS object instance definitions in csv format
- RDFS literals definitions in csv format
- RDFS object-mapping defintions in csv format.

For INMDB, the following procedure was used to generate the database schema script and data insert script:

1. Parse the RDF schema and convert into a logical schema.
2. Convert the logical schema into an INMDB schema format and output to IDL script file.
3. Create logical tables for each RDF class with its domain properties as columns.
4. Load the instance types for subjects into these tables.
5. For each literal and object-mapping associated with a subject, insert it into appropriate table.
6. Convert the logical tables representing the RDF data into IML insert scripts and output to file.

The first step here was to parse the RDF schema for DBPedia. Because the schema was well structured (i.e., Did not have circular inheritance, duplicate triples or other unexpected anomalies) this was very straight forward. Loading the RDF literal and

object data into logical files is the next part. The pseudo code for creating schema script and the data insert script for INMDB is as follows:

INMDB Loader Pseudo Code

```
//SET UP LOGICAL CLASS AND PROPERTY STRUCTURES
//set up classes
For Each triple in RDF Schema
  If triple is a class definition and we have not already created this logical class{
    Create a logical class}

For Each triple in RDF Schema
  If triple is an equivalent class statement{
     Record for our logical class that it has an equivalent class}

For Each triple in RDF Schema
  If triple is a subClassOf definition{
       Record for our logical class that it is a subClassOf}

While still adding for each logical class{
       Record a list of all classes which this class is a subclass of}

For Each triple in RDF Schema
  If triple is a subClassOf definition{
       Record for our logical class that it is a superclass of another class}

While still adding for each logical class{
       Record a list of all classes which this class is a superclass of}

//set up properties
For Each triple in RDF Schema
  If triple is a Property definition and we have not already created this logical property{
       Create a logical property
       For each triple in Schema
            If triple is an equivalent property definition{
                Record for this logical property that it has an equivalent class}
   }

For Each triple in RDF Schema
  If triple is a range definition{
        Add this range to its corresponding logical property }

//Note: list of own properties are the properties that are assigned to the class and  not those
that are inherited.
//Note2: list of extended properties are the properties that a class has been assigned AND
inherits from subClassOf

For Each triple in RDF Schema
```

```
  If triple is a domain definition{
       Add the logical property to the corresponding logical class' list of own properties}

For Each logical class
    Add own Properties to list of extended properties.

//Note3: this creates a list of all the properties a class has including inherited properties.
While you can still add classes to list of superclass for each logical class{
    For each of its superclasses {
        add the superclass' extended properties to its extended properties}}

For Each property
  If the range does not refer to a class or equivalent class in the list of logical classes{
    Set the range to String }

For Each class{
  For Each of the class' properties{
    If the range of the property points to this class or sub or super class of this class{
      Set the type of the property to 'role' }
     Else{
      Set the type of the property to 'normal'}}}


Output INMDB IDL schema script

//CREATE LOGICAL TABLES AND LOAD DATA INTO THESE TABLES
//create logical tables
For Each Logical Class{
  Create a logical class data table with a column for each logical property
  Create binary tree index for subject column of this table}

//load instance types
For Each triple in the instance types file{
   If the triple is an instance type definition{
      For each logical class{
        If class is defined by triple or equivalent class{
          Add subject to logical class table
          Add subject to corresponding binary tree with row number for the subject}
   }}}

//load literals
For each triple in the literal mappings file{
    For each logical class{
       If the logical class table contains the triple's subject (using binary tree to check){
         If the corresponding field for the subject row is empty{
           Add literal's object value to the field}
         Else{
           Concatenate and add literal's object value to field}
       }}}
```

```
//load objects
For each triple in the object mappings file{
    For each logical class{
        If the logical class table contains the triple's subject (using binary tree to check){
          If the class is the range {
            If the corresponding field for the subject row is empty{
               Add target class and object value to corresponding logical property table}
             Else{
               Concatenate and  target class name add object value to field}
        }}}}
```

Generate INMDB IML data insert script file

5.3.1.1 INMDB Loader Time Complexity

Inputs:

- N_{schema} – the number of triples in the schema
- C – the number of classes
- P- the number of predicates
- $N_{instances}$ – number of triple instance definitions
- $N_{literals}$ – number of literal triples
- $N_{objects}$ – number of object-mapping triples
- N – total number of triples in the dataset.

The time complexity for the INMDB Loader algorithm is as follows:

- Setting up logical class and property structures
    - $O(C * N_{schema})$
- Creating empty logical tables
    - O(C * P)

- Loading instance type csv[28]
    - $O(C * N_{instances}^2)$
- Load literals csv
    - $O(C * N_{literals} * \log(N_{literals}))$
- Load object mappings
    - $O(C * N_{objects} * \log(N_{objects}) * C * \log(N_{instances}))$
- Output IML
    - $O(C)$
- Output IDL
    - $O(N_{instances})$

Total time complexity is:

$$O(C *(N_{schema} + P + N_{instances}^2 + N_{literals} * \log(N_{literals}) + N_{objects} * \log(N_{objects}) * C * \log(N_{instances})) + C + N_{instances})$$

Simplified to:

$$O(N^3)$$

[28] Note: This is N^2 because the current binary tree implementation does not allow bulk loading. With bulk loading this complexity would become Nlog(N)

5.3.2 MySQL Loader Pseudo Code

For MySQL a similar approach is taken, however, the main difference is that the database schema is output last and after loading the literal and object data into logical tables we normalize those that are found to be multi-valued.

As input to this algorithm:

- RDFS schema in csv format
- RDFS object instance definitions in csv format
- RDFS literals definitions in csv format
- RDFS object-mapping defintions in csv format.

For MySQL, the following procedure was used to generate the database schema script and data insert script:

1. Parse the RDF schema and convert into a logical data structure.
2. Create logical N-ary tables for each RDF class with it's domain properties as columns.
3. Load the instance types for subjects into these tables.
4. For each literal and object associated with a subject, insert it into appropriate table. When a multi-valued insert is encountered, concatenate with a separator to existing data and record that property is multi-valued.
5. For each logical class datatable, for each property, if the property is multi-valued create a property table for this property (if not already created) and transfer tokenized values from the class table to the property table.
6. Create MySQL table schema for class tables and property tables and output to file.

7. Convert the logical tables representing the RDF data into data insert scripts and output to file.

The pseudo code for creating schema and csv data files for MySQL is as follows:

MYSQL Loader Pseudo Code

```
//SET UP LOGICAL CLASS AND PROPERTY STRUCTURES
//set up classes
For Each triple in RDF Schema
  If triple is a class definition and we have not already create this logical class{
    Create a logical class}

For Each triple in RDF Schema
  If triple is an equivalent class statement{
     Record for our logical class that it has an equivalent class}

For Each triple in RDF Schema
  If triple is a subClassOf definition{
       Record for our logical class that it is a subClassOf}

For Each triple in RDF Schema
  If triple is a subClassOf definition{
       Record for our logical class that it is a superclass of another class}

While still adding for each logical class
       Record a list of all classes which this class is a superclass of

//set up properties
For Each triple in RDF Schema
  If triple is a Property definition and we have not already created this logical property{
       Create a logical property
       For each triple in Schema
            If triple is an equivalent property definition{
                Record for this logical property that it has an equivalent class}
  }

For Each triple in RDF Schema
  If triple is a range definition{
        Add this range to its corresponding logical property }

//Note: list of own properties are the properties that are assigned to the class and not those
that are inherited.
//Note2: list of extended properties are the properties that a class has been assigned AND
inherits from subClassOf

For Each triple in RDF Schema
   If triple is a domain definition{
```

```
        Add the logical property to the corresponding logical class' list of own properties}

For Each logical class
    Add own Properties to list of extended properties.

//Note3: this creates a list of all the properties a class has including inherited properties.

While you can still add classes to list of superclass for each logical class
    For each of its superclasses, add the superclass' extended properties to its extended
properties

For Each property
  If the range does not refer to a class or equivalent class{
    Set the range to String }


//CREATE LOGICAL TABLES AND LOAD WITH DATA
//create logical tables
For Each Logical Class{
  Create a logical class data table
  Create binary tree index for subject column of this table}

//load instance types
For Each triple in the instance types file{
   If the triple is an instance type definition{
      For each logical class
        If class is defined by triple or equivalent class{
          Add subject to logical class table
          Add subject to corresponding binary tree with row number for the subject}
   }}

//load literals
For each triple in the literal mappings file{
    For each logical class{
       If the logical class table contains the triple's subject (using binary tree to check){
         If the corresponding field for the subject row is empty{
            Add literal's object value to the field}
          Else{
            Concatenate and add literal's object value to field with a separator
            Record that property is multi-valued}
       }}}

//load objects
For each triple in the object mappings file{
    For each logical class{
       If the logical class table contains the triple's subject (using binary tree to check){
         If the corresponding field for the subject row is empty{
            Add object mapping's object value to the field
          Else{
            Concatenate and add Object mapping's object value to field with a separator
```

```
            Record that property is multi-valued}
        }}}

For Each Class
  For each Property of this Class
    If the property is multi-valued and we have not already created a property table for it{
      Create a logical property table}

For Each Class
   For each Property of this Class
     If the property is multi-valued {
       For each field in this class table for this property {
          Tokenize each value for this field based on the seperator
          Insert each token with subject into corresponding property table
}}
```

