AN EFFICIENT FRAMEWORK FOR INFORMATION RETRIEVAL FROM LINKED DATA

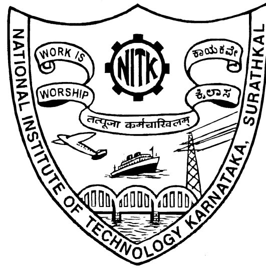
Thesis

Submitted in partial fulfilment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

by

Mr. SAKTHI MURUGAN R



DEPARTMENT OF INFORMATION TECHNOLOGY

NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA

SURATHKAL, MANGALORE - 575025

JUNE 2020

Declaration

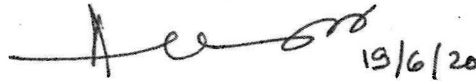
I hereby *declare* that the Research Thesis entitled "An Efficient Framework For Information Retrieval From Linked Data" which is being submitted to the National Institute of Technology Karnataka, Surathkal in partial fulfilment of the requirements for the award of the Degree of Doctor of Philosophy in Information Technology is a *bonafide report of the research work carried out by me*. The material contained in this thesis has not been submitted to any University or Institution for the award of any degree.

Mr. SAKTHI MURUGAN R
Register No.: 145012-IT14P01
Department of Information Technology

Place: NITK Surathkal
Date: 19-JUNE-2020.

Certificate

This is to *certify* that the Research Thesis entitled "An Efficient Framework For Information Retrieval From Linked Data" submitted by Mr. SAKTHI MURUGAN R (Register Number: 145012-IT14P01) as the record of the research work carried out by him, is *accepted as the Research Thesis submission* in partial fulfilment of the requirements for the award of degree of Doctor of Philosophy.



Dr. Ananthanarayana V S
Research Guide
Professor
Department of Information Technology
NITK Surathkal - 575025

Chairman - DRPC
(Signature with Date and Seal)

Chairman - DUGC / DPGC / DRPC
Department of Information Technology
NITK - Surathkal, Srinivasnagar P. O.,
Mangalore - 575 025, INDIA

This thesis is dedicated to my little daughter, *Harshini*.

Acknowledgements

Foremost, I would like to express my sincere gratitude to my research guide *Prof. Ananthanarayana V. S.* for the continuous support of my research, for his patience, motivation, and immense knowledge.

My sincere thanks also goes to the Head of the Information Technology Department, and Research Progress Assessment Committee members for their continuous support and encouragement.

I thank all my fellow doctoral students, teaching and non-teaching staffs of the Department of Information Technology for their cooperation.

Last but not least, I would like to thank my family: my parents, my wife and my brothers for supporting me throughout writing this thesis.

(Mr. SAKTHI MURUGAN R)

Abstract

Linked data is a method of publishing machine-processable data over the web. The resource description framework (RDF) is a standard model for publishing linked data. Currently, distributed compendiums of linked data are available over multiple SPARQL endpoints that can be queried using SPARQL queries. The size of the linked data is steadily increasing as many companies and government organizations have begun adopting linked data technologies. Many frameworks have been proposed for information retrieval from linked data, and the volume of data is a common challenge. Due to the huge volume, many existing linked data search engines have not indexed the latest data. The major components of information retrieval from linked data include storing, partitioning, indexing and ranking. This thesis presents a novel framework for information retrieval from a distributed compendium of linked data called the '*Linked Data Search Framework*', abbreviated as *LDSF*. The significant contributions of *LDSF* include the method of storage, partitioning, indexing and ranking of linked data.

The storage cost of RDF data is one of the primary concerns for searching linked data. The main objective of linked data is to represent the data as URI's in a format that is both human understandable and machine processable. This intermediate URI form of representation is difficult for humans to understand and consumes massive storage in the case of machines. Humans read, think and speak text data as words and not as characters, but computers use character-based encoding such as Unicode to handle text data (including linked data). This thesis presents an approach named '*WordCode*', a word-based encoding of text data (including linked data), that enables computers to store and process text data as words. A trie based code page named '*WordTrie*' is proposed to store words for rapid encoding and decoding. Experimental results from encoding text files from the standard corpus using *WordCode* show an up to 19.9% reduction in file size compared to that achieved with character-based encoding. The proposed *WordCode* method of encoding words in a machine-processable format used less storage space, resulting in faster processing and communication of text data (including linked data).

Query processing in massive linked data is performed by distributing the storage across multiple partitions. Considerable research has been conducted to partition linked data based on clusters. Additionally, substantial research on hash-based partitioning, cloud-based partitioning, and graph-based partitioning has been reported. However, these sophisticated partitioning algorithms are not based on the semantic relatedness

of the data and suffer from high preprocessing cost. In this thesis, a semantic-based partitioning method using a novel *nexus* clustering algorithm is discussed. For every concept, the core properties of the linked data are identified, and bilevel, *nexus*-based hierarchical agglomerative clustering is used to partition the linked data. The proposed method is evaluated using the gold standard test data sampled from DBpedia across eight closely related categories. The proposed clustering technique partitions the linked data with a precision of 98.7% on the gold standard dataset.

Multiple indexing strategies have been proposed to search and access linked data easily at any given time. All these extensive indexing schemes involve substantial redundant data, which greatly increases the required storage and computational resources needed to update the index of the dynamically growing linked data. This thesis introduces '*trist*', a hybrid data structure combining a tree and doubly linked list to index linked data. The linked data contain URIs and values. The URIs and values are separately indexed using '*URI trist*' and '*Value trist*', respectively. Compared to the existing indexing strategies, this indexing approach reduces the storage consumption. The experimental results using the sampled DBpedia dataset demonstrate that trist-based indexing achieves a space-saving of 60% compared to regular graph-based storage of linked data. Also, the proposed trist-based indexing is 6000% faster in accessing the linked data from the graph than the regular graph without indexing.

The ultimate goal of an information retrieval system is to rank the linked data that will be appealing to the end user. The existing approaches for ranking linked data are all atomistic. Often, the problem with ranking linked data is that the data are of various kinds from multiple sources. This thesis presents a holistic approach to rank linked data from multiple SPARQL endpoints and presenting the integrated results. The holistic rank is computed based on four subranks: endpoint rank, concept rank, predicate rank and value rank. LDSF also provides an approach to represent the URI form of linked data to the user in an easily understandable manner. The ordering of Wikipedia is agreed to be readable by its users over the web, and the ranking in this thesis is evaluated based on the ordering of Wikipedia: the proposed ranking correlates up to 99% with the ordering of Wikipedia with the DBpedia sampled dataset.

Overall, the proposed *LDSF* is an efficient framework for storing, partitioning, indexing and ranking linked data that produces a more satisfactory query result than that of existing systems.

Keywords: Semantic Web; Linked Data; RDF; SPARQL; Information Retrieval; Storing; Indexing; Partitioning; Clustering; Ranking; Word Encoding; Text Encoding.

Contents

1 Introduction **1**

1.1 Basic concepts and notation 1

1.1.1 Semantic web 1

1.1.2 Linked data 1

1.1.3 RDF 2

1.1.4 SPARQL query language 4

1.1.5 Triplestore 5

1.1.6 Encoding 6

1.2 State of the art 6

1.2.1 Storing 7

1.2.2 Partitioning 7

1.2.3 Indexing 7

1.2.4 Ranking 8

1.2.5 Research motivation 8

1.3 Methodology 8

1.3.1 Research objectives 10

1.4 Chapter overview 10

2 Storing **11**

2.1 Introduction 11

2.2 Background and related work 12

2.2.1 Background 12

2.2.2 Related work 14

2.3 Proposed word-based text encoding technique 15

2.3.1 Dynamic context-based coding 15

2.3.2 WordCode construction 15

2.3.3 WordCode operation 19

2.4 Proposed trie-based code page 25

2.4.1 Time-optimised WordTrie 26

2.4.2 Space-optimised WordTrie . . . 29
2.5 Experimental results . . . 35
2.5.1 WordCode code page implementation . . . 35
2.5.2 WordCode encoding evaluation . . . 37
2.6 Summary . . . 41

3 Partitioning 42
3.1 Introduction . . . 42
3.2 Related works . . . 43
3.2.1 Graph partitioning . . . 43
3.2.2 Hash partitioning . . . 44
3.2.3 Cloud partitioning . . . 44
3.2.4 Clustering linked data . . . 44
3.3 Proposed cluster-based partitioning . . . 45
3.3.1 Core and common predicates . . . 45
3.3.2 Bilevel *nexus*-based hierarchical agglomerative clustering . . . 47
3.3.3 Nexus clustering algorithm . . . 49
3.4 Experimental setup and evaluation . . . 50
3.4.1 Experimental setup . . . 50
3.4.2 Evaluation metrics . . . 51
3.4.3 Experimental results . . . 53
3.5 Summary . . . 56

4 Indexing 57
4.1 Introduction . . . 57
4.2 Related work . . . 58
4.3 Proposed indexing technique . . . 59
4.3.1 Trist data structure . . . 59
4.3.2 URI trist . . . 60
4.3.3 Value trist . . . 62

4.3.4 Inverted index 63
4.3.5 RDF graph 64
4.4 Experimental results 66
4.4.1 Time-optimised RDF graph 66
4.4.2 Space-optimised RDF graph 66
4.5 Summary 68

5 Ranking **69**
5.1 Introduction 69
5.2 Related work 70
5.3 Proposed ranking technique 72
5.3.1 Endpoint ranking 72
5.3.2 Concept ranking 72
5.3.3 Predicate ranking 73
5.3.4 Value ranking 74
5.3.5 Aggregate ranking approach 74
5.4 User interface: Case of entity browsing 74
5.5 Experimental results 75
5.6 Summary 77

6 Conclusions **78**

References **80**

List of Tables

1.1 Sample statements 3
1.2 Common URI prefix 4

2.1 List of communication task codes 13
2.2 Size of Unicode 17
2.3 WordCode structure 17
2.4 Sample text encoded in Unicode and WordCode 21
2.5 Number of words generated 37
2.6 File size comparison 39
2.7 Normalised file size comparison 39
2.8 WordCode compression ratio and space savings 40
2.9 WordTrie word hit and word miss 40
2.10 Comparison of WordCode with related work 40

3.1 Notation of secondary-level predicates 48
3.2 Data available from the SPARQL endpoints 51
3.3 Clustering precision at various levels 54
3.4 Nexus clustering evaluation using the core occurrence ratio 56
3.5 Cluster comparison 56

4.1 Permutation based indexing of linked data 58
4.2 Object grammar 62
4.3 Storage size comparison of RDF graphs 67
4.4 Compression ratio and space savings 67

5.1 Properties describing concepts 73
5.2 Spearman’s rank correlation coefficients of DBpedia and LDSF . . . 76
5.3 Comparison of LDSF ranking with related work 76

List of Figures

1.1 Triple structure 2
1.2 Triple example 2
1.3 Triple example with URI 3
1.4 High-level architecture of the LDSF 9

2.1 WordCode code page model 19
2.2 Sample code page of WordCode 20
2.3 WordCode housekeeping 21
2.4 WordTrie: Computation-time-optimised version 27
2.5 WordTrie: Storage-space optimised version 30
2.6 Position-to-WordCode mapping 31
2.7 Normalised comparison of file size 39

3.1 Sample subjects with core and common predicates 46
3.2 Slope from common to core predicates 47
3.3 Example clustering 48
3.4 Clustering sample 52
3.5 Mean precision at different levels 55

4.1 Trist structure 60
4.2 URI trist 61
4.3 Value trist 63
4.4 Inverted index 64
4.5 Time optimised RDF graph 65
4.6 Space optimised RDF graph 65

List of Abbreviations

Abbreviation	Meaning
API	Application Program Interface
ASCII	American Standard Code for Information Interchange
BGP	Basic Graph Pattern
COR	Core Occurrence Ratio
DC	Dublin Core
EOF	End of file
FOAF	Friend of a Friend
HDFS	Hadoop Distributed File System
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IANA	Internet Assigned Numbers Authority
IRI	Internationalized Resource Identifiers
JSON	JavaScript Object Notation
LDSF	Linked Data Search Framework
LOD	Linked Open Data
MLR	Machine Learned Ranking
MR	Matching Ratio
OR	Occurrence Ratio
OWL	Web Ontology Language
P2WC	Position to WordCode
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
SKOS	Simple Knowledge Organization System
SPARQL	SPARQL Protocol and RDF Query Language
SWSE	Semantic web search engine
URI	Uniform Resource Identifiers
URL	Uniform Resource Locator
UTF	Unicode Transformation Format
W3C	World Wide Web Consortium
WC2P	WordCode to position
WDA	WordCode decoding algorithm
WEA	WordCode encoding algorithm
WWW	World Wide Web
XML	Extensible Markup Language

Chapter 1

Introduction

This chapter consists of four parts: (1) a brief introduction to the basic concepts and technologies used in this thesis, (2) state-of-the-art methods related to this thesis and its challenges, (3) a brief explanation of the contributions of this thesis, and (4) a short introduction to each of the chapters.