Generate MySQL database schema script.

Save class and multi-valued-property tables as .csv files

Generate script that loads each .csv file into MySQL.

5.3.2.1 MySQL Loader Time Complexity

Inputs:

- N_{schema} – the number of triples in the schema
- C – the number of classes
- P- the number of predicates
- $N_{instances}$ – number of triple instance definitions
- $N_{literals}$ – number of literal triples
- $N_{objects}$ – number of object-mapping triples
- N – total number of triples in the dataset.

The time complexity for the MySQL Loader algorithm is as follows:

- Setting up logical class and property structures

- $O(C * N_{schema})$

- Creating empty logical tables
  - $O(C * P)$
- Loading instance type csv[29]
  - $O(C * N_{instances}^2)$
- Load literals csv
  - $O(C * N_{literals} * \log(N_{literals}))$
- Load object mappings
  - $O(C * N_{objects} * \log(N_{objects}))$
- Split multivalued properties into their own tables
  - $O(N_{literals} + N_{objects})$
- Output Schema
  - $O(C + P)$
- Output csv loader script
  - $O(C + P)$
- Output csv tables
  - $O(N_{instances})$

Total time complexity is:

$$O(C *(N_{schema} + P + N_{instances}^2 + N_{literals} * \log(N_{literals}) + N_{objects} * \log(N_{objects}) * C * \log(N_{instances}))$$

[29] Note: This is N^2 because the current binary tree implementation does not allow bulk loading. With bulk loading this complexity would become Nlog(N)

$$+ N_{literals} + N_{objects} + 2*(C + P) + N_{instances})$$

Simplified to:

$$O(N^3)$$

5.4 Sample Schemas and Insert Statements

5.4.1 INMDB Samples

Sample schema creation IDL statements and IML data loading statements are shown below for INM.

Sample INMDB IDL Class Creation Statement For Artist

```
// Class Name: Artist
// Description: Contains all properties for the DBPedia class “Artist”
// Sub-classes: {"Dancer","Painter","Comedian","ComicsCreator","MusicalArtist",
// "FashionDesigner","Actor","Sculptor","Humorist","Photographer"}
// Super Class: {Person}


create class "Artist" SUBSUME
{"Dancer","Painter","Comedian","ComicsCreator","MusicalArtist","FashionDesigner","Actor","
Sculptor","Humorist","Photographer"} [
  normal "mentor"(N:N):"Artist",
  @"voiceType"*:string,
  @"field"*:string,
  normal "disciple"(N:N):"Artist",
  role "tonyAward"(N:N):"Award",
  @"style"*:string,
  role "cesarAward"(N:N):"Award",
  role "polishFilmAward"(N:N):"Award",
  role "training"(N:N):"EducationalInstitution",
  role "gaudiAward"(N:N):"Award",
  role "emmyAward"(N:N):"Award",
  normal "associatedAct"(N:N):"Artist",
  role "filmFareAward"(N:N):"Award",
  role "goldenGlobeAward"(N:N):"Award",
  role "goyaAward"(N:N):"Award",
  role "baftaAward"(N:N):"Award",
  role "grammyAward"(N:N):"Award",
  @"movement"*:string,
```

```
  role "academyAward"(N:N):"Award",
  role "instrument"(N:N):"Instrument",
  role "afiAward"(N:N):"Award",
  @"dutchRKDCode"*:string,
];
```

Sample INMDB IML Object Instance Creation Statement For Artist

```
// Class of object being inserted: Artist
// Description: Inserts an instance of the a DBPedia Artist named "Andy Warhol"
// Subject of RDF data: "<http://dbpedia.org/resource/Andy_Warhol>"

insert "Artist" "<http://dbpedia.org/resource/Andy_Warhol>" [
  @"field":{"<http://dbpedia.org/resource/Printmaking>"},
  role "training":{"University" "<http://dbpedia.org/resource/Carnegie_Mellon_University>"},
  @"movement":{"<http://dbpedia.org/resource/Pop_art>"},
  @"birthName":{"+Andrew Warhola+@en ."},
  @"deathDate":{"+1987-02-22+^^<http://www.w3.org/2001/XMLSchema#date>"},
  @"birthDate":{"+1928-08-06+^^<http://www.w3.org/2001/XMLSchema#date>"},
  role "birthPlace":{"City" "<http://dbpedia.org/resource/Pittsburgh>"},
];
```

5.4.2 MySQL Samples

Sample schema table creation statements and CSV file loading script are shown below for MySQL.

Sample MySQL Schema Table Creation Statement For Class Table Agent

```
// Table Name: CT_Agent
// Description: Contains all single-valued attributes for the DBPedia class "Agent".[30]
// Literal Datatype: Text[31]
// Object Mapping Datatype: Text[31]
// number of indexes: 1 (on Subject Attribute)

DROP TABLE IF EXISTS `CT_Agent`;
CREATE TABLE `CT_Agent` (
   `Subject` VARCHAR(256),
   INDEX `484_subject` (`Subject`),
   `generalCouncil` TEXT,
```

[30] Note: Multi-valued attributes are stored in property tables (Tables with PT prefix, see PT_illustrator)
[31] Note: Could not use VARCHAR as datatype because it exceeded maximum table size.

```
   `playerSeason` TEXT,
   `owns` TEXT,
   `season` TEXT,
   `age` TEXT,
   `roleInEvent` TEXT,
   `nationalSelection` TEXT,
   `juniorSeason` TEXT,
   `regionalCouncil` TEXT,
   `artPatron` TEXT,
   `managerSeason` TEXT
) ENGINE = MYISAM;
```

Sample MySQL Schema Table Creation Statement For Property Table Illustrator

```
//Table Name: PT_illsutrator
// Description: Stores data for the multi-valued DBPedia Property "illustrator"
// number of attributes: 2 (subject and object)
// number of indexes: 2 (1 for subject, 1 for subject and object combined)

DROP TABLE IF EXISTS `PT_illustrator`;
CREATE TABLE `PT_illustrator` (
   `Subject` VARCHAR(256),
   INDEX `741_subject` (`Subject`),
   `Object` VARCHAR(256),
   INDEX `742_Subject_Object` (`Subject`(50),`Object`)
) ENGINE = MYISAM;
```

Sample MySQL Data Loading Script

```
//Description: This script loads the csv file of DBPedia RDF data into the "CT_Agent" table.

LOAD DATA LOCAL INFILE 'C:/Users/Patrick/ MYSQL/CT_Agent.csv'
INTO TABLE `CT_Agent`
FIELDS TERMINATED BY ',' LINES TERMINATED BY '\r\n' IGNORE 1 LINES
(`Subject`, `generalCouncil`, @dummy, `playerSeason`, @dummy, `owns`, `season`, `age`,
`roleInEvent`, `nationalSelection`, `juniorSeason`, `regionalCouncil`, @dummy, `artPatron`,
`managerSeason`);
```

5.5 Sample Queries on DBPedia

In this section some sample RDF queries are executed on both INMDB and MySQL. The queries are based of those in section 4.2.1. By showing IQL and SQL side-by-side, the simplicity of IQL's query language for querying RDF is shown.