1.1 Basic concepts and notation

This section gives a brief introduction to semantic web technologies such as linked data, RDF, SPARQL, triplestore and SPARQL endpoints. As part of the work is related to text encoding, the basics of encoding are also introduced in this section.

1.1.1 Semantic web

Tim Berners-Lee, who developed a way to link documents together over a network and thus created the World Wide Web, coined the term "*Semantic Web*". The existing web contains a huge amount of information, which in general, can only be understood by humans. The semantic web is an extension of the current web that provides an easier way to find, share, combine and reuse information: it empowers machines to not only present but also to process information.

Tim Berners-Lee expressed his vision of the semantic web as follows:

> "*The semantic web is not a separate web but an extension of the current one, in which information is given well-defined meaning, better enabling computers and people to work in cooperation.*" - Berners-Lee et al. (2001).

According to the World Wide Web Consortium (W3C):

> "*The semantic web provides a common framework that allows data to be shared and reused across application, enterprise, and community boundaries.*"[1]

1.1.2 Linked data

Linked data are at the heart of the semantic web and represent the best practice for publishing structured data on the Web. Linked data, also known as the Web of Data, is a set of design principles for sharing machine-processable data on the web.

[1]https://www.w3.org/2001/sw/

Linked open data (LOD) are a powerful blend of linked data and open data that are both linked and released under an open license that does not impede its reuse for free. One notable example of an LOD set is DBpedia[2] – a crowd-sourced community effort to extract structured information from Wikipedia[3] and make it available on the web.

1.1.3 RDF

The resource description framework (RDF) is a common framework used to create and share linked data on the web. The RDF extends the linking structure of the web to link data using a uniform resource identifier (URI) (Berners-Lee et al., 2005). Items in the RDF are identified and referred to using their URI.

For example, the institute 'NIT-K' is represented in URI as "http://example.org/resource/NIT-K". Unit data in RDF are called triples. A triple is a combination of a resource, a property, and a value (known as the subject, predicate and object of a triple). Figure 1.1 presents the structure of a triple.

Figure 1.1: Triple structure

Let us consider an example statement to obtain a better understanding. Figure 1.2 shows the triple representation for the statement: "NIT-K is located in India". The subject of the statement is 'NIT-K', the predicate is 'located in', and the object is 'India'.

Figure 1.2: Triple example

Consider the two statements "NIT-K is located in India" and "NIT-K is established in 1960". Table 1.1 shows the subjects, predicates and objects of these statements.

[2] https://dbpedia.org
[3] https://www.wikipedia.org

Table 1.1: Sample statements

Subject	Predicate	Object
NIT-K	located in	India
NIT-K	established	1960

Figure 1.3 shows a graphical representation of the statements in Table 1.1 with URIs. The subjects and predicates are always represented using URI. The object may be represented using URI or value. The objects containing value are tagged with the datatype and language. The subject or object nodes represented with URI are called concept. In Figure 1.3, the nodes 'NIT-K' and 'India' are the concepts. This labelled directed graph representation is used for visual explanation; in reality, the RDF model is represented using many formats, such as RDF/XML, N-Triples, Turtle and JSON-LD.

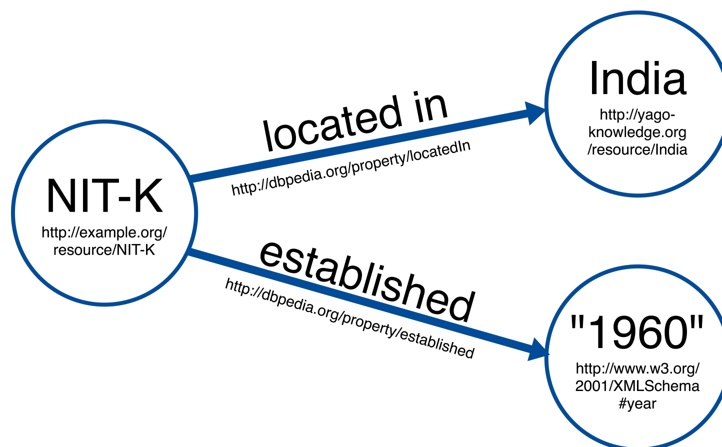


Figure 1.3: Triple example with URI

N-Triple is the simplest form for storing and transmitting RDF triples as $\langle subject \rangle \langle predicate \rangle \langle object \rangle$, terminated with a full stop. The statements from Figure 1.3 are represented in N-Triple format as

```
<http://example.org/resource/NIT-K>
   <http://dbpedia.org/property/locatedIn>
      <http://yago-knowledge.org/resource/India> .
<http://example.org/resource/NIT-K>
   <http://dbpedia.org/property/established>
      "1960"^^<http://www.w3.org/2001/XMLSchema#integer> .
```

Turtle is a much easier RDF format for humans, as the prefix definition shortens the URI and makes it easier to read. The same two statements from Figure 1.3 can be written in turtle format as

```
@prefix eg:  <http://example.org/resource/> .
@prefix dbp: <http://dbpedia.org/property/> .
@prefix ygr: <http://yago-knowledge.org/resource/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

eg:NIT-K  dbp:locatedIn  ygr:India .
eg:NIT-K  dbp:established  "1960"^^xsd:integer .
```

In the above turtle representation, the first four lines define the prefix of the URI, which is later used to write the URIs in their short format. Table 1.2 contains the list of prefix used this thesis to shorten the URI, i.e., the URI '`http://www.w3.org/1999/02/22-rdf-syntax-ns#Property`' is shortened as '`rdf:Property`'. The prefix 'example' is used for illustration of examples.

Table 1.2: Common URI prefix

Prefix	**URL**
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#
dc	http://purl.org/dc/elements/1.1/
foaf	http://xmlns.com/foaf/0.1/
skos	http://www.w3.org/2004/02/skos/core#
dbr	http://dbpedia.org/resource/
dbo	http://dbpedia.org/ontology/
dbp	http://dbpedia.org/property/
example	http://example.org/

1.1.4 SPARQL query language

The SPARQL Protocol and RDF Query Language (SPARQL) is a W3C standard to query RDF data (Clark et al., 2008). SPARQL queries are based on triple patterns. RDF data can be seen as a set of relationships among resources; SPARQL queries provide one or more patterns against such relationships. These triple patterns are similar to

RDF triples, except one or more of the constituent resource references are variables. A SPARQL engine would return the resources for all triples that match these patterns. A sample SPARQL query to find the country and establishment year of 'NIT-K' from Figure 1.3 is

```
@prefix eg:  <http://example.org/resource/> .
@prefix dbp: <http://dbpedia.org/property/> .

SELECT ?country ?establishedYear
WHERE
  {
     eg:NIT-K  dbp:locatedIn  ?country .
     eg:NIT-K  dbp:established  ?establishedYear .
  }
```

In the above SPARQL query, the '?country' and '?establishedYear' are the variables rendered to the output of the SPARQL query.

1.1.5 Triplestore

Triplestore is a database management system (DBMS) used to store and retrieve RDF data. A triplestore is a software program for efficiently storing and indexing RDF triples. Triplestores are queried using SPARQL queries. Most triplestores allow receiving and processing SPARQL query requests via SPARQL endpoints. SPARQL endpoints allow access to triplestores via HTTP. OpenLink Virtuoso[4], GraphDB[5], AllegroGraph[6], RDF4J[7] and Apache Jena Fuseki[8] are popular triplestores. OpenLink Virtuoso is a popular triplestore used for large data sets such as DBpedia[9] and Europe's Government data[10]. Apache Jena[11] is a Java API used to manage RDF data and access triplestores.

[4] https://virtuoso.openlinksw.com
[5] https://www.ontotext.com/products/graphdb/
[6] https://allegrograph.com/products/allegrograph/
[7] https://rdf4j.org
[8] https://jena.apache.org/documentation/fuseki2/
[9] https://dbpedia.org/sparql
[10] http://digital-agenda-data.eu/sparql
[11] https://jena.apache.org

1.1.6 Encoding

Linked data is a kind of text data that uses Unicode-like encoding to store and process data. Encoding is a system of rules for converting data into codes. Codes are sequences of bits used in text encoding to uniquely identify character symbols. The code page is a mapping between elemental characters and their respective codes. Text data (including linked data) are encoded as codes for storage, processing and communication.

Approximately 257 types of character-based text-encoding methods exist[12]. These text-encoding methods have defined code pages. Unicode is a popular encoding method that contains a unique code for almost all the characters of all languages (The Unicode Consortium, 2015). Unicode has three different forms — UTF-8, UTF-16 and UTF-32 — defined by the Universal Coded Character Set (ISO/IEC 10646, 2017). UTF-8 is the encoding technique used by approximately 95.0% of websites on the Internet[13].

1.2 State of the art

The amount of linked data has grown enormously in recent years, with 252 billion triples from 1255 different datasets, and this number is constantly increasing[14]. The need for information retrieval from linked data increases as the number and scale of the semantic web in real-world applications increase. Many search engines use the semantic web and linked data technologies, but the popularity of search engines over linked data is minimal. Prominent examples of search engines over linked data include Swoogle (Ding et al., 2004), Sindice (Tummarello et al., 2007), Falcons (Cheng et al., 2008), Sigma (Tummarello et al., 2010), Watson (d'Aquin & Motta, 2011) and SWSE (Hogan et al., 2011). These linked data search engines are inspired by web search engines and process linked data in the same manner as processing web documents. The results of these linked data search engines are similar to those of a web search: they provide links to data sources with matching keywords. However, the results are unprocessed and difficult to interpret. Although the amount of linked data is enormous, the challenge of indexing such large and dynamic data has limited advancements in research. The major modules of these linked data search engines are crawling, storing, indexing, partitioning and ranking. Unlike web search engines, linked data search engines do not require crawlers, as the links (URIs) may be empty. The actual linked

[12] http://www.iana.org/assignments/character-sets/character-sets.xhtml, Accessed on 12/12/2018.
[13] https://w3techs.com/technologies/overview/character_encoding, Accessed on 21/05/2020.
[14] https://lod-cloud.net, Accessed on June 2, 2020.

data are stored in a DBMS-like engine called a triplestore, which is accessed with simple queries. The extensive surveys conducted by Kaoudi & Manolescu (2015), Özsu (2016), Ma et al. (2016), Pan et al. (2018), Wylot et al. (2018) and Ali et al. (2020) highlight the methods and challenges of storing, partitioning and indexing linked data briefly described in sections 1.2.1, 1.2.2 and 1.2.3 respectively.

1.2.1 Storing

Linked data are stored as a triple table, property table, binary table, mixed table and graph storage. The triple table model storage has considerable redundancy. The property table has multiple null values leading to a sparse table. The binary table model is efficient but has high read and insert costs. Although the mixed table and graph-based model are the best existing routines, the volume of linked data motivates research to find a better storage solution.

1.2.2 Partitioning

Partitioning techniques, such as random partitioning, hash-based partitioning, cloud-based partitioning and graph-based partitioning, have been discussed in previous surveys. The major challenges identified are the following.

1. Most frameworks use sophisticated partitioning techniques that suffer from high preprocessing costs.

2. The partitions are not based on the semantic relatedness, and multiple partitions are often required to answer a query, resulting in high communication and I/O costs.

1.2.3 Indexing

Linked data storage is indexed to accelerate access. Triple-based indexing, aggregate indexing and extensive indexing have been discussed. These existing indexing models have the following challenges:

1. Extensive indexing schemes increase the storage requirement because of the large index size.

2. Rebuilding the index for dynamic data entails substantial huge computation (in the case of HDFS-based indexing).

3. The approach of using the whole index to answer a user query is complex.

1.2.4 Ranking

Ranking linked data is a broad subject. Existing ranking approaches include RDF triple ranking, resource ranking, property ranking and literal ranking. Considerable effort has been made to develop property ranking, as it is widely applied in linked data entity browsing. When the triples are from multiple endpoints, the existing research uses page rank. Most of the proposed ranking approaches are supervised, and those that follow unsupervised approaches produce poor results. No efficient approaches exist for ranking linked data from multiple endpoints.

Considering the challenges and limitations of the existing systems, a different approach to store, partition, index and rank linked data has become our focus.

1.2.5 Research motivation

"To develop a framework to effectively retrieve the information from linked data by incorporating efficient techniques for storing, partitioning, indexing and ranking".

1.3 Methodology

In this thesis, a novel approach for information retrieval from linked data, named the linked data search framework (LDSF) is discussed. The primary goal of LDSF is to provide fast access to enormous quantities of linked data by storing, partitioning, indexing and providing processed results via a ranking. Figure 1.4 shows the high-level architecture of the LDSF.

The primary goal of LDSF is to access linked data present over multiple SPARQL endpoints via simple queries. LDSF allows semantic web users to explore linked data from all SPARQL endpoints with simple SPARQL queries and still obtain coherent results. The main components of LDSF include the storing, partitioning, indexing and ranking of linked data. The linked open data form all the SPARQL endpoints are first stored in the LDSF storage. The major part of the linked data consists of URIs, and the values contain mostly repeated words. Examples of repeated words in URIs include '*http*', '*com*', '*dbpedia*', '*resource*' and '*property*'. These URIs and values are stored in linked data storage using a character encoding technique.

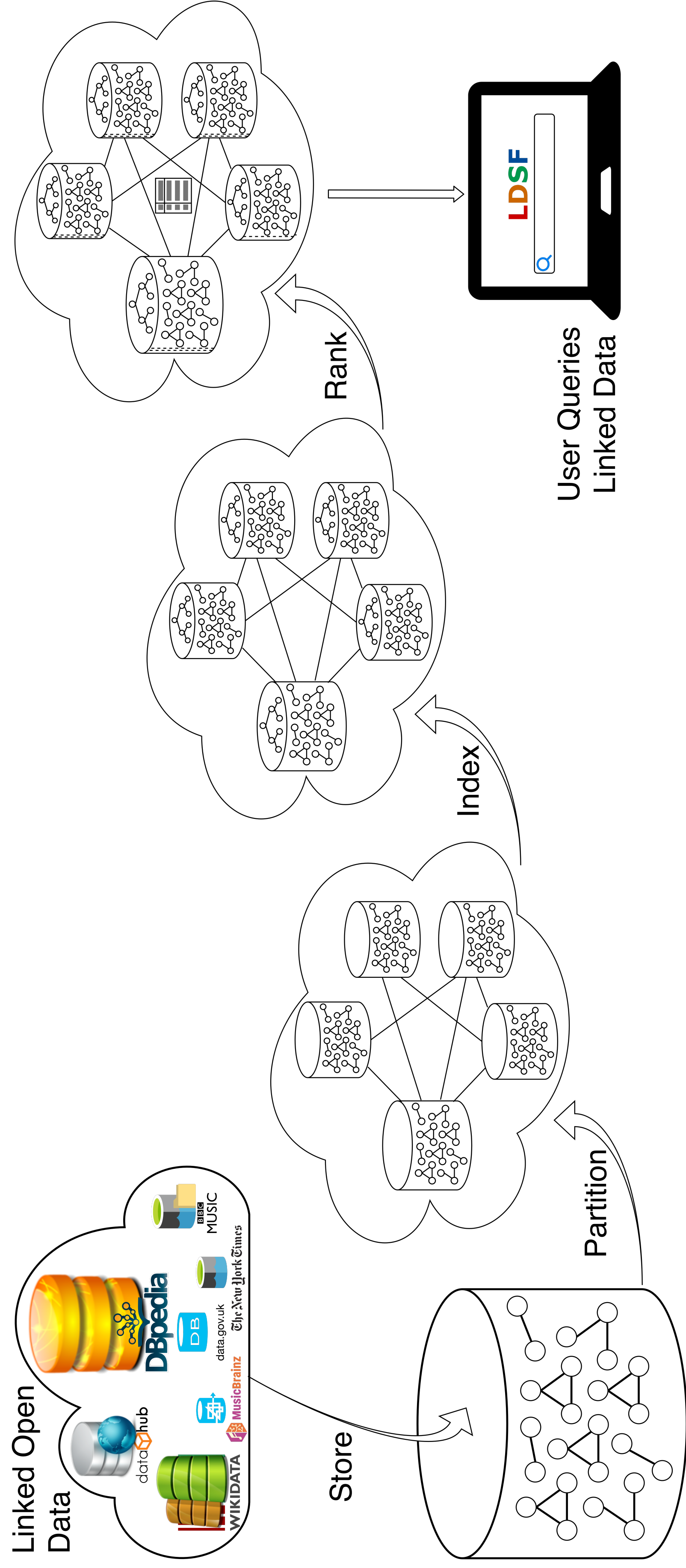


Figure 1.4: High-level architecture of the LDSF

In this thesis, LDSF introduces a word-based encoding technique to store the linked data. This part of the work is generic: the system is applicable not only for linked data but for the entire community of text data, which are often stored as words. This method of storing linked data (and other text data) as words reduces storage consumption. The linked data are then partitioned based on the semantic relatedness of the entity. Entities that are semantically related are identified using a bilevel, *nexus*-based hierarchical agglomerative clustering algorithm. Most of the queries execute with triples with related fields. For example, a single query with biomedical data and real estate data are rare. This type of partitioning of linked data by clustering semantically similar entities reduces the query processing and I/O cost. The linked data are indexed for rapid access to RDF triples. In this thesis, LDSF introduces a trie-based inverted index that consumes less storage space and provides fast access to data. In LDSF, the query results are also given ranks at various levels based on endpoint, concept, predicate and value, so the user querying the data will receive ordered results in a readable format.

1.3.1 Research objectives

The research objectives are defined as follows.

1. To efficiently store linked data.
2. To partition linked data based on semantically similar entity clusters.
3. To efficiently index linked data.
4. To rank linked data from multiple SPARQL endpoints.

1.4 Chapter overview

The technique used to store linked data is discussed in Chapter 2. The method for partitioning linked data based on clusters is presented in Chapter 3. Chapter 4 discusses the approach to indexing linked data. The techniques for ranking linked data are given in Chapter 5. Finally, Chapter 6 concludes this work and proposes directions for future research.

Chapter 2

Storing

2.1 Introduction

Data storage is one of the primary concerns for linked data as the data gets generated at a huge volume and continues to scale. Many tabular and graphical approaches have been proposed to address this issue. All these approaches employ the same character-based encoding for storing data. The character-based encoding uses a sequence of bits to uniquely identify a character symbol. Character-based encodings such as Unicode are used to store RDF files such as RDF/XML, N-Triples, Turtle and JSON-LD. Even triplestores such as OpenLink Virtuoso[1], GraphDB[2], AllegroGraph[3], RDF4J[4] and Apache Jena Fuseki[5] use character-based encodings to store linked data. Data storage is a challenge, not only for linked data but also for text data as a whole.

Humans learn characters while learning a new language, but once the language has become familiar, humans think and speak in words. However, until 2019, computers handled text data at the character level. This chapter proposes a technique for computers to handle text data at the word level by encoding text data directly as words. Even though this chapter aims to present a method for efficiently storing linked data, the proposed word-based encoding is designed to be applied for all types of text data, including linked data. No standard encoding format is accepted by IANA[6] for encoding text data as words. Until 2019, the encoding of text data (including linked data) was based on character encoding, i.e., each character of the text data is represented by a code.

In text data, characters are often used to represent words; similarly, for linked data, URIs and values are formed using words. Because characters are often used to represent words, an encoding technique called WordCode was developed for directly encoding words. A word is a continuous set of letters and numbers that occurs in text data. In general, word-based encodings are not preferred for the following reasons:

[1]https://virtuoso.openlinksw.com
[2]https://www.ontotext.com/products/graphdb/
[3]https://allegrograph.com/products/allegrograph/
[4]https://rdf4j.org
[5]https://jena.apache.org/documentation/fuseki2/
[6]https://www.iana.org/assignments/character-sets/

- Many different languages exist, and each language has its own set of words.
- New words often emerge.
- Substantial memory is required to store all the words of a language.
- Encoding requires a large amount of code page.
- Encoding and decoding require considerable memory and computational time.

All these issues were considered during the development of a technique called WordCode, which dynamically encodes text based on words. In addition, a customised trie, called WordTrie, was introduced for storing the WordCode code page. Moreover, facilities for adding new words to the WordCode code page have been provided. WordCode was designed as a language-independent model to allow words from any language to be encoded. WordCode was designed such that the file size will always be smaller than or equal to that of existing character-based encoding techniques.

The remainder of this chapter is organised as follows. Section 2.2 discusses the background and related work. Section 2.3 provides a detailed description of the proposed word-based text encoding technique. Section 2.4 describes the proposed trie-based code page model. Section 2.5 presents the experimental results of the WordCode encoding for approximately 3058 text files from the standard corpus. Finally, Section 2.6 summarises the work.

2.2 Background and related work

2.2.1 Background

The American Standard Code for Information Interchange (ASCII) serves as the basis for most text-encoding methods, including Unicode. ASCII is a seven-bit code that encodes 128 characters, including numbers, lowercase and uppercase letters, basic punctuation symbols and control codes. Control codes originated with teletype machines with no associated glyphs. ASCII and Unicode have 33 and 64 control codes, respectively. Control codes include codes for text manipulation and communication tasks. Having a single code page defined for both text manipulation and communication tasks wastes considerable code space. The Unicode control codes not used in text representations are listed as communication task codes in Table 2.1.

Table 2.1: List of communication task codes

Sl. No.	Unicode	Decimal	Description
1	U+0001	1	Start of heading
2	U+0004	4	End of transmission
3	U+0005	5	Enquiry
4	U+0006	6	Acknowledge
5	U+0007	7	Bell
6	U+000E	14	Shift out
7	U+000F	15	Shift in
8	U+0010	16	Data link escape
9	U+0011	17	Device control one
10	U+0012	18	Device control two
11	U+0013	19	Device control three
12	U+0014	20	Device control four
13	U+0015	21	Negative acknowledge
14	U+0016	22	Synchronous idle
15	U+0017	23	End of transmission block
16	U+0018	24	Cancel
17	U+0019	25	End of medium
18	U+001A	26	Substitute
19	U+001B	27	Escape

Text data compression is necessary since text data on the Internet and in industry has exponential increased. The encoded files are generally compressed for efficient storage and transmission. Entropy encoding and dictionary coders are popular techniques used to compress character-based text data. Huffman coding by Knuth (1985) and arithmetic coding by Witten et al. (1987) are two popular types of entropy encoding techniques. This model compresses data by replacing the fixed-length code with variable-length code, and the shortest code is used for the most common symbol.

LZ77 and LZ78 by Ziv & Lempel (1977, 1978) are two popular types of dictionary coders. This model compresses data by replacing repeated occurrences of data with a single copy in the header, which is referenced in all subsequent occurrences. The main drawback of these data compression and data decompression models is that they introduce overhead to the system. In the present work, the effort required by these types of compression and decompression methods for directly encoding text based on words is reduced.

This chapter presents a mechanism for encoding words that allows for efficient storage and access. A customised trie, called WordTrie, is introduced. A trie is an ordered

tree data structure in which all the descendants of a node have a common prefix string associated with every node (Fredkin, 1960; Aoe et al., 1992).

2.2.2 Related work

Methods for compressing text data by replacing words from the dictionary have been developed. Azad et al. (2005) designed a model for reducing the file size by providing a lookup table for English words with a 19-bit fixed code. The code space of this model is limited to 524,288 codes and can accommodate only case-independent English-based words. The code page necessary for storing all words will have considerable storage costs. Because the search space is linear, this technique consumes a vast amount of computation time, with the worst case being of $\theta(n)$, where 'n' is the total number of words in the dictionary.

Grabowski & Swacha (2010) recommended language-independent word-based text compression and decompression. This method involves a type of dictionary coder in which the dictionary is generated and attached as a prefix to the compressed file.

Waidyasooriya et al. (2014) considered a word-pair encoding method in which the most frequent character pair is replaced by a character not used in the data. This method limits the encoding to only two characters. Because this method uses the header to store the encodings, this method can be called a compression method.

Sinaga et al. (2015) also designed a trie-based encoding for encoding symbols, conjunctions, prefixes, suffixes, affixes and the most commonly used Indonesian words. Although they have 16-bit code words that can hold 65,536 codes, their initial code page is limited to 903 codes. Case-insensitive words are stored in the trie. Of the 16-bit code, 2 bits are used to differentiate between word cases, which reduces the size of the code page to 16,384 codes; however, mid-word capitalizations, as in 'iPhone', cannot be encoded with their model.