Each query is performed on the dataset of each respective database (MySQL and INMDB). Each database has its own interface for running the queries. MySQL uses a graphical user interface called the MySQL Workbench, but also accepts queries using a PowerShell script. INM uses a terminal console to run queries. INM only returns "real" time results, which includes time fetch results of a query together with time to output the text to console. In order to have a fair comparison of timing, the command "Measure-Command { }" which returns a detailed report of how long it takes for a PowerShell command to execute and output results to console was used to get the "real" time for MySQL. The timing results of each query are shown in section 5.6.

Some queries exceeded memory for both MySQL and INMDB due to the size of the data being retrieved (e.g., query all persons and their subclasses). In general, large queries such as these were avoided because of the limitations of the host computer system. The amount of memory used during querying was not recorded as this workload feature was not available for both systems.

Results of the queries where as expected for both MySQL and INMDB, however both systems output results in different formats. For each query, a sample output is given in the format that an RDFS storage system application would provide (tabular format).

5.5.1 Basic Graph Pattern Belonging to a Class and it's Subclasses

Description

Select all politicians and sub-class of politician.

SPARQL Pattern

```
SELECT ?s
WHERE
```

```
{
  ?s a <http://dbpedia.org/ontology/Politician>
}
```

SQL Translation

```
SELECT subject FROM CT_President
UNION
SELECT subject FROM CT_PrimeMinister
UNION
SELECT subject FROM CT_Deputy
UNION
SELECT subject FROM CT_Congressman
UNION
SELECT subject FROM CT_Senator
UNION
SELECT subject FROM CT_VicePresident
UNION
SELECT subject FROM CT_VicePrimeMinister
UNION
SELECT subject FROM CT_Mayor
UNION
SELECT subject FROM CT_Lieutenant
UNION
SELECT subject FROM CT_MemberOfParliament
UNION
SELECT subject FROM CT_Governor
UNION
SELECT subject FROM CT_Chancellor
UNION
SELECT subject FROM CT_Politician
```

IQL Translation

```
Query Politician* $s
construct $s;
```

Sample Results

s
http://dbpedia.org/resource/Axel_Oxenstierna
http://dbpedia.org/resource/Carlos_Menem
http://dbpedia.org/resource/Carter_Harrison_Jr.
http://dbpedia.org/resource/Carter_Harrison_Sr.

http://dbpedia.org/resource/Chen_Shui-bian
http://dbpedia.org/resource/David_Lloyd_George
http://dbpedia.org/resource/Enrico_Berlinguer
http://dbpedia.org/resource/Eric_Williams
http://dbpedia.org/resource/Erich_Honecker
http://dbpedia.org/resource/Eva_Perón

5.5.2 Pattern with JOIN (subject-subject)

Description

Query Animal class and get their kingdom and family attribute.

SPARQL Pattern

```
SELECT ?s, ?k, $f
WHERE
{
  ?s rdf:type <http://dbpedia.org/ontology/Animal>.
  ?s <http://dbpedia.org/ontology/kingdom> ?k .
  ?s <http://dbpedia.org/ontology/family> ?f .
}
```

SQL Translation

```
SELECT t1.subject, t2.object, t3.object
FROM ((CT_Animal as t1 JOIN PT_family as t2 ON t1.subject = t2.subject)
        JOIN PT_kingdom as t3 ON t1.subject = t3.subject)
```

IQL Translation

```
Query Animal $s[family:$f, kingdom:$k]
construct $s[$k , $f];
```

Sample Results

s	k	f
http://dbpedia.org/resource/African_clawed_frog	http://dbpedia.org/resource/Animal	http://dbpedia.org/resource/Pipidae
http://dbpedia.org/resource/American_black_bear	http://dbpedia.org/resource/Animal	http://dbpedia.org/resource/Bear

http://dbpedia.org/resource/American_pickerel	http://dbpedia.org/resource/Animal	http://dbpedia.org/resource/Esocidae

5.5.3 Pattern with JOIN (subject-object)

Description:

Find governors, their nationality and the dissolution date of the country.

SPARQL Pattern

```
SELECT ?s, ?n, ?d
WHERE
{
  ?s rdf:type <http://dbpedia.org/ontology/Governor> .
  ?s <http://dbpedia.org/ontology/nationality> ?n .
  ?n <http://dbpedia.org/ontology/dissolutionDate> ?d .
}
```

SQL Translation

```
SELECT t1.subject, t2.Object, t4.object
FROM (((CT_Governor As t1
        JOIN pt_nationality as t2 ON t1.subject = t2.subject)
        JOIN ct_country as t3 ON t2.Object  = t3.Subject)
        JOIN PT_DissolutionDate as t4 ON t4.subject = t3.subject)
```

IQL Translation

```
Query Governor $s [nationality:$n [dissolutionDate:$d]]
construct $s[$n[$d]];
```

Sample Results

s	n	d
http://dbpedia.org/resource/Richard_Clement_Moody	http://dbpedia.org/resource/United_Kingdom_of_Great_Britain_and_Ireland	1922-12-06
http://dbpedia.org/resource/Carel_Reyniersz	http://dbpedia.org/resource/Dutch_Republic	1795-01-18
http://dbpedia.org/resource/Thomas_Hinckley	http://dbpedia.org/resource/Kingdom_of_England	1707-05-01

http://dbpedia.org/resource/Anthony_van_Diemen	http://dbpedia.org/resource/Dutch_Republic	1795-01-18

5.5.4 UNION of Two BGPs

Description

Find governors and their nationality Union with Mayors and their nationality.

SPARQL Pattern

```
SELECT ?s, ?n
WHERE
{
{ ?s rdf:type <http://dbpedia.org/ontology/Governor> .
  ?s <http://dbpedia.org/ontology/nationality> ?n .
}
UNION
{ ?s rdf:type <http://dbpedia.org/ontology/Mayor> .
  ?s <http://dbpedia.org/ontology/nationality> ?n .  }
}
```

SQL Translation

```
SELECT t1.subject, t2.object
FROM (CT_Governor As t1 Join pt_nationality as t2 ON t1.subject = t2.subject)
UNION
SELECT t1.subject, t2.object
FROM (CT_Mayor As t1 Join pt_nationality as t2 ON t1.subject = t2.subject);
```

IQL Translation

```
Query Governor $s[nationality:$n], Mayor $ss[nationality:$nn]
construct $s[$n], $ss[$nn];
```

Sample Results:

s	n
http://dbpedia.org/resource/Henry_B._Quinby	http://dbpedia.org/resource/United_States
http://dbpedia.org/resource/Herbert_W._Ladd	http://dbpedia.org/resource/United_States

http://dbpedia.org/resource/Manuel_Uribe_Ángel	http://dbpedia.org/resource/Colombian_people
http://dbpedia.org/resource/Robert_Duff_(politician,_born_1835)	http://dbpedia.org/resource/British_people
http://dbpedia.org/resource/Tom_Pauling	http://dbpedia.org/resource/Australia
http://dbpedia.org/resource/Göran_Tunhammar	http://dbpedia.org/resource/Sweden
http://dbpedia.org/resource/Maria_Norrfalk	http://dbpedia.org/resource/Sweden

5.5.5 Pattern with OPTIONAL JOIN

Description

Find governors and their nationality, and if it exists, the dissolution date of their country.