Kalajdzic et al. (2015) proposed a compression technique for short messages by building a small dictionary with 1110 entries from a list of the most frequent words appearing in a large set of messages. This method is not sufficient for mapping all words, and the chances of a word not being found in the dictionary are very high in this case.

Considering the limitations of existing systems, the present study proposes WordCode, with a code page size of 67.9 billion, which is discussed in detail in Section 2.3.

2.3 Proposed word-based text encoding technique

WordCode is an extension of the character-based technique for encoding text based on both characters and words. To implement WordCode, the code page must be constructed and configured on devices similar to Unicode. Because most modern devices are configured with Unicode, WordCode uses a dynamic context-based coding using Unicode codes, as described in Section 2.3.1. Section 2.3.2 illustrates the construction of the WordCode code page. Section 2.3.3 describes the operation of WordCode.

2.3.1 Dynamic context-based coding

The method in which today's webpages are communicated is different from that of teletype machines. Because teletype machines transmitted typed messages, a common code page was defined for both text representation and communication tasks. Today's webpages undergo encryption and compression before being transmitted. Using a common code page such as ASCII and Unicode for handling both communication and text data manipulation increases overhead. Defining separate code pages based on their purpose and using them dynamically will reduce this overhead. In WordCode, the control codes used for communication tasks by Unicode are assigned to a separate code page, and then, these codes are used to represent words. The method for WordCode using the communication task codes of Unicode is described in Section 2.3.2.

2.3.2 WordCode construction

To construct the WordCode code page, a word list should be generated, and a unique code must be allocated for all words. Section 2.3.2.1 describes the method used for word list generation, Section 2.3.2.2 outlines the process of code generation, and Section 2.3.2.3 illustrates the method for allocating the codes to the words.

2.3.2.1 Word list generation

The word list is generated from the words frequently used on the web. Webpages are crawled using web crawlers similar to search engines. The words from the webpages are extracted along with their frequency of occurrence. A word is a combination of characters and numbers, excluding symbols and special characters. Words are case sensitive, i.e., the words '`the`', '`The`' and '`THE`' are treated as different words. Words with frequencies less than the threshold are filtered out when creating the word list.

2.3.2.2 Code generation

A single code page is used to encode both Unicode characters and WordCode words. Unicode has 6400 code points, which range from 57344 to 63743 and are reserved for characters defined by private third-parties; however, WordCode does not use these codes because this require two bytes and is limited to 6400 codes.

The one-byte control codes defined for communication tasks by Unicode are used to encode words in WordCode. Because only 19 control codes are defined for the communication tasks in Table 2.1, they can be used as the prefix with one or more bytecodes as the suffix to represent words in WordCode.

Table 2.2 shows the total number of Unicodes with their size in bytes used by WordCode as the suffix. A total of 128 and 61,299 codes of one and two bytes, respectively, exist. These 61,427 codes can be used in combination with the 19 prefix codes (communication task codes) to encode the words in WordCode. The UTF-16 encoding has 2048 codes, from 55296 to 57343, that cannot be used because these codes are reserved by Unicode.

Table 2.3 shows the structure of WordCode, with the size in bytes. The prefix in Table 2.3 is the communication task code of Unicode, as shown in Table 2.1. The one-byte Unicode in Table 2.3 is the Unicode UTF-8 code, which ranges from 0 to 127 and occupies a single byte of memory, as referred to in Table 2.2. The two-byte Unicode in Table 2.3 is the Unicode UTF-16 code, which ranges from 128 to 55295 and from 57344 to 63474 and occupies two bytes of memory, as given in Table 2.2. WordCode is formed by the prefix code followed by the combination of one-byte Unicode and two-byte Unicode.

Table 2.2: Size of Unicode

Unicode	**Bytes**	**Total Codes**
0-127	1	128
128-55295, 57344-63474	2	61299
55296- 57343	-	2048 (Reserved)

Table 2.3: WordCode structure

Type	**WordCode Template**	**Bytes**	**Range**	**Total Codes**
WC1	<prefix><One-byte Unicode>	2	1-0 to 1-127	128
WC2	<prefix><One-byte Unicode><One-byte Unicode>	3	4-0.0 to 27-127.127	294912
WC3	<prefix><One-byte Unicode><Two-byte Unicode>	4	4-0.128 to 27-127.63474	141232896
WC4	<prefix><Two-byte Unicode><One-byte Unicode>	4	4-128.0 to 27-63474.127	141232896
WC5	<prefix><Two-byte Unicode><Two-byte Unicode>	5	4-128.128 to 27-63474.63474	67636213218

WC1 denotes WordCode type-1 with a prefix code and a one-byte Unicode. WC2 denotes WordCode type-2 with a prefix code and a pair of one-byte Unicodes. WC3 denotes WordCode type-3 with a prefix code, a one-byte Unicode and a two-byte Unicode. WC4 denotes WordCode type-4 with a prefix code, a two-byte Unicode and a one-byte Unicode. WC5 denotes WordCode type-5 with a prefix code and a subsequent pair of two-byte Unicodes.

The first prefix code is used only for type WC1, and the remaining 18 codes are used for types WC2, WC3, WC4 and WC5. The codes from '1-0' to '1-127' are of type WC1. The hyphen in the code indicates the separation between the prefix and suffix of the code and does not occur in the real code page. The codes from '4-0.0' to '27-127.127' are of type WC2. The period in the code is used to illustrate the separation between the two suffix codes and does not occur in the real code page. The codes from '4-0.128' to '27-127.63474' are of type WC3. The codes from '4-128.0' to '27-63474.127' are of type WC4. The codes from '4-128.128' to '27-63474.63474' are of type WC5.

WordCode types WC1 to WC5 can encode a total of 67.9 billion (67,918,974,050) words. Section 2.3.2.3 provides a detailed description of the allocation of these codes to the words generated in Section 2.3.2.1.

2.3.2.3 Code allocation

Figure 2.1 shows a model of the WordCode code page containing the codes for both characters and words. The codes for the characters are the same as those of Unicode, and the codes for the words are those generated in Section 2.3.2.1.

Figure 2.2 presents a sample of the WordCode code page. In Figure 2.2, the column type, bytes and number of codes are given for illustration and are not included in the actual code page of WordCode. Here, the codes '0' to '63474' (excluding the control codes from Table 2.1) represent the Unicode characters. The codes from '1-0' to '27-63474.63474' represent the WordCode words. The words generated in Section 2.3.2.1 are sorted based on length and then frequency. Words with lengths less than or equal to two bytes are not considered because at least two bytes are used to represent a word in WordCode. The codes generated in Section 2.3.2.2 are allocated for the words based on Huffman coding with the following conditions.

Code	Character/ Word
Unicode	Unicode Characters (Excluding Communication Control Codes)
WordCode: Prefix (Communication Task Code) + Code	More Frequent Words ⋮ Less Frequent Words

Figure 2.1: WordCode code page model

1. The length of the WordCode is shorter than the total length of the character code.
2. High-frequency words receive a shorter WordCode.

The codes for words with different cases are different, i.e., the words '`the`' and '`The`' have WordCode codes '`1-0`' and '`1-3`', respectively, as shown in Figure 2.2. Although the WordCode code page can accommodate 67.9 billion words up to WC5, only 3.1 million (31,90,612) words were identified during the crawl, and the code page was created for only 3.1 million words. The remaining codes are reserved for new words that may be found in future crawls.

2.3.3 WordCode operation

Once the code page is ready, it is shared among the systems such that the system can use this code page for encoding text data. Figure 2.3 shows the housekeeping of WordCode. Once the WordCode server generates the code page, it is configured for devices as in Unicode. The text data are then encoded, stored, processed and communicated as a WordCode-encoded file. Meanwhile, the WordCode server checks the Internet for newly generated words. These newly generated words periodically contribute to the next version of the WordCode code page. The newly found words are allocated to the

Code	Character/ Word	Type	Bytes	No. of Codes
0 2 3 8 ⋮ 13 28 ⋮ 63474	Unicode Characters (Excluding Communication Control Codes)	Unicode	1 2	63475
1 - 0 1 - 1 1 - 2 1 - 3 ⋮ 1 - 127	the and that The ⋮ apple	WC1	2	128
4 - 0 . 0 4 - 0 . 1 ⋮ 4 - 127 . 127 5 - 0 . 0 ⋮ 27 - 127 . 127	some good ⋮ current Teacher ⋮ predictable	WC2	3	294912
4 - 0 . 128 ⋮ 4 - 127 . 63474 5 - 0 . 128 ⋮ 27 - 127 . 63474	background ⋮ copyright className ⋮ <Reserved for future words>	WC3	4	141232896
4 - 128 . 0 ⋮ 27 - 63474 . 127	<Reserved for future words> ⋮ <Reserved for future words>	WC4	4	141232896
4 - 128 . 128 ⋮ 27 - 63474 . 63474	<Reserved for future words> ⋮ <Reserved for future words>	WC5	5	67636213218

Figure 2.2: Sample code page of WordCode

unreserved codes in the code page, and WordCode-configured devices send an update request to the WordCode server to update to the latest version of the WordCode code page.

In the web environment, if a client makes an HTTP request for a webpage, then the browser transmits the encoding format as WordCode to handle the text data. The server responds to the user machine with the webpage encoded in WordCode with a version that is less than or equal to the client's WordCode version. On the client side, the WordCode-encoded webpage is displayed in the browser. The method of encoding a

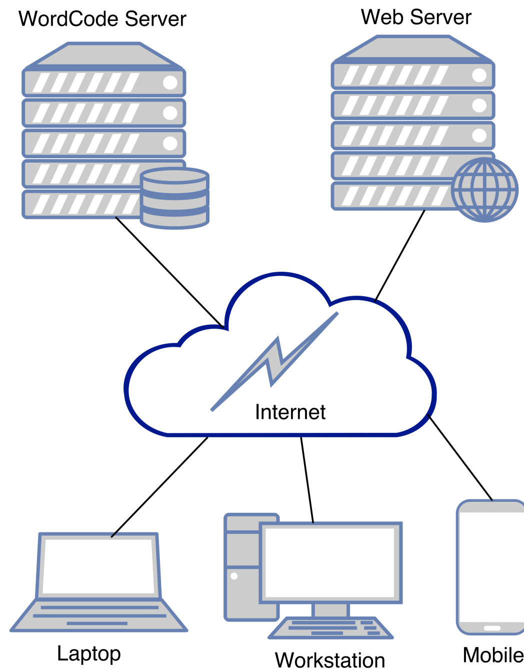


Figure 2.3: WordCode housekeeping

Unicode-coded file to WordCode is described in Section 2.3.3.2, and the procedure for decoding a WordCode-encoded file to Unicode coding is described in Section 2.3.3.3.

2.3.3.1 Comparison of Unicode and WordCode

Table 2.4 shows the sample text "`The apple is good.`" encoded in Unicode and WordCode. In Unicode, each character is defined by a code; in contrast, in WordCode, each word is interpreted by a code, and the symbols, special characters and words that are not in the code page are encoded in Unicode.

Table 2.4: Sample text encoded in Unicode and WordCode

Text	T	h	e		a	p	p	l	e		i	s		g	o	o	d	.
Unicode	84	104	101	32	97	112	112	108	101	32	105	115	32	103	111	111	100	46
WordCode	1-3			32	1-127					32	105	115	32	4-0.1				46

In the sample sentence, the first word '`The`' is indicated by '`84`', '`104`' and '`101`' in Unicode, whereas in WordCode, it is directly encoded as '`1-3`'. The word '`The`' requires three bytes in Unicode, whereas in WordCode, it consumes only two bytes, as '`1-3`' belongs to WC1. The special characters blank space and full stop are denoted using the same Unicode codes used for WordCode, i.e., '`32`' and '`46`', respectively.

In the sample sentence, the word 'apple' is represented by '97', '112', '112', '108' and '101' in Unicode, whereas in WordCode, it is encoded as '1-127'. The word 'apple' requires five bytes in Unicode, whereas in WordCode, it requires only two bytes, as '1-127' belongs to WC1. Because the word 'is' does not have an associated WordCode, it is character encoded with Unicode. In the sample sentence, the word 'good' is represented by '103', '111', '111' and '100' in Unicode, whereas in WordCode, it is directly encoded as '4-0.1'. The word 'good' requires four bytes in Unicode, whereas in WordCode, it requires only three bytes, as '4-0.1' belongs to WC2.

2.3.3.2 WordCode encoding

Algorithm 1 is the WordCode encoding algorithm (WEA). WEA receives a file with Unicode encoding and converts it into a file with WordCode encoding. The 'ReadFirstCharacter' method in WEA is used to read the first character from the Unicode file. The encoding proceeds until it reaches the end of the file (EOF). If the character being read is a prefix code used by WordCode, then WEA returns the Unicode file without encoding the file with WordCode. WEA checks whether the read character is a word character. Word characters are lowercase letters (English), uppercase letters (English) and numbers.

As long as WEA reads a word character, it appends the word character to 'NewWord'. Once WEA encounters a new character 'NewChar' other than a word character, WEA checks whether any word has been read to 'NewWord'. If so, then WEA checks whether the 'NewWord' has a WordCode associated with it on the WordCode code page. If it does have an associated WordCode, then it is a word hit, and WEA writes the WordCode of the 'NewWord'; otherwise, WEA writes the Unicode associated with the 'NewWord'. WEA then continues to read the next character from the Unicode file and repeats the process until it reaches the EOF.

Consider an input Unicode file containing only the single sentence "The apple is good.", as shown in Table 2.4. The algorithm starts to read the first character 'T'. Because it is a word character, the algorithm appends the character 'T' to the 'NewWord =T'. The algorithm continues to read the next character 'h'. Because it is a word character, the algorithm includes the character 'h' in the 'NewWord =Th'.

Algorithm 1: WordCode encoding algorithm (WEA)

```
   Input: UF: File with Unicode encoding.
   Output: WF: File with WordCode encoding.
1  begin
2     NewWord = “”
3     NewChar = ReadFirstCharacter(UF)
4     while NewChar ≠ EOF do
5        if NewChar == PrefixCode then
6           return(UF)
7        end
8        else
9           if NewChar == WordCharacter then
10             NewWord = Append(NewWord, NewChar)
11          end
12          else
13             if NewWord ≠ “” then
14                if WordCode(NewWord) ≠ 0 then
15                   write(WordCode(NewWord), WF)
16                end
17                else
18                   write(Unicode(NewWord), WF)
19                end
20             end
21             write(Unicode(NewChar), WF)
22          end
23       end
24       NewChar ⟵ ReadNextCharacter(UF)
25    end
26    return(WF)
27 end
```

The algorithm continues and reads the next character ‘e’. Because it is a word character, the algorithm adds the character ‘e’ to the ‘NewWord =The’. The algorithm proceeds to read the next character ‘space’. Because it is not a word character, the algorithm checks whether the ‘NewWord =The’ has a WordCode associated with it on the code page. Because ‘NewWord =The’ has an associated WordCode, the algorithm writes the WordCode ‘1-3’ to the WordCode file. The algorithm writes the Unicode code for ‘space’, which has been read. Similarly, the algorithm reads the word ‘apple’ and writes the WordCode ‘1-127’. However, when the algorithm reads the word ‘is’ and searches the WordCode code page, it will not find the code and will

write the Unicode code in the WordCode file. The algorithm continues down until it finishes reading all the characters from the input Unicode file.

2.3.3.3 WordCode decoding

Algorithm 2 is the WordCode decoding algorithm (WDA). The WDA receives a file with WordCode encoding and converts it to a file with Unicode encoding. The '`ReadFirstCode`' method in WDA is used to read the first code from the WordCode file. The decoding proceeds until it reaches the EOF. If the '`NewCode`' is not a prefix code used by WordCode, then it is Unicode, and the same code is written to the output file. If the '`NewCode`' is a prefix code, then the code length of the prefix code is obtained through the '`getCodeLength`' method. Until the code length is reached, the next code is read from the WordCode file and appended to the '`WordCode`'. The '`getWord`' method fetches the word associated with the WordCode from the WordCode code page. The word is written to the file in Unicode code. WDA continues until it reaches the EOF.

Algorithm 2: WordCode decoding algorithm (WDA)

Input: WF: File with WordCode encoding.
Output: UF: File with Unicode encoding.

```
1  begin
2      NewCode = ReadFirstCode(WF)
3      while NewCode ≠ EOF do
4          if NewCode ≠ PrefixCode then
5              write(Unicode(NewCode), UF)
6          end
7          else
8              NewPrefixCode = NewCode
9              CodeLength = getCodeLength(NewPrefixCode)
10             WordCode = NewPrefixCode
11         end
12         for i=1 to CodeLength do
13             WordCode = Append(WordCode, ReadNextCode(WF))
14         end
15         word = getWord(WordCode)
16         for j=1 to word.length() do
17             write(Unicode(word.charAt(j)), UF)
18         end
19         NewCode = ReadNextCode(WF)
20     end
21     return(UF)
22 end
```

Consider an input WordCode file containing only the single sentence "`The apple is good.`", as in Table 2.4, encoded in WordCode. The algorithm starts to read the first code '1'. Because it is a prefix code, the algorithm obtains the code length for the prefix code '1'. For the prefix '1', the suffix code length is one. The algorithm reads the following one code '3' and appends it to the WordCode along with the prefix code as '`WordCode = ` $1-3$'. The word associated with the '`WordCode = ` $1-3$' is looked up in the WordCode code page as '`The`' and written to the Unicode file. The algorithm reads the next code '32'. Because it is a character code, the algorithm writes the same character code as the Unicode. Similarly, the algorithm reads the next code as '1', and because it is a prefix code, the algorithm reads the following suffix code '127'. The algorithm appends the prefix code along with the suffix code to create '`WordCode =` $1-127$'. The word associated with '`WordCode = ` $1-127$' is looked up in the WordCode code page as '`apple`' and written to the Unicode file. The algorithm continues until it has finished reading all the codes from the input WordCode file.

2.4 Proposed trie-based code page

If the code page for all the words is stored in a linear manner, as shown in Figure 2.2, then this will require considerable memory and computation time for encoding and decoding WordCode. For this reason, a hybrid storage method with a trie and an array named WordTrie is proposed here to store all the words. Here, the words are split into characters and stored as nodes in a trie, where words with the same starting substring (prefix) will share nodes of the trie branch starting from the root, thereby reducing the required memory. A tradeoff exists between computational time and computational space; whether to have a lesser computational time by increasing the storage requirement, or to have a lesser storage space by increasing the computational time. To address this issue, two forms of WordTrie are proposed: one with an optimised computation time, which is reviewed in Section 2.4.1, and another with an optimised storage space, which is discussed in Section 2.4.2. WordCode codes can be explicitly stored in the WordTrie or computed. In the time-optimised method, WordCode codes are stored in the WordTrie, thereby increasing the storage space by reducing the computation time. In the space-optimised method, the WordCode is computed, which increases the computation time required and thereby reduces the storage space required.

2.4.1 Time-optimised WordTrie

Figure 2.4 shows the WordTrie with the optimised computation time. Here, the words are split into characters and stored as nodes in the trie, with the first character connected to the root node. Words with the same starting substring (prefix) will share the nodes of the trie branch starting from the root. The array in the time-optimised WordTrie is an array of structures, with each entry containing two fields: the WordCode codes and the pointer connecting the WordCode codes to their respective words. The pointer in the array contains the address of the trie node containing the last character of the word corresponding to the WordCode code.

Thus, if the WordCode code page has 'm' words, then an array of structures of size 'm' is used to store the WordCode code and the respective words' address values. The WordCode codes are stored in the order of the allocation of the code. The nodes containing the last character of the words in the trie are linked to the corresponding WordCode code in the array. The links connecting the nodes of the trie are bidirectional, and the links pointing to the array from the trie are unidirectional.

The method for traversing the time-optimised WordTrie to find the WordCode code associated with a word is described in Section 2.4.1.1. The method for traversing the time-optimised WordTrie to find the word associated with a WordCode code is described in Section 2.4.1.2.

2.4.1.1 Traversing the time-optimised WordTrie to find the WordCode code associated with a word

To find the WordCode for a given word in the time-optimised WordTrie, the WordTrie must be traversed from the root. Whether the root node has a child node that matches the first character of the word is determined, and this process continues until the last character of the word is encountered. Finally, whether a link exists from the last character node to the array is determined. If so, then it is a word hit, and the algorithm will retrieve the WordCode from the array. In cases for which the word cannot be successfully traversed in the trie or for which no link from the last character of the word in the trie to the array exists, it is a word miss.

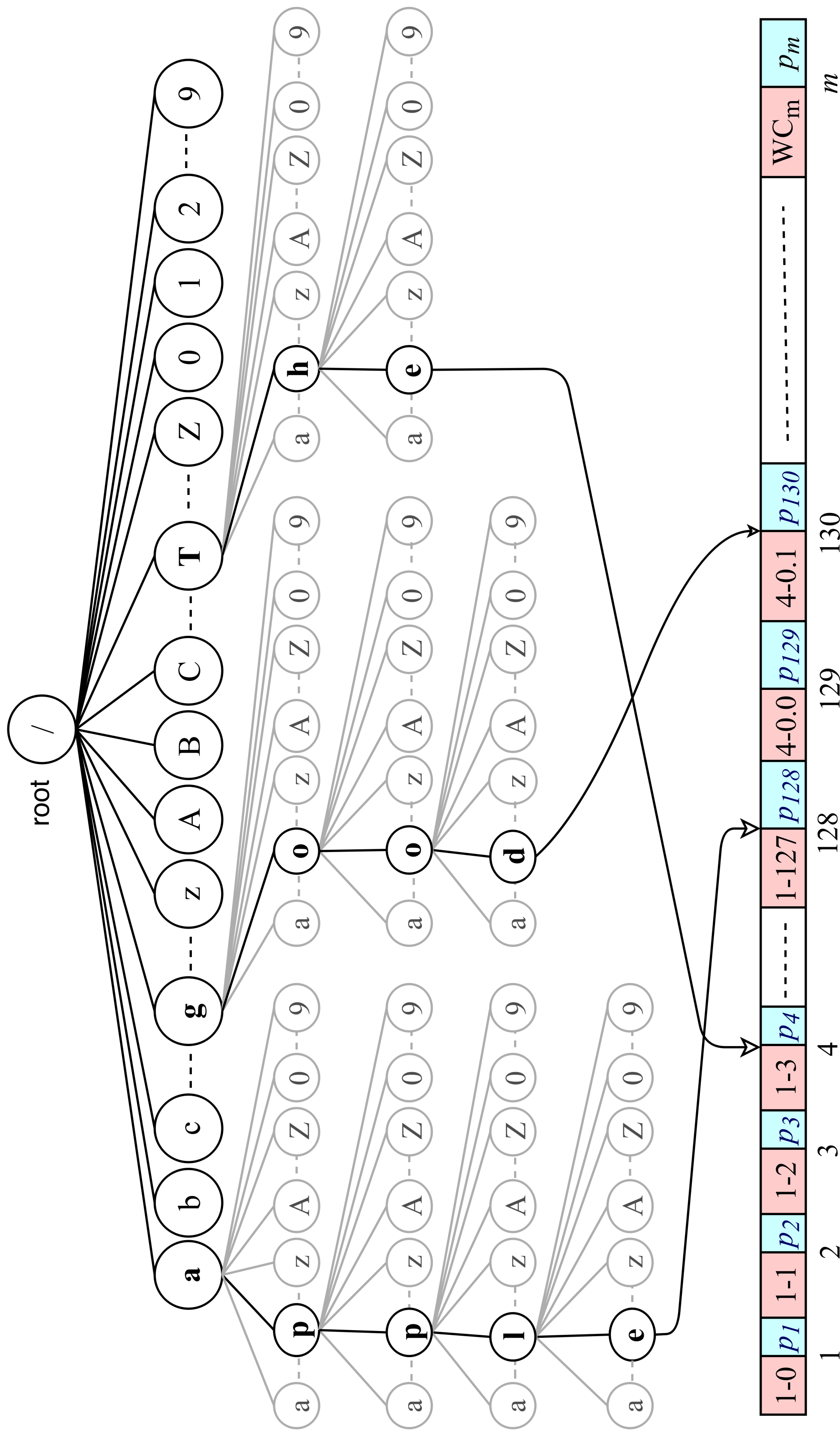


Figure 2.4: WordTrie: Computation-time-optimised version

Consider the word 'The' for which the code must be found in the time-optimised WordTrie. The first character of the word is 'T'; whether the root node has a child 'T' is determined, and the path from the root node to node 'T' is traversed. The next character is 'h'; whether node 'T' has a child node 'h' is determined, and the path from node 'T' to node 'h' is traversed. The last character is 'e'; whether node 'h' has a child node 'e' is determined, and the path from node 'h' to node 'e' is traversed. Finally, whether a link from the last character node 'e' to the array exists is determined, and the algorithm will return the code '1-3', which points to the last character node 'e' in the array.