SPARQL Pattern

```
SELECT ?s, ?n, ?d
WHERE
{
  ?s rdf:type <http://dbpedia.org/ontology/Governor> .
  ?s <http://dbpedia.org/ontology/nationality> ?n .
  OPTIONAL {
        ?n rdf:type <http://dbpedia.org/ontology/Country> .
        ?n <http://dbpedia.org/ontology/dissolutionDate> ?d .
        }
}
```

SQL Translation

```
SELECT t1.subject, t2.Object, t4.object
FROM (((CT_Governor As t1
        JOIN pt_nationality as t2 ON t1.subject = t2.subject)
        LEFT JOIN ct_country as t3 ON t2.Object  = t3.Subject)
        JOIN PT_DissolutionDate as t4 ON t4.subject = t3.subject)
```

IQL Translation

```
Query Governor $s [nationality:$n [dissolutionDate:$d | !dissolutionDate is null]]
construct $s[$n[$d]];
```

Sample Results

s	n	d
http://dbpedia.org/resource/Richard_Clement_Moody	http://dbpedia.org/resource/United_Kingdom_of_Great_Britain_and_Ireland	1922-12-06
http://dbpedia.org/resource/Henry_B._Quinby	http://dbpedia.org/resource/United_States	
http://dbpedia.org/resource/Herbert_W._Ladd	http://dbpedia.org/resource/United_States	
http://dbpedia.org/resource/Manuel_Uribe_Ángel	http://dbpedia.org/resource/Colombian_people	
http://dbpedia.org/resource/Robert_Duff_(politician,_born_1835)	http://dbpedia.org/resource/British_people	

5.5.6 Pattern with more than One OPTIONAL JOIN

Description

Find governors who optionally have a nationality and who optionally have a spouse.

SPARQL Pattern

```
SELECT ?s, ?n, ?d
WHERE
{
  ?s rdf:type <http://dbpedia.org/ontology/Governor> .
  OPTIONAL { ?s <http://dbpedia.org/ontology/nationality> ?n . }
  OPTIONAL {?n <http://dbpedia.org/ontology/spouse> ?d .}
}
```

SQL Translation

```
SELECT t1.subject, t2.Object, t3.object
FROM (CT_Governor As t1 LEFT JOIN pt_nationality as t2
        ON t1.subject = t2.subject)
LEFT JOIN pt_spouse as t3 ON t1.Subject  = t3.Subject;
```

IQL Translation

```
Query
Governor $s [  !nationality is null, spouse:$xd |
               nationality:$xn, !spouse is null |
               nationality:$n, spouse:$d |
               !nationality is null, !spouse is null]
```

construct $s[$n,$d,$xd,$xn];

Sample Results

s	n	d
http://dbpedia.org/resource/Henry_B._Quinby	http://dbpedia.org/resource/United_States	
http://dbpedia.org/resource/Herbert_W._Ladd	http://dbpedia.org/resource/United_States	
http://dbpedia.org/resource/Francisco_Pizarro	http://dbpedia.org/resource/Eliza_Jumel	http://dbpedia.org/resource/Aaron_Burr

5.5.7 Pattern with a FILTER

Description

Find governors, their nationality, dissolution date of their nationality which was in the year 1922.

SPARQL Pattern

```
SELECT ?s, ?n, ?d
WHERE
{
  ?s rdf:type <http://dbpedia.org/ontology/Governor> .
  ?s <http://dbpedia.org/ontology/nationality> ?n .
  ?n <http://dbpedia.org/ontology/dissolutionDate> ?d
  FILTER regex (?d , “1922”)
}
```

SQL Translation

```
SELECT t1.subject, t2.Object, t4.object
FROM (((CT_Governor As t1
        JOIN pt_nationality as t2 ON t1.subject = t2.subject)
        JOIN ct_country as t3 ON t2.Object  = t3.Subject)
        JOIN PT_DissolutionDate as t4 ON t4.subject = t3.subject)
WHERE t4.object like "%1922%";
```

IQL Translation

Query Governor $s [nationality:$n [dissolutionDate:$d = %”1922”%]]
construct $s[$n[$d]];

Sample Results

s	n	d
http://dbpedia.org/resource/Richard_Clement_Moody	http://dbpedia.org/resource/United_Kingdom_of_Great_Britain_and_Ireland	1922-12-06

5.5.8 Pattern with UNION and OPTIONAL

Description

Find governors and optionally their nationality unioned with mayors and optionally their nationality.

SPARQL Pattern

```
SELECT ?s, ?n
WHERE
{
  {
  ?s rdf:type <http://dbpedia.org/ontology/Governor> .
  OPTIONAL { ?s <http://dbpedia.org/ontology/nationality> ?n . }
  }
UNION
  {
  ?s rdf:type <http://dbpedia.org/ontology/Mayor> .
  OPTIONAL { ?s <http://dbpedia.org/ontology/nationality> ?n . }
  }
}
```

SQL Translation

```
SELECT t1.subject, t2.Object
FROM (CT_Governor As t1 LEFT JOIN pt_nationality as t2
  ON t1.subject = t2.subject)
UNION
SELECT t1.subject, t2.Object
FROM (CT_Mayor As t1 LEFT JOIN pt_nationality as t2
  ON t1.subject = t2.subject)
```

IQL Translation

```
Query
Governor $s [!nationality is null |
        nationality:$n [!dissolutionDate is null |dissolutionDate:$d ]],
Mayor $ss [!nationality is null |
        nationality:$nn [!dissolutionDate is null | dissolutionDate:$dd]]
construct $s[$n[$d]], $ss[$nn[$dd]];
```

Sample Results

s	n
http://dbpedia.org/resource/Henry_B._Quinby	http://dbpedia.org/resource/United_States
http://dbpedia.org/resource/Herbert_W._Ladd	http://dbpedia.org/resource/United_States
http://dbpedia.org/resource/Manuel_Uribe_Ángel	http://dbpedia.org/resource/Colombian_people
http://dbpedia.org/resource/Robert_Duff_(politician,_born_1835)	http://dbpedia.org/resource/British_people
http://dbpedia.org/resource/Tom_Pauling	http://dbpedia.org/resource/Australia
http://dbpedia.org/resource/Göran_Tunhammar	http://dbpedia.org/resource/Sweden
http://dbpedia.org/resource/Maria_Norrfalk	http://dbpedia.org/resource/Sweden

5.5.9 Pattern Involving MINUS

Description

Select mammals that are not Dogs.

SPARQL Pattern

```
SELECT ?mammal
WHERE
{
  { ?mammal a <http://dbpedia.org/ontology/Mammal> .}
   MINUS
  {?mammal rdf:type <http://dbpedia.org/ontology/Dog> .
}
```

SQL Translation

***NOTE: Minus operator is not supported in MySQL. Below is a query that gives equivalent results**

```
SELECT subject
FROM CT_Mammal
UNION
SELECT subject
FROM CT_Cat
UNION
SELECT subject
FROM CT_Horse
UNION
SELECT subject
FROM CT_RaceHorse
```

IQL Translation

```
Query Mammal* $m, not Dog* $m
construct $m;
```

Sample Results

mammal
http://dbpedia.org/resource/American_black_bear
http://dbpedia.org/resource/Beefalo
http://dbpedia.org/resource/Black_rat
http://dbpedia.org/resource/Brown_rat
http://dbpedia.org/resource/Donkey
http://dbpedia.org/resource/Even-toed_ungulate
http://dbpedia.org/resource/Galago
http://dbpedia.org/resource/Hare
http://dbpedia.org/resource/Lagomorpha

5.5.10 Pattern Involving SUBQUERY

Description

Select countries and their dissolutions dates, where the dissolution date is also a Painter's birthdate.

SPARQL Pattern

```
SELECT ?c, ?p, ?d
WHERE
{
  ?c rdf:type <http://dbpedia.org/ontology/Country> .
  ?c <http://dbpedia.org/ontology/dissolutionDate> ?d .
  {SELECT ?p, ?bd AS ?d
  WHERE
  {
     ?p rdf:type <http://dbpedia.org/ontology/Person> .
     ?p <http://dbpedia.org/ontology/birthDate> ?bd
   }}
}
```

SQL Translation

```
SELECT t1.subject, t2.subject, t1.dissolutionDate
FROM (CT_Country As t1 Join
(SELECT t2.subject, t2.birthDate
FROM CT_Person as t2))
WHERE t2.birthDate = t1.dissolutionDate;
```

IQL Translation

```
Query Country $c/dissolutionDate:$d, Person $p/birthDate:$d
construct $c/$d, $p/$d;
```

Sample Results:

c	p	d
http://dbpedia.org/resource/State_of_Katanga	http://dbpedia.org/resource/Bruce_Schneier	1963-01-15
http://dbpedia.org/resource/Surinam_(Dutch_colony)	http://dbpedia.org/resource/Alex_Cox	1954-12-15
http://dbpedia.org/resource/Curaçao_and_Dependencies	http://dbpedia.org/resource/Alex_Cox	1954-12-15
http://dbpedia.org/resource/French_Cochinchina	http://dbpedia.org/resource/Harry_Turtledove	1949-06-14
http://dbpedia.org/resource/Duchy_of_Brunswick	http://dbpedia.org/resource/Hermann_Zapf	1918-11-08
http://dbpedia.org/resource/Mutawakkilite_Kingdom_of_Yemen	http://dbpedia.org/resource/Melissa_Sue_Anderson	1962-09-26

5.5.11 Pattern Involving GROUP BY with COUNT

Description

Count the number of kingdoms for each plant in the plant class.