Let 'n' be the number of characters in the word to be encoded and 'p' be the number of distinct characters in the WordTrie. Traversing the time-optimised WordTrie to find the WordCode code associated with a word is of $\theta(p \times n)$ because, for every character in the word, at most 'p' comparisons are made in the WordTrie at each level.

2.4.1.2 Traversing the time-optimised WordTrie to find the word associated with a WordCode code

To find the word for a given WordCode in the time-optimised WordTrie, the WordTrie must be traversed from the array. Consider the WordCode '1-3' for which the word needs to be found in the time-optimised WordTrie. The node of the array with the WordCode '1-3' is fetched. The pointer in the node that matches the WordCode code contains the address of the trie node containing the last character of the word associated with the WordCode. In this case, the pointer 'p_4', which is in the node of the array with WordCode '1-3', contains the address of the trie node containing the last character of the word associated with the WordCode.

The last character node of the trie is traversed to the root of the trie to fetch the word associated with the code. Node 'e', which is in the address 'p_4', is visited. The algorithm will look for the parent node of 'e', and it will traverse to node 'h'. Because the root node is not reached, the algorithm will again look for the parent node of 'h' and traverse to node 'T'. Because the root node is not reached, the algorithm will again look for the parent node of 'T' and traverse to the root node. Once the root node is reached, the reverse of the path traversed will provide the word associated with the code, and the word 'The' is returned.

Traversing the time-optimised WordTrie to find the word associated with a WordCode code is of $\theta(1)$ because once the WordCode is obtained, the index of the array

with the WordCode is obtained, and the path back to the root node is traversed to obtain the word associated with the WordCode.

2.4.2 Space-optimised WordTrie

Figure 2.5 shows the optimised storage space version of WordTrie, where the words are split into characters and stored as nodes in the trie, with the first character connected to the root node. Words with the same starting substring (prefix) will share common nodes of the trie branch starting from the root. The array in the space-optimised WordTrie is an array of pointers (for 'm' words in the WordCode code page), and an array with 'm' nodes is initialised. The address of the node containing the last character of the word in the trie is stored in the array in the order in which code is allocated for the words. Additionally, the nodes containing the last character of the words in the trie are linked to the corresponding addresses in the array.

The WordCode code is dynamically computed based on the pointer from the node holding the last character of the word in the array using the P2WC and WC2P algorithms, as described in Sections 2.4.2.1 and 2.4.2.2, respectively. Figure 2.6 shows the template for mapping the WordCode code to the position of the array. The links connecting the nodes of the trie are bidirectional, and the links pointing to the array from the trie are unidirectional.

The method for traversing the space-optimised WordTrie to find the WordCode code associated with a word is described in Section 2.4.2.3. The method for traversing the space-optimised WordTrie to find the word associated with a WordCode code is outlined in Section 2.4.2.4.

2.4.2.1 Position to WordCode (P2WC)

Algorithm 3 is the position-to-WordCode (P2WC) algorithm that is used to convert the position of the array in the space-optimised WordTrie to the WordCode code. The '`getPrefixCode(i)`' method is used to obtain the 'i^{th}' prefix code from Table 2.1. The method '`get1ByteCode(j)`' is used to obtain the 'j^{th}' one-byte code, and the '`get2ByteCode(k)`' method is used to obtain the 'k^{th}' two-byte code from Table 2.2.

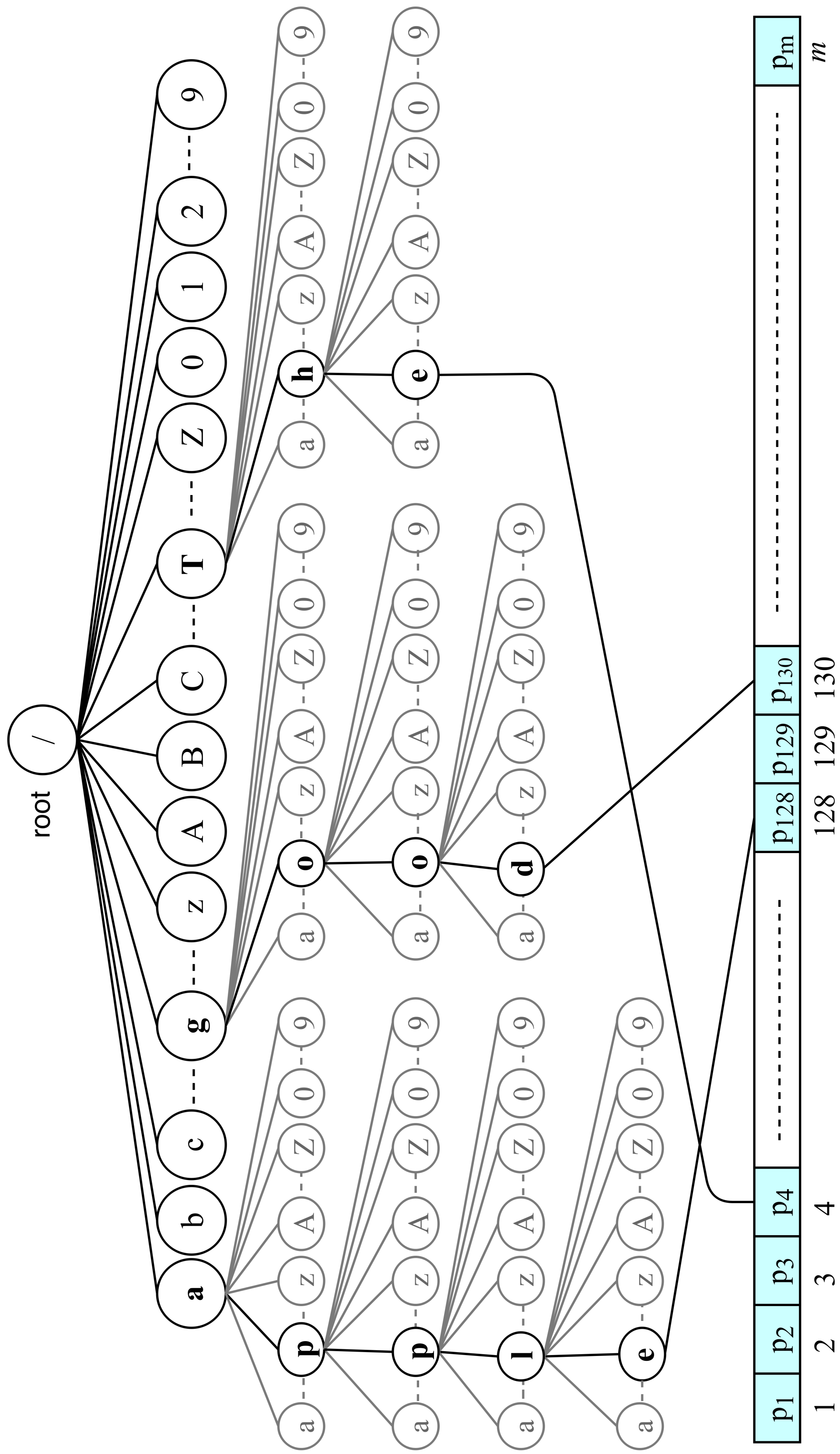


Figure 2.5: WordTrie: Storage-space optimised version

Figure 2.6: Position-to-WordCode mapping

ArrayList position	WordCode	Type
1 2 3 ⋮ 128	1 - 0 1 - 1 1 - 2 ⋮ 1 - 127	WC1
129 130 ⋮ 256 257 ⋮ 16512 16513 ⋮ 295040	4 - 0 . 0 4 - 0 . 1 ⋮ 4 - 0 . 127 4 - 1 . 127 ⋮ 4 - 127 . 127 5 - 0 . 0 ⋮ 27 - 127 . 127	WC2
295041 ⋮ 8141312 8141313 ⋮ 141527936	4 - 0 . 128 ⋮ 4 - 127 . 63474 5 - 0 . 128 ⋮ 27 - 127 . 63474	WC3
141527937 ⋮ 282760832	4 - 128 . 0 ⋮ 27 - 63474 . 127	WC4
282760833 ⋮ 67918974050	4 - 128 . 128 ⋮ 27 - 63474 . 63474	WC5

If the position is less than or equal to '128', then it is of type WC1. For WC1, the prefix value is the first prefix code, and the suffix value is the one-byte code of the position. The '`Append`' method appends the prefix and suffix values to convert it into WordCode. If the position is greater than 128 and less than or equal to $295,040$, then it is of type WC2. First, the position value is subtracted from 129 (one plus the total number of codes of WC1). Then, the prefix index is computed as one plus the ratio of the position to the total number of suffix codes ($128 * 128$). The '`getPrefixCode(Prefix Index)`' method is used to obtain the '`Prefix`' code from Table 2.1. A temporary variable named '`temp`' is used to store the percentage of the position value for the suffixes.

Algorithm 3: Position-to-WordCode (P2WC) algorithm

```
    Input: Position P.
    Output: WordCode WC.
 1  begin
 2      if P > 0 and P <= 128 then
 3          Prefix = getPrefixCode[1]
 4          Suffix = get1ByteCode[P]
 5          WC = Append(Prefix, Suffix)
 6      end
 7      if P > 128 and P <= 295040 then
 8          P = P − 129
 9          Prefix = getPrefixCode[1 + ((P)/(128 * 128))]
10          temp = P%(128 * 128)
11          Suffix_1 = get1ByteCode[temp/128]
12          Suffix_2 = get1ByteCode[temp%128]
13      end
14      if P > 295040 and P <= 141527936 then
15          P = P − 295041
16          Prefix = getPrefixCode[1 + ((P)/(128 * 16299))]
17          temp = P%(128 * 16299)
18          Suffix_1 = get1ByteCode[temp/16299]
19          Suffix_2 = get2ByteCode[temp%128]
20      end
21      if P > 141527936 and P <= 282760832 then
22          P = P − 141527937
23          Prefix = getPrefixCode[1 + ((P)/(16299 * 128))]
24          temp = P%(16299 * 128)
25          Suffix_1 = get2ByteCode[temp/128]
26          Suffix_2 = get1ByteCode[temp%16299]
27      end
28      if P > 282760832 and P <= 67918974050 then
29          P = P − 282760833
30          Prefix = getPrefixCode[1 + ((P)/(16299 * 16299))]
31          temp = P%(16299 * 16299)
32          Suffix_1 = get2ByteCode[temp/16299]
33          Suffix_2 = get2ByteCode[temp%16299]
34      end
35      WC = Append(Prefix, Suffix_1, Suffix_2)
36      return(WC)
37  end
```

The '`Suffix_1`' is computed as the ratio of '`temp`' to the total number of codes in '`Suffix_2`' (128). The '`Suffix_2`' is calculated as the percentage of '`temp`' to the

total number of codes in '`Suffix_1`' (128). Finally, the '`Append`' method appends the '`Prefix`', '`Suffix_1`' and '`Suffix_2`' values to convert the position to WordCode. If the position is greater than $295,040$ and less than or equal to $141,527,936$, then it is of type WC3. If the position is greater than $141,527,936$ and less than or equal to $282,760,832$, then it is of type WC4. For types WC3 and WC4, the WordCode is computed in the same manner as for type WC2; however, the total number of '`Suffix_2`' in WC3 and '`Suffix_1`' in WC4 is $16,299$ (total number of two-byte codes, as specified in Table 2.3). If the position is greater than $282,760,832$ and less than or equal to $67,918,974,050$, then it is of type WC5. Finally, the method returns the WordCode code for the position.

2.4.2.2 WordCode-to-position (WC2P) algorithm

Algorithm 4 is the WordCode-to-position (WC2P) algorithm that is used to convert the WordCode code to the position of the array in the space-optimised WordTrie. The '`getSize(WC)`' method is used to obtain the size of the WordCode '`WC`' in bytes, and the '`getPrefix(WC)`' method is used to obtain the prefix part from the WordCode. If the WordCode is of type WC1, then the '`getSuffix(WC)`' method is used to obtain the suffix part from the WordCode. If the WordCode is of type WC2, WC3, WC4 or WC5, then the '`getSuffix_1(WC)`' method is used to obtain the first suffix part from the WordCode, and the '`getSuffix_2(WC)`' method is used to obtain the second suffix part from the WordCode. The '`getPIndex(Code)`' method is used to obtain the prefix index of '`Code`' from Table 2.1. The '`get1BIndex(Code)`' method obtains the one-byte suffix index of '`Code`' from Table 2.2. The '`get2BIndex(Code)`' method is used to obtain the two-byte suffix index of '`Code`' from Table 2.2. If the WordCode size is two bytes, then it is of type WC1. The index of the suffix part provides the position of the array for type WC1. If the WordCode size is three bytes, then it is of type WC2. If the WordCode size is four bytes and the '`Suffix_1`' size is one byte, then it is of type WC3. If the WordCode size is four bytes and the '`Suffix_1`' size is two bytes, then it is of type WC4. If the WordCode size is five bytes, then it is of type WC5. The position for the WC2, WC3, WC4 and WC5 types is computed as the sum of the number of codes at the previous level and the product of '`Prefix`', '`Suffix_1`' and '`Suffix_2`'. Finally, the method returns the position for the WordCode code.

Algorithm 4: WordCode-to-Position (WC2P) algorithm

Input: WordCode WC.
Output: Position P.

```
1  begin
2      size = getSize(WC)
3      if size == 2 then
4          P = get1BIndex(getSuffix(WC))
5      end
6      if size == 3 then
7          Prefix = getPIndex(getPrefix(WC)) − 1
8          Suffix_1 = get1BIndex(getSuffix_1(WC))
9          Suffix_2 = get1BIndex(getSuffix_2(WC))
10         P = 128 + (Prefix ∗ Suffix_1 ∗ Suffix_2)
11     end
12     if size == 4 then
13         Prefix = getPIndex(getPrefix(WC)) − 1
14         if getSize(getSuffix_1(WC)) == 1 then
15             Suffix_1 = get1BIndex(getSuffix_1(WC))
16             Suffix_2 = get2BIndex(getSuffix_2(WC))
17             P = 295040 + (Prefix ∗ Suffix_1 ∗ Suffix_2)
18         end
19         else
20             Suffix_1 = get2BIndex(getSuffix_1(WC))
21             Suffix_2 = get1BIndex(getSuffix_2(WC))
22             P = 141527936 + (Prefix ∗ Suffix_1 ∗ Suffix_2)
23         end
24     end
25     if size == 5 then
26         Prefix = getPIndex(getPrefix(WC)) − 1
27         Suffix_1 = get2BIndex(getSuffix_1(WC))
28         Suffix_2 = get2BIndex(getSuffix_2(WC))
29         P = 282760832 + (Prefix ∗ Suffix_1 ∗ Suffix_2)
30     end
31     return(P)
32 end
```

2.4.2.3 Traversing the space-optimised WordTrie to find the WordCode associated with a word

To find the WordCode for a given word in the space-optimised WordTrie, the WordTrie must be traversed from the root. Consider the word '`The`', for which the code must be found in the space-optimised WordTrie. The first character of the word is '`T`'; the algorithm checks whether the root node has a child '`T`' and traverses from the root node

to node ‘T’. The next character is ‘h’; the algorithm checks whether node ‘T’ has a child node ‘h’ and traverses from node ‘T’ to node ‘h’. The last character is ‘e’; the algorithm checks whether node ‘h’ has a child node ‘e’ and traverses from node ‘h’ to node ‘e’. If a link from the last character node to the array exists, then the position in the array is retrieved. In this case, a link from the last character node ‘e’ to the array exists, and the position ‘4’ is returned. The position ‘4’ is converted to the WordCode ‘1-3’ using the P2WC algorithm.

2.4.2.4 Traversing the space-optimised WordTrie to find the word associated with a WordCode

To find the word for a given WordCode in the space-optimised WordTrie, the WordTrie must be traversed from the array. Consider the WordCode ‘1-3’, for which the word must be found in the space-optimised WordTrie. The WC2P algorithm is used to convert the WordCode to the position in the array. The WordCode ‘1-3’ is passed to the WC2P method, and position ‘4’ is returned by the method. The ‘4th’ position in the array contains ‘p_4’, which is the address of the trie node containing the last character of the word. The last character node is traversed to the root of the trie to fetch the word associated with the code. In this case, node ‘e’, which is in the address ‘p_4’, is visited first. Then, the algorithm looks for the parent node of ‘e’ and traverses to node ‘h’. Because the root node is not reached, the algorithm again looks for the parent node of ‘h’ and moves to node ‘T’. Because the root node is not reached, the algorithm again looks for the parent node of ‘T’ and moves to the root node. Once the root node is reached, the reverse of the path traversed provides the word associated with the code, and the word ‘The’ is returned.

2.5 Experimental results

2.5.1 WordCode code page implementation

Because the English language is used by 59% of all websites[7], the experimental setup and evaluation were performed for encoding text files with English text. A word is formed by a sequence of Unicode, irrespective of the language. This work can also be extended to other languages by adding the words of the other languages to the WordTrie and allocating the subsequent unassigned WordCode codes. During the encoding with

[7] Usage Statistics of Content Languages for Websites, https://w3techs.com/technologies/overview/content_language, Accessed on 25/05/2020.

WordCode described in Section 2.3.3.2, for the English language, the word characters are considered as the set of uppercase characters (UTF 97 to 122), lowercase characters (UTF 65 to 90) and numbers (UTF 48 to 57). For other languages, the corresponding UTF code can also be included. WordCode performs better for text-containing words only.

The complete word list to be used by the WordCode code page can be obtained by crawling the entire web; however, due to resource limitations, the crawl is limited to the websites Wikipedia[8] and DMOZ[9]. Most of the words used across the web are also found on Wikipedia and DMOZ. The Wikipedia and DMOZ webpages were parsed for continuous occurrences of word characters, i.e., uppercase characters ($A--Z$), lowercase characters ($a--z$) and numbers ($0--9$). Words with lengths ranging from 1 to 29, were obtained, representing English words, words from various domains and words used to represent HTML syntax. Because WordCode uses a minimum of two bytes to represent a word, the words with lengths of one and two were eliminated. Additionally, words that occurred less than 100 times in the overall crawl were eliminated.

Table 2.5 shows the number of words and their lengths generated by parsing the Wikipedia and DMOZ websites. A total of 3.1 million (3,190,612) words were generated and used to construct the WordTrie. Although these 3.1 million words contribute 28 million (28,440,857) characters in total, only 9.6 million (9,608,014) characters are required in the WordTrie because words with the same starting substring will share common nodes of the trie. This WordTrie is used by the WordCode code page for encoding text files. Both the time-optimised WordTrie and the space-optimised WordTrie contain the same set of words and will obtain the same WordCode for any given word and the same word for any given WordCode. The difference between the time-optimised WordTrie and space-optimised WordTrie is the method by which the WordTrie is stored and traversed. Space-optimised WordTrie takes less memory but requires more computations to obtain the word and WordCode for the given WordCode and word, respectively. Conversely, the time-optimised WordTrie requires more memory but fewer computations to obtain the word and WordCode for the given WordCode and word, respectively.

[8]Wikipedia, The Free Encyclopedia, https://en.wikipedia.org, Accessed on 08/24/2017.
[9]DMOZ- Open directory project, https://www.dmoz.org/, Accessed on 12/24/2016.

Table 2.5: Number of words generated

Word Length	Number of Words
3	614
4	3669
5	291886
6	428831
7	479778
8	471482
9	404327
10	327541
11	238361
12	169845
13	116623
14	81133
15	55708
16	37529
17	25692
18	17652
19	12399
20	8391
21	5880
22	4139
23	2849
24	1945
25	1498
26	1068
27	779
28	582
29	411

2.5.2 WordCode encoding evaluation

The WordCode encoding algorithm has been evaluated over files containing text data from the Gutenberg corpus, Canterbury corpus, large corpus, Calgary corpus and Silesia corpus. These corpora contains text files, database files, program files and web documents. These corpora contain benchmarks for testing lossless data compression algorithms.

Table 2.6 shows a comparison of the size of the test data files with Unicode encoding and WordCode encoding. Table 2.6 also includes the number of files taken from each corpus, along with the total file size of the existing files and the total file size after WordCode encoding. Because the total file size of the Gutenberg corpus is in the millions and

because the total size of all other corpora is in the thousands, Table 2.6 is reproduced in Table 2.7, with the total file size being normalised to 100 KB to facilitate a comparison of the results. Table 2.7 also contains the average file size being computed, where, for a 100 KB Unicode file, WordCode encoding takes only 80.1 KB, thereby saving 19.9 KB of memory. Figure 2.7 presents the normalised graph of the comparison of the total file size of the actual file with the total file size of the WordCode file.

Table 2.8 shows the compression ratio and space savings of WordCode encoding with respect to Unicode encoding on the test data. The compression ratio is computed as the ratio between the Unicode file size and that of the WordCode file size. On average, compressing a 100-MB file to 80.1 MB achieves a compression ratio of $100/80.1 = 1.24$, where the ratio of the Unicode file size to the WordCode file size is $1.24 : 1$. The space savings are obtained using one minus the ratio of the WordCode file size to the Unicode file size. On average, compressing a 100-MB file to 80.1 MB achieves a space savings of $1 - (80.1/100) = 0.199$, where the space savings of WordCode amounts to 19.9% compared to the actual Unicode.

If the request to find the WordCode for the word during encoding is successful and the WordCode is returned, then it is a word hit; otherwise, it is a word miss. Table 2.9 shows the comparison of the percentage of word hits and word misses on the WordTrie for the test data. Version one of the WordTrie achieves 95.95% word hits on average with the word list generated from the Wikipedia and DMOZ websites. Crawling the entire web, adding newly occurring words periodically to the WordTrie and updating the WordTrie on all WordCode-configured devices will further increase the word hit percentage.

Table 2.10 compares the proposed WordCode with the related work described in Section 2.2. Although the compression ratio is average compared to other work, WordCode is the only approach that is capable of encoding 67.9 billion words.

Table 2.6: File size comparison

Corpus	Gutenberg Corpus	Canterbury Corpus	Silesia Corpus	Calgary Corpus	Large Corpus
Number of Files	3034	8	2	12	2
Total File Size (KB)	1210086.402	1229.584	51651.149	1505.093	6520.792
Total WordCode File Size (KB)	969420.643	965.453	43481.437	1179.284	5211.648

Table 2.7: Normalised file size comparison

Corpus	Gutenberg Corpus	Canterbury Corpus	Silesia Corpus	Calgary Corpus	Large Corpus	Average
Total File Size (KB)	100	100	100	100	100	100
Total WordCode File Size (KB)	80.11	78.51	84.18	78.35	79.92	80.10

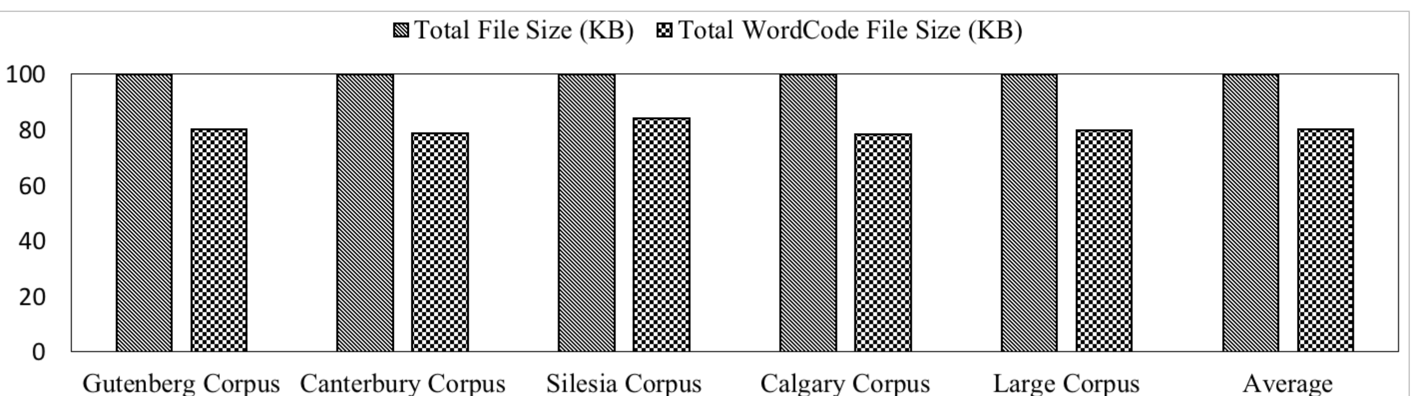


Figure 2.7: Normalised comparison of file size

Table 2.8: WordCode compression ratio and space savings

Corpus	**Gutenberg Corpus**	**Canterbury Corpus**	**Silesia Corpus**	**Calgary Corpus**	**Large Corpus**	**Average**
Compression Ratio	1.24	1.27	1.18	1.27	1.25	1.24
Space Savings	0.19	0.21	0.15	0.21	0.20	0.19

Table 2.9: WordTrie word hit and word miss

Corpus	**Gutenberg Corpus**	**Canterbury Corpus**	**Silesia Corpus**	**Calgary Corpus**	**Large Corpus**	**Average**
WordTrie Word Hit	96.01	93.73	78.78	85.89	93.18	95.95
WordTrie Word Miss	3.98	6.26	21.21	14.10	6.81	4.04

Table 2.10: Comparison of WordCode with related work

Method	**Language**	**No. of Words**	**Compression Ratio**
Azad et al. (2005)	English	5.24 Lakhs	2.15
Grabowski & Swacha (2010)	Language Independent	Words are stored in header	2.3
Sinaga et al. (2015)	Indonesian	16.3 Thousand	3.2
Kalajdzic et al. (2015)	Small Dictionary	1.11 Thousand	1.11
Proposed WordCode	Language Independent	67.9 Billion	1.24

2.6 Summary

This chapter presents an advancement of existing character-based encoding methods for text data (including linked data). The proposed system, i.e., "*WordCode*", can encode up to 67.9 billion words irrespective of language, each with a maximum size of five bytes. Additionally, the structure of the code page is upgraded from the regular table-based storage used in character encoding to a customised trie model called "*WordTrie*". WordCode is an optimised variable-length coding in which the most-frequent words receive small codes, and the size of the WordCode is always smaller than the size of the Unicode encoding. Traversing the WordTrie to find the word associated with a WordCode code is of $\theta(1)$, and traversing the WordTrie to find the WordCode code associated with a word of length 'n' is of $\theta(p \times n)$, where 'p' is the number of distinct characters in the WordTrie. Because the WordCode-encoded files are smaller than the Unicode-encoded files, machines handling text data with WordCode encoding achieve a reduced workload compared to machines processing text data with Unicode encoding.