SPARQL Pattern

```
SELECT ?plant, COUNT(?k) AS ?numKingdom
WHERE
{
  ?plant rdf:type <http://dbpedia.org/ontology/Plant> .
  ?plant <http://dbpedia.org/ontology/kingdom> ?k
} GROUP BY ?plant
```

SQL Translation

```
SELECT t1.subject, count(t2.object)
FROM (CT_Plant As t1 Join PT_kingdom as t2 ON t1.subject=t2.subject)
GROUP BY t1.subject;
```

IQL Translation

```
Query Plant $plant/kingdom:$k
construct $plant[Count({$k})];
```

Sample Results

plant	numKingdom
http://dbpedia.org/resource/Brachyscome_scapigera	1
http://dbpedia.org/resource/Coprosma_acerosa	1
http://dbpedia.org/resource/Iva_xanthiifolia	1
http://dbpedia.org/resource/Melanophylla_crenata	1
http://dbpedia.org/resource/Pachycormus_discolor	1
http://dbpedia.org/resource/Croptilon	1
http://dbpedia.org/resource/Cytisopsis	1
http://dbpedia.org/resource/Nephrodesmus	1

http://dbpedia.org/resource/Parapiptadenia	1
http://dbpedia.org/resource/Bletia_purpurata	1
http://dbpedia.org/resource/Dorema_aucheri	1

5.5.12 Pattern Involving ORDER BY

Description

Query all mammals and their genus, ordered by mammals and ordered their genus.

SPARQL Pattern

```
SELECT ?mammal, ?genus
WHERE
{
  ?mammal a <http://dbpedia.org/ontology/Mammal> .
  ?mammal <http://dbpedia.org/ontology/genus> ?genus
} ORDER BY ?mammal ?genus
```

SQL Translation

```
SELECT t3.subject, t3.object as genus
FROM (
SELECT t1a.subject, t2a.object
FROM CT_Mammal as t1a join pt_genus as t2a ON t1a.subject = t2a.subject
UNION
SELECT t1b.subject, t2b.object
FROM CT_Dog as t1b join pt_genus as t2b ON t1b.subject = t2b.subject
UNION
SELECT t1e.subject, t2e.object
FROM CT_Cat as t1e join pt_genus as t2e ON t1e.subject = t2e.subject
UNION
SELECT t1c.subject, t2c.object
FROM CT_Horse as t1c join pt_genus as t2c ON t1c.subject = t2c.subject
UNION
SELECT t1d.subject, t2d.object
FROM CT_RaceHorse as t1d join pt_genus as t2d ON t1d.subject = t2d.subject) as t3
ORDER BY t3.subject, genus
```

IQL Translation

```
Query Mammal* $mammal/genus:$g
```

construct $mammal order by $mammal [$g order by $g];

Sample Results

mammal	genus
http://dbpedia.org/resource/%3FOryzomys_pliocaenicus	http://dbpedia.org/resource/Bensonomys
http://dbpedia.org/resource/%3FOryzomys_pliocaenicus	http://dbpedia.org/resource/Jacobsomys
http://dbpedia.org/resource/%3FOryzomys_pliocaenicus	http://dbpedia.org/resource/Oryzomys
http://dbpedia.org/resource/'Anchomomys'_milleri	http://dbpedia.org/resource/Anchomomys
http://dbpedia.org/resource/ABC_Islands_bears	http://dbpedia.org/resource/Ursus_(biology)
http://dbpedia.org/resource/Aba_roundleaf_bat	http://dbpedia.org/resource/Hipposideros
http://dbpedia.org/resource/Aberdare_mole_shrew	http://dbpedia.org/resource/Surdisorex
http://dbpedia.org/resource/Abert's_squirrel	http://dbpedia.org/resource/Sciurus

5.5.13 Pattern Involving OFFSET and LIMIT

Description

Query 10 mammals.

SPARQL Pattern

```
SELECT ?mammal
WHERE
{
  ?mammal a <http://dbpedia.org/ontology/Mammal> .
} LIMIT 10
```

SQL Translation

```
SELECT subject
FROM (SELECT t1.subject
        FROM CT_Mammal as t1
        UNION
        SELECT t1.subject
        FROM CT_Cat as t1
        UNION
        SELECT t1.subject
        FROM CT_Dog as t1
        UNION
```

```
        SELECT t1.subject
        FROM CT_Horse as t1
        UNION
        SELECT t1.subject
        FROM CT_RaceHorse as t1)
LIMIT10
```

IQL Translation

```
Query Mammal* $mammal
construct $mammal top(1,10);
```

Sample Results[32]

mammal
http://dbpedia.org/resource/American_black_bear
http://dbpedia.org/resource/Beefalo
http://dbpedia.org/resource/Black_rat
http://dbpedia.org/resource/Brown_rat
http://dbpedia.org/resource/Donkey
http://dbpedia.org/resource/Even-toed_ungulate
http://dbpedia.org/resource/Galago
http://dbpedia.org/resource/Hare
http://dbpedia.org/resource/Lagomorpha
http://dbpedia.org/resource/Mule

5.6 Summary of Storing and Querying DBPedia

The results show that storing RDFS is possible in both MySQL and INMDB. Overall the schema of INMDB permits the representation of more RDFS features then what can be done with a relational or object-relational database.

Querying RDF in INMDB has a clear set of advantages. Query syntax is extremely easy to use compared to MySQL's query syntax. Table 34 shows the execution

[32] Note: the ordering of these outputs is different between INMDB and MySQL

times for each of the sample queries presented in section 5.5 for both INMDB and MySQL. In this table: green indicates better performance than MySQL, yellow indicates similar performance to MySQL and red indicates worse performance than MySQL. As shown, most queries are faster in INMDB than in MySQL, however some database functions (FILTER and SUBQUERY) take much longer to process.

Table 34: DBPedia Sample Query Time Results

Q #	Sample Query	INMDB (seconds[33])	MySQL (seconds[33])
5.5.1	Basic Graph Pattern Belonging to a Class and it's Subclasses	48.6	31.7
5.5.2	Pattern with JOIN (subject-subject)	1.5	7.7
5.5.3	Pattern with JOIN (subject-object)	1.3	1.5
5.5.4	UNION of Two BGPs	0.6	5.4
5.5.5	Pattern with OPTIONAL JOIN	0.5	3.6
5.5.6	Pattern with more than One OPTIONAL JOIN	0.2	4.0
5.5.7	Pattern with a FILTER	113.0	0.9
5.5.8	Pattern with UNION and OPTIONAL	7.3	5.1
5.5.9	Pattern Involving MINUS	1.5	11.3
5.5.10	Pattern Involving SUBQUERY	Over 500	45.8
5.5.11	Pattern Involving GROUP BY with COUNT	19.291	48.781
5.5.12	Pattern Involving ORDER BY	3.564	15.631
5.5.13	Pattern Involving OFFSET and LIMIT	0.404	14.007

INMDB has a clear set of advantages for storing and querying RDFS, but also some minor disadvantages. The advantages and disadvantages of using INMDB for RDF storage are shown below.

Advantages of storing RDF in INMDB

- Ability to semantically represent the following RDFS features:
  - RDFS classes represented as database classes
  - Property inheritance
  - Multi-valued attributes
  - Sub-Attributes and Sub-Relationships
  - Subject-object relationships

[33] Seconds measured in "Real" Time (time it takes to output all results to console). Based on the observed variability when re-executing queries, these times can be assumed accurate ±0.15s.

        - Inverse Subject-object relationships
    - Schema is more straight forward to generate then MySQL. This is because in MySQL extra steps need to be taken to manage multivalued data, whereas with INMDB it is automatically handles multi-valued attributes for the user.
    - Data input scripts are easy to generate
    - Querying RDF has very simple syntax compared to SQL.
        - Class and sub-class queries are extremely simple, and simplify all queries that require these. The corresponding SQL for these types of queries can easily explode into several pages of query syntax.
        - Querying subject-object joins is extremely simple
        - Querying multi-valued attributes is extremely simple.
    - Querying RDF is often much faster in INMDB than in MySQL.
    - Translating SPARQL to IQL is straight forward because both languages are declarative graph query languages.

Disadvantages of storing RDF in INMDB

- It is more difficult to manage INMDB's schema compared to MySQL when working with dynamic datasets where the RDFS schema changes regularly. This is because links must be managed within the schema in INMDB which poses additional management issues, where MySQL does not have to deal with this.
- Storage of the DBPedia dataset takes up a very large amount of disk space 41.6GB compared to MySQL 3.6GB. This is due to extensive indexing in INMDB. Approximately 10x storage space.
- The entire DBPedia dataset was loaded much faster into MySQL then INMDB. This is because INMDB must build several indexes.
- Not all RDF queries are supported (i.e., Bind).
- Some queries can be overly complicated (i.e., optional joins), however new versions of IQL can address this issue.
- Some queries (FILTER and SUBQUERY) take longer to complete than in MySQL.

Chapter 6

Conclusion and Future Work

In conclusion, we have demonstrated an RDF storage approach that is able to represent more RDFS features with database schema than existing RDF storage solutions by using INMDB's capacity for handling RDF queries. INMDB is able to represent RDFS data using its schema in a way that is both simple and natural by taking advantage of multiple INMDB features such as: property inheritance, sub-attributes, multi-valued attributes and linking.

The work done in the thesis is as follows. A novel algorithm was given for loading logically consistent, static RDFS datasets (snapshots) into INMDB (section 5.3.1). In addition, a companion algorithm for loading RDFS data into MySQL was also presented (section 5.3.1.1). A systematic way of translating SPARQL graph queries to INMDB's query language was presented (section 4.2.1). Then, by using several SPARQL queries that are representative of its features, it was verified whether or not those queries are translatable to INMDB query language (IQL) and whether they are supported by INMDB (section 4.2.2). The performance of the SPARQL queries translated to IQL was compared to their equivalent SQL translation, and executed on the DBPedia dataset (section 5.5). All of the work done in the thesis contributed to the investigation of whether or not it is possible to represent, store and query RDFS data in INMDB.

The results of the investigation are that many RDF queries are supported in INMDB and those that are, often have much simpler syntax and are faster to execute then their equivalent SQL query. Most notably of these supported queries are those that

involve patterns matching a class and its sub-classes, subject-subject joins and subject-object joins, which are much more naturally composed in IQL syntax than in SQL. The level of support for RDF query features provided by INMDB currently, allows for many common or typical user queries to be asked. Furthermore, the results show that INMDB is able to represent RDFS as database schema better than existing RDF storage approaches and is capable of storing real RDFS datasets such as DBPedia.

The novel contributions made by the thesis are summarized as follows. A novel mapping from RDFS to database schema that is able to capture more features of RDFS then existing mappings was presented. An experimental assessment determining if INMDB is able to represent, store and query RDFS. This included an evaluation of supported SPARQL queries on INMDB and the comparison of its relative query performance to a relational database system. Finally, an algorithm for loading static RDFS datasets into INMDB and A systematic way of translating SPARQL queries to the query language of INMDB were presented.

As of writing this thesis work is being undertaken to develop a distributed database version of INMDB. When this software becomes available, it will be of interest to test how it compares to the current version of INMDB for RDF storage as well as to investigate its feasibility as a distributed RDF datastore. If INMDB's semantic abilities can be applied to a distributed storage approach this would be very useful for real world RDF storage solutions. Furthermore, future work will include an investigation of life cycle management for INMDB and support for dynamic datasets (e.g., datasets requiring update, delete, insert statements), which will build the fundamental components required for an RDFS storage application for managing RDFS datasets.

References

1. Lassila O and Swick R. Resource Description Framework (RDF) Model and Syntax Specification. W3C Working Draft. 1999.
2. Shepherd A, Kerschberg L. Prism: a Knowledge Based System for Semantic Integrity Specification and Enforcement in Database Systems. In Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data (SIGMOD). Jun 18, 1984. pp. 307-315.
3. Spackman KA, Campbell KE, Côté RA. SNOMED RT: a Reference Terminology for Health Care. In Proceedings of the 1997 American Medical Informatics Association annual fall symposium (AMIA). 1997. pp. 640.
4. Soualmia LF, Golbreich C, Darmoni SJ. Representing the MeSH in OWL: Towards a Semi-Automatic Migration. KR-MED 102. pp. 81-87. 2004.
5. Szekely P, Knoblock CA, Slepicka J, Philpot A, Singh A, Yin C, Kapoor D, Natarajan P, Marcu D, Knight K, Stallard D. Building and Using a Knowledge Graph to Combat Human Trafficking. In Proceedings of Part II of the 14th International Semantic Web Conference (ISWC). Oct 11, 2015. pp. 205-221.
6. Kerdjoudj F, Curé O. RDF Knowledge Graph Visualization from a Knowledge Extraction System. CoRR. 2015. pp. 1-17.
7. Shi L, Li S, Yang X, Qi J, Pan G, Zhou B. Semantic Health Knowledge Graph: Semantic Integration of Heterogeneous Medical Knowledge and Services. BioMed Research International. Article ID 2858423. 2017. pp. 1-12.

8. Fadi M, John E. Data Catalog Vocabulary (DCAT). W3C Recommendation. Jan 16, 2014.

9. Weibel S, Kunze J, Lagoze C, Wolf M. Dublin Core Metadata for Resource Discovery. September 1998. pp. 1-8.

10. Quilitz B, Leser U. Querying Distributed RDF Data Sources with SPARQL. In Proceedings of the 5th European Semantic Web Conference. Springer. Jun 1, 2008. pp. 524-538.

11. Hausenblas M, Halb W, Raimond Y, Heath T. What is the Size of the Semantic Web. In I-Semantics 2008: International Conference on Semantic Systems (ICSS). Sep 3, 2008. pp. 9-16.

12. Dieter F, Federico Michele F, Elena S, Toma I. Semantic Web Services. Springer. ISBN 3642191924. 2011. pp. 99.

13. Anja J, Richard C, Chris B. State of the LOD Cloud 2011. University of Mannheim. 2011.

14. Max S, Christian B, Heiko P. State of the LOD Cloud 2014. University of Mannheim. 2014.

15. Schmachtenberg M, Bizer C, Paulheim H. Adoption of the Linked Data Best Practices in Different Topical Domains. In Proceedings of the 13th International Semantic Web Conference (ISWC). Oct 19, 2014. pp. 245-260.

16. Beek, W, Rietveld L, Bazoobandi HR, Wielemaker J, Schlobach S. LOD Laundromat: A Uniform Way of Publishing Other People's Dirty Data. In Proceedings of the 13th International Semantic Web Conference (ISWC). Oct 19, 2014. pp. 213-228

17. David C. Faye, Olivier Cure, Guillaume Blin. A Survey of RDF Storage Approaches. Revue Africaine de la Recherche en Informatique et Mathematiques Appliquees 15. INRIA. pp. 11-35. 2012.

18. Ma Z, Capretz MAM, Yan L. Storing massive Resource Description Framework (RDF) data: a Survey. The Knowledge Engineering Review 31(4). 2016. pp. 391-413.

19. Özsu M. A Survey of RDF Data Management Systems. Frontiers of Computer Science 10(3). Springer. 2016. Pp. 418-432.

20. Nitta K, Savnik I. Survey of RDF Storage Managers. In Proceedings of the Sixth International Conference on Advances in Databases, Knowledge, and Data Applications (DBKDA). Apr 20, 2014. pp. 148-153.

21. Modoni GE, Sacco M, Terkaj W. A Survey of RDF Store Solutions. In Proceedings of the 2014 International Conference on Engineering, Technology and Innovation (ICE). IEEE. Jun 23, 2014. pp. 1-7.

22. Cheng J, Ma ZM, Tong Q. RDF Storage and Querying: A Literature Review. Handbook of Research on Innovative Database Query Processing Techniques. 2015. pp. 460.

23. Albahli S, Melton A. RDF Data Management: A Survey of RDBMS-Based Approaches. In Proceedings of the 6th International Conference on web intelligence, mining and semantics (WIMS). Jun 2016. p.31.

24. Hertel A, Broekstra J, Stuckenschmidt H. RDF Storage and Retrieval Systems. Handbook on ontologies. Springer. 2009. pp. 489-508.

25. del Mar Roldan-Garcia M., Aldana-Montes JF. A Survey on Disk Oriented Querying and Reasoning on the Semantic Web. In Proceedings of 22nd International Conference on Data Engineering (ICDE) Workshops. IEEE. Apr 3, 2006. pp. 58.

26. Prud'Hommeaux E, Seaborne A. SPARQL Query Language for RDF. W3C recommendation. Jan 15, 2008.

27. Pérez J, Arenas M, Gutierrez C. Semantics and Complexity of SPARQL. ACM TODS 34(3). ACM. Nov 5, 2006. pp. 30-43.

28. McBride B. Jena: Implementing the RDF Model and Syntax Specification. In Proceedings of the Second International Conference on Semantic Web (ISWC). CEUR-WS. May 1, 2001. pp. 23-28.

29. Harris S, Lamb N, Shadbolt N. 4store: The Design and Implementation of a Clustered RDF Store. In Proceedings of the 5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS). Oct 25, 2009. pp. 94-109.

30. Neumann T, Weikum G. The RDF-3X Engine for Scalable Management of RDF Data. The VLDB Journal 19(1). Springer. 2010. pp. 91–113.

31. Güting RH. GraphDB: Modeling and Querying Graphs in Databases. In Proceedings of 20th International Conference on Very Large Data Bases (VLDB). Sep 12, 1994. pp. 297-308.

32. Dehainsala H, Pierra G, Bellatreche L. Ontodb: An Ontology-Based Database for Data Intensive Applications. In Proceedings of the 12th International Conference

on Database Systems for Advanced Applications. Springer. Apr 9, 2007. pp. 497-508.

33. Leeka J, Bedathur S. RQ-RDF-3X: Going Beyond Triplestores. In Proceedings of the Data Engineering Workshops (ICDEW), 2014 IEEE 30th International Conference. Mar 31, 2014. pp. 263-268.

34. Neumann T, Weikum G. RDF-3X: a RISC-Style Engine for RDF. Proceedings of the VLDB Endowment 1(1). 2008. pp. 647-659.

35. Zou L, Mo J, Chen L, Özsu MT, Zhao D. gStore: Answering SPARQL Queries via Subgraph Matching. Proceedings of the VLDB Endowment 4(8). pp. 482-493. 2011.

36. Zou L, Özsu MT, Chen L, Shen X, Huang R, Zhao D. gStore: a Graph-Based SPARQL Query Engine. The VLDB Journal 23(4). 2014. pp.565-590.

37. Abadi DJ, Marcus A, Madden SR, and Hollenbach K. Scalable Semantic Web Data Management Using Vertical Partitioning. In Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB). VLDB Endowment. Sep 23, 2007. pp. 411–422.

38. Bornea MA, Dolby J, Kementsietsidis A, Srinivas K, Dantressangle P, Udrea O, Bhattacharjee B. Building an Efficient RDF Store Over a Relational Database. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD). Jun 22, 2013. pp. 121-132.

39. Chebotko A, Lu S, Fei X, Fotouhi F. RDFProv: A Relational RDF Store for Querying and Managing Scientific Workflow Provenance. Data & Knowledge Engineering 69(8). Elsevier. 2010. pp. 836-865.

40. Broekstra J, Kampman A, Van Harmelen F. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In Proceedings of the First International Semantic Web Conference (ISWC). Springer. Jun 9, 2002. pp. 54-68.

41. Wilkinson K, Sayers C, Kuno H, Reynolds D. Efficient RDF Storage and Retrieval in Jena2. In Proceedings of the First International Conference on Semantic Web and Databases. CEUR-WS. Sep 7, 2003. pp. 120-139.

42. Chong EI, Das S, Eadon G, Srini-Vasan J. An Efficient SQL-Based RDF Querying Scheme. In Proceedings of the 31st International Conference on Very Large Data Bases (VLDB). Oct 4, 2005. pp. 1216-1227.

43. Harris S, Gibbins N. 3store: Efficient Bulk RDF Storage. In Proceedings of the 1st International Workshop on Practical and Scalable Semantic Systems. 2003. pp. 1-15.

44. Beckett D. The Design and Implementation of the Redland RDF Application Framework. Computer Networks 39(5). Aug 2002. pp. 577-588.

45. Reggiori A, van Gulik DW, Bjelogrlic Z. Indexing and Retrieving Semantic Web Resources: the RDFStore Model. In Proceedings of the SWAD-Europe Workshop on Semantic Web Storage and Retrieval. Nov 13, 2003. pp. 13-14.

46. Erling O, Mikhailov I. RDF Support in the Virtuoso DBMS. Networked Knowledge-Networked Media. Springer. 2009. pp. 7-24.

47. Albahli S, Melton A. ohStore: Ontology Hierarchy Solution to Improve RDF Data Management. In Proceedings of the 9th International Conference for Internet Technology and Secured Transactions (ICITST). IEEE. Dec 8, 2014. pp. 340-348.

48. Albahli S, Melton A. TripleFCA: FCA-Based Approach to Enhance Semantic Web Data Management. In Proceedings of the IEEE 40th Annual Computer Software and Applications Conference (COMPSAC). Jun 10, 2016. pp. 625-630.

49. Faye D, Curé O, Blin G, Thiam C. RDF Triples Management in roStore. In Proceedings of the 22èmes Journées francophones d'Ingénierie des Connaissances. 2012. pp. 755-770.

50. Ma L, Su Z, Pan Y, Zhang L, Liu T. Rstar: An RDF Storage and Query System for Enterprise Resource Management. In Proceedings of the thirteenth ACM International Conference on Information and Knowledge Management (CIKM). Nov 13, 2004. pp. 484-491.

51. Bonstrom V, Hinze A, Schweppe H. Storing RDF as a Graph. In Proceedings of the IEEE/LEOS 3rd International Conference on Numerical Simulation of Semiconductor Optoelectronic Devices. Oct 14, 2003. pp. 27-36.

52. Chao CM, Chang PL. Storage and Retrieval of RDF Documents Using an Object-Oriented DBMS. Soochow University, China. 2004. pp. 1-23.

53. Astrova I, Kalja A, Jaeger E, Jones M, Ludascher B, Mock S. Storing OWL Ontologies in SQL3 Object-Relational Databases. In AIC'08 Proceedings of the 8th Conference on Applied informatics and communications. Aug 20, 2008. pp. 99-103.

54. Athanasiadis IN, Villa F, Rizzoli AE. Enabling Knowledge-Based Software Engineering Through Semantic-Object-Relational Mappings. In Proceedings of the 3rd International Workshop on Semantic Web Enabled Software Engineering. Jun 2007. pp. 16-30.

55. Bassiliades N, Vlahavas I. R-DEVICE: an Object-Oriented Knowledge Base for RDF Metadata. International Journal on Semantic Web and Information Systems 2(2). 2006. pp. 24-90.

56. Meditskos G., Bassiliades N. O-DEVICE: An Object-Oriented Knowledge Base System for OWL Ontologies. In Proceedings of the Advances in Artificial Intelligence, 4th Helenic Conference on AI. May 18, 2006. pp. 256-266.

57. Meditskos G, Bassiliades N. A Rule-Based Object-Oriented OWL Reasoner. IEEE Transactions on Knowledge and Data Engineering 20(3). pp. 397-410. 2008.

58. Virgilio RD, Giunchiglia F, Tanca L. Semantic Web Information Management: A Model-Based Perspective. Springer Science & Business Media. ISBN 978-3-642-04328-4. Springer. 2010. p.79.

59. Sakr S, Al-Naymat G. Relational Processing of RDF Queries: a Survey. SIGMOD Record 38(4). ACM. 2010. pp. 23-28.

60. Beckmann JL, Halverson A, Krishnamurthy R, Naughton JF. Extending RDBMSs to Support Sparse Datasets Using an Interpreted Attribute Storage Format. In Proceedings of the 22nd International Conference on Data Engineering (ICDE). IEEE. Apr 3, 2006. p.58.

61. Liu M, Hu J. Information Networking Model. In Proceedings of the 28th International Conference on Conceptual Modeling (ER). Springer. Nov 9, 2009. pp. 131-144.

62. RDF Schema 1.1. W3C recommendation. Feb 25, 2014.

63. Bechhofer S, Harmelen FV, Hendler J, Horrocks I, McGuinness DL, Patel-Schneider PF, Stein LA. OWL Web Ontology Language Reference. W3C recommendation. Feb 10, 2004.

64. Prud'Hommeaux E, Steve Harris, Seaborne A. SPARQL 1.1 Query Language. W3C recommendation. Mar 21, 2013.

65. Miller L, Seaborne A, Reggiori A. Three Implementations of SquishQL, a Simple RDF Query Language. In Proceedings of the First International Semantic Web Conference (ISWC). Springer. Jun 9, 2002. pp. 423-435.

66. Broekstra J, Kampman A. SeRQL: a Second Generation RDF Query Language. In Proceedings of SWAD-Europe Workshop on Semantic Web Storage and Retrieval. Nov 4, 2003. pp. 13-14.

67. Karvounarakis G, Alexaki S, Christophides V, Plexousakis D, Scholl M. RQL: a Declarative Query Language for RDF. In Proceedings of the 11th International Conference on World Wide Web (WWW). ACM. May 7, 2002. pp. 592-603.

68. Haase P, Broekstra J, Eberhart A, Volz R. A Comparison of RDF Query Languages. In Proceedings of the Third International Semantic Web Conference (ISWC). Springer. Nov 7, 2004. pp. 502-517.

69. Hammond T, Hannay T, Lund B. The Role of RSS in Science Publishing. D-Lib Magazine 10(12). 2004.

70. Adida B, Birbeck M, McCarron S, Pemberton S. RDFa in XHTML: Syntax and processing. W3C Recommendation. Oct 7, 2008.

71. Stuhr M, Roman D, Norheim D. LODWheel-JavaScript-based Visualization of RDF Data. In Proceedings of the Second International Conference on Consuming Linked Data. CEUR-WS. Oct 23, 2011. pp. 73-84.

72. Berners-Lee T, Chen Y, Chilton L, Connolly D, Dhanaraj R, Hollenbach J, Lerer A, Sheets D. Tabulator: Exploring and analyzing linked data on the semantic web. In Proceedings of the 3rd International Semantic Web User Interaction Workshop (SWUI). Nov 6, 2006. pp. 159.

73. Huynh D, Karger DR, Quan D. Haystack: A Platform for Creating, Organizing and Visualizing Information Using RDF. In Proceedings of the 8th International Conference on Intelligent User Interfaces (IUI). ACM. 2003. pp. 323.

74. Kalyanpur A, Parsia B, Hendler J. A Tool for Working With Web Ontologies. International Journal on Semantic Web and Information Systems 1(1). 2005. pp. 36-49.

75. Huynh, D., Mazzocchi, S., Karger, D. Piggy Bank: Experience the Semantic Web Inside Your Web Browser. In Proceedings of the 4th International Semantic Web Conference (ISWC). Springer. Nov 6, 2005. pp. 413–430.

76. Harth A, Decker S. Optimized Index Structures for Querying RDF from the Web. In Proceedings of the Third Latin American Web Congress (LA-WEB 2005). IEEE. Nov 2005. pp. 71-80.

77. Adams T, Gearon P, Wood D. Kowari: A Platform for Semantic Web Storage and Analysis. In Proceedings of the XTech 2005 Conference. May 2005. pp. 5-13.

78. McGlothlin JA, Khan L. RDFJoin: A Scalable Data Model for Persistence and Efficient Querying of RDF datasets. University of Texas at Dallas. Tech Report UTDCS-08-09. 2009. pp. 1-12.

79. Weiss C, Karras P, Bernstein A. Hexastore: Sextuple Indexing for Semantic Web Data Management. Proceedings of the VLDB Endowment 1(1). 2008. pp. 1008-1019.

80. Matono A, Amagasa T, Yoshikawa M, Uemura S. A Path-Based Relational RDF Database. In Proceedings of the 16th Australasian Database Conference (ADB). Australian Computer Society, Inc. Jan 30, 2005. pp. 95-103.

81. Zou L, Özsu MT. Graph-Based RDF Data Management. Data Science and Engineering 2(1). 2017. pp. 56-70.

82. Huang J, Abadi DJ, Ren K. Scalable SPARQL Querying of Large RDF Graphs. Proceedings of the VLDB Endowment 4(11). 2011. pp. 1123-1134.

83. Aluç G. Workload Matters: A Robust Approach to Physical RDF Database Design. Doctoral dissertation, University of Waterloo. 2015. pp. 1-227.

84. Janik M, Kochut K. BRAHMS: a Workbench RDF Store and High Performance Memory System for Semantic Association Discovery. In Proceedings of the 4th International Semantic Web Conference (ISWC). Springer. Nov 6, 2005. pp. 431-445.

85. Atre M, Srinivasan J, Hendler JA. BitMat: A Main Memory RDF triple Store. In Proceedings of the 5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS). Jan 2009. pp.33-48.

86. Fletcher GH, Beck PW. Scalable Indexing of RDF graphs for Efficient Join Processing. In Proceedings of the 18th ACM Conference on Information and Knowledge Management (CIKM). Nov 2, 2009. pp. 1513-1516.

87. Oldakowski R, Bizer C, Westphal D. RAP: RDF API for PHP. In Proceedings of the Workshop on Scripting for the Semantic Web (SFSW'05). May 30, 2005. pp. 1-10.

88. Abadi DJ. Query execution in column-oriented database systems. Doctoral dissertation, Massachusetts Institute of Technology. 2008. pp. 1-148.

89. Sinan Yuksel A. An Analysis of RDF Storage Models and Query Optimization Techniques. International Journal of Information Engineering and Electronic Business 7(2). 2015. pp. 20-26.

90. Luo Y, Picalausa F, Fletcher GHL, Hidders J, Vansummeren S. Storing and Indexing Massive RDF Datasets. Semantic search over the web. ISBN 978-3-642-25007-1. Springer. 2012. pp. 31-60.

91. Sidirourgos L, Goncalves R, Kersten M, Nes N, Manegold S. Column-Store Support for RDF Data Management: Not All Swans are White. Proceedings of the VLDB Endowment 1(2). 2008. pp. 1553-1563.

92. Wilkonson K. Jena Property Table Implementation. HP Labs. Tech Report: HPL-2006-140. 2006. pp. 1-13.

93. Levandoski JJ, Mokbel MF. RDF Data-Centric Storage. In Proceedings of the 2009 IEEE International Conference on Web Services (ICWS). Jul 6, 2009. pp. 911-918.

94. Rodriguez-Muro M, Rezk M. Efficient SPARQL-to-SQL with R2RML Mappings. Web Semantics: Science, Services and Agents on the World Wide Web 33. 2015. pp. 141-169.

95. Stonebraker M, Rowe LA. The Design of Postgres. ACM 15(2). 1986. pp. 340-355.

96. Riley G. Clips: An Expert System Building Tool. In Proceedings of the Second National Technology Transfer Conference and Exposition. Dec 3, 1991. pp. 149-158.

97. Kolas D, Emmons I, Dean M. Efficient Linked-List RDF Indexing in Parliament. In Proceedings of the 5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS). 2009. pp. 17-32.