Chapter 3

Partitioning

3.1 Introduction

The LDSF must handle linked data from all the SPARQL endpoints, and thus, it will be difficult for a single machine to store and process the entire linked data. Partitioning the linked data into multiple machines has two main advantages:

- Distributing the storage to multiple machines will improve the read and write time.
- Distributing the query processing load to multiple systems reduces the workload.

The query processing load can be reduced only by good partitioning techniques. No single design or architecture is a clear winner for the complex SPARQL workload (Aluç et al., 2014). In some applications, partitioning the linked data based on its contributors might be preferred. However, in an information retrieval framework, partitioning must be based on topics (i.e., different domains) because queries are often on a smaller set of topics. A good partitioning technique will use a minimum amount of data transmitted between nodes to execute the query. Substantial research on hash-based partitioning, cloud-based partitioning, and graph-based partitioning has been reported. However, these sophisticated partitioning algorithms have high preprocessing costs and do not partition on topics.

In an information retrieval framework, the queries are often bounded to "*the set of subjects belonging to the same type*", termed as a '*nexus*' in this thesis. For example, the *car nexus* contains the collection of subjects representing *cars*; similarly, the *medicine nexus* consists of the collection of subjects representing *medicine*. Since the chance of inter-nexus queries with *cars* and *medicine* is minimal, query processing can be improved by partitioning the linked data based on *nexus*. However, identifying the nexus is difficult, and many assisted clustering algorithms have been reported to group the linked data based on subjects. In practice, manually assisting the machine in clustering the subjects might be impossible considering the volume and diversity of data (Stevens et al., 2019).

In this chapter, an automated approach to cluster the linked data is proposed. The proposed methods cluster the linked data by identifying the nexus semantically using a novel clustering algorithm. The proposed algorithm identifies the core properties of the subjects and performs bilevel, *nexus*-based hierarchical agglomerative clustering to partition the linked data. The proposed algorithm does not depend on training data or expert assistance and can partition the schema-less linked data efficiently. The proposed clustering technique partitions the linked data with a precision of 98.7% on the gold standard dataset.

The remainder of this chapter is organised as follows. Section 3.2 discusses related work. Section 3.3 provides a detailed description of the proposed cluster-based partitioning technique. Section 3.4 presents the experimental results. Finally, Section 3.5 summarises the work.

3.2 Related works

Extensive studies of partitioning linked data have been conducted by Özsu (2016), Ma et al. (2016), Pan et al. (2018) and Wylot et al. (2018), and these works have highlighted three main categories of partitioning of linked data:

1. Graph partitioning
2. Hash partitioning
3. Cloud partitioning

3.2.1 Graph partitioning

Graph partitioning (Huang et al., 2011; Galárraga et al., 2014) considers the linked data as a graph and partitions it into subgraphs. In linked data, the edges for subjects belonging to the same topic will be minimised, and the edges connecting different topics are maximised. For example, consider the subjects '`IBM Laptop 1`' and '`IBM Laptop 2`'; there will be fewer edges connecting these two compared with connections between these subjects and other domains, such as '`Processor`' and '`Storage`'. The query for the '`IBM Laptop`' will require processing across multiple nodes. These types of partitioning incur high communication costs during query execution, as the data need to be executed and transferred from multiple nodes.

3.2.2 Hash partitioning

Hash partitioning (Lee & Liu, 2013; Harbi et al., 2016) methods distribute triples among the partitions by hashing the subject URI. The drawback of hash partitioning is that it does not consider the topic or semantic relatedness while partitioning. The subjects that belong to the same topic might not be placed in the same partition, resulting in a considerable amount of data being shipped between nodes for query processing. The popular triplestore Virtuoso Cluster[1] partitions the linked data using a hashing technique.

3.2.3 Cloud partitioning

Cloud-based approaches (Papailiou et al., 2014) employ triple pattern-based join processing such as MapReduce on existing cloud computing platforms, such as Hadoop[2] or Cassandra[3], to store the RDF graph. These approaches enjoy the benefits offered by cloud platforms, such as high scalability and fault-tolerance, but suffer from lower performance because it is difficult to adapt MapReduce to graph computation.

These proposed partitioning approaches are more suitable for single-endpoint linked data. For an information retrieval framework handling linked data from multiple endpoints, clustering the linked data and partitioning based on the clusters might be a more appropriate method. Section 3.2.4 discusses existing approaches to cluster linked data.

3.2.4 Clustering linked data

Mirizzi et al. (2010) proposed clustering based on the similarity between two subjects. The similarity is computed based on the number of webpages containing the value of both subjects '`rdfs:label`'. Consider two professors, '`Prof.ABC`' and '`Prof.XYZ`', from '`Stanford University`' and '`Princeton University`', respectively. Although '`Prof.ABC`' and '`Prof.XYZ`' belong to the same category (Professor), if no webpages contain the names of both professors, then they would be mismarked as '*not similar*'. '`Prof.ABC`' and '`Stanford University`' belong to different categories, i.e., '`Professor`' and '`University`'. If a webpage contains both of these labels, they would be incorrectly marked as relevant.

[1]http://docs.openlinksw.com/virtuoso/virtuosofaq10/
[2]http://hadoop.apache.org
[3]http://cassandra.apache.org

Nentwig et al. (2016, 2017) proposed a user knowledge-based clustering technique that takes the domain knowledge about the subjects and a function to determine the similarity between the subjects as an input from the user to the cluster. Ferrara et al. (2015) proposed a dimensional clustering technique to cluster the subjects; however, the technique's drawback is that the dimensions of the data need to be manually defined.

Gong et al. (2018) studied different approaches for lexical-based clustering of properties without weighing the semantic value of the linked data. However, in the proposed technique, the linked data are clustered based on their semantic relatedness rather than their lexical relatedness.

3.3 Proposed cluster-based partitioning

Cluster-based partitioning of linked data subjects is proposed. The term *nexus* is defined as "*the set of subjects belonging to the same type*" in the linked data. The method of identifying *nexus* of the subjects using the core predicates is discussed in Section 3.3.1. The bilevel, *nexus*-based hierarchical agglomerative clustering of linked data is narrated in Section 3.3.2. Section 3.3.3 outlines the *nexus* clustering algorithm.

3.3.1 Core and common predicates

The predicates are first grouped into common predicates and core predicates.

1. Common predicates are those that frequently occur in common across subjects and do not contribute to differentiating the subjects belonging to different clusters.

2. Core predicates are those other than the common predicates that uniquely identify the nexus of the subject.

Figure 3.1 represents the sample classification of predicates as core predicates and common predicates for the two film subjects '*Titanic*' and '*Avatar*' and the two actor subjects '*Leonard DiCaprio*' and '*Zoe Saldana*'. In Figure 3.1, title, abstract, type, label and images are some of the common predicates that occur in both films and actors. In Figure 3.1, the core predicates of the film are imdbID, producer, starring, budget, runtime, language, etc., and the core predicates of the actors are awards, name, imdbID, starring, etc. The method of finding the core and common predicates is discussed in Section 3.3.1.

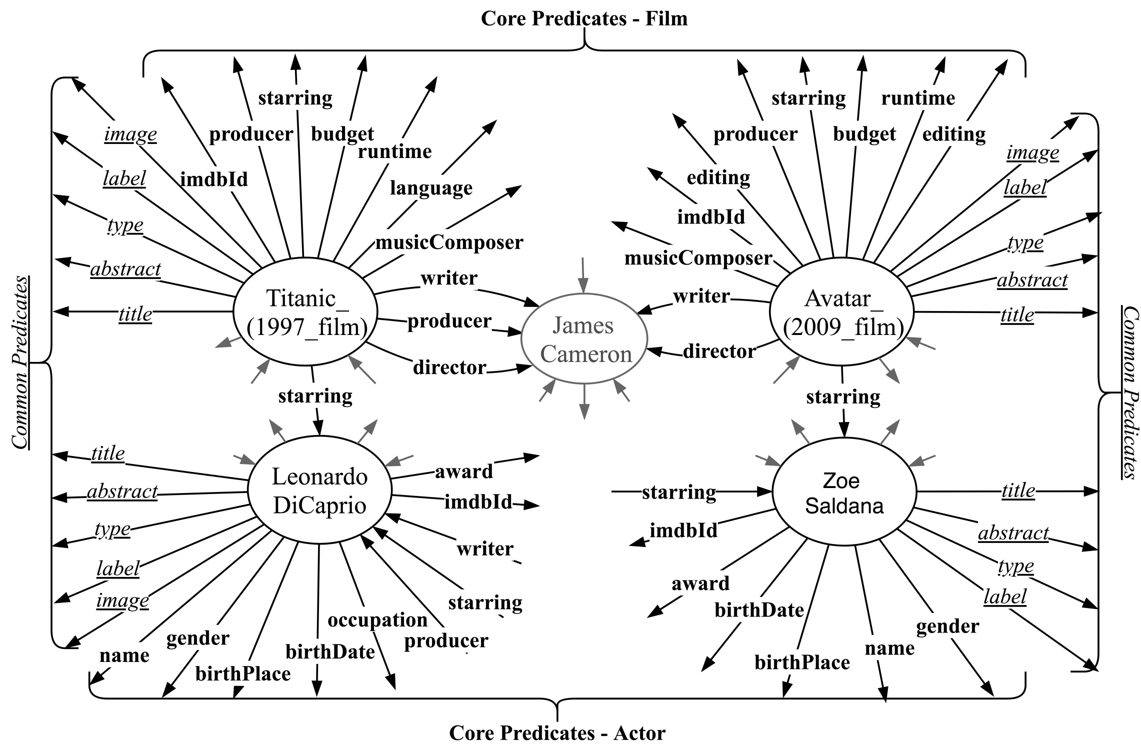


Figure 3.1: Sample subjects with core and common predicates

3.3.1.1 Finding core and common predicates

The core and common properties can be found by plotting the overall predicates and their count using the following SPARQL query

```
SELECT
  DISTINCT(?Predicate), ( COUNT(?Predicate) AS ?PCount )
WHERE
  {
     ?Subject ?Predicate ?Object.
  }
ORDER BY DESC(COUNT(?Predicate))
```

This SPARQL query generates the count of all predicates sorted in descending order. Figure 3.2 shows the structure of the graph plotted for the above SPARQL query executed in DBpedia SPARQL endpoint[4]. Since the core predicates are those that occur

[4]http://dbpedia.org/sparql, Accessed on 20/12/2019

only in the subjects belonging to the nexus, the count of core predicates will be low, and the common predicates occurring across the nexus will have a high count. As a result, the core and common predicates can be differentiated using the slope in the graph from α to β. From `1` to α are the common predicates, and from β are the core predicates. There will be semi-common predicates from α to β, such as '`foaf:name`', which will occur for a small group of nexuses.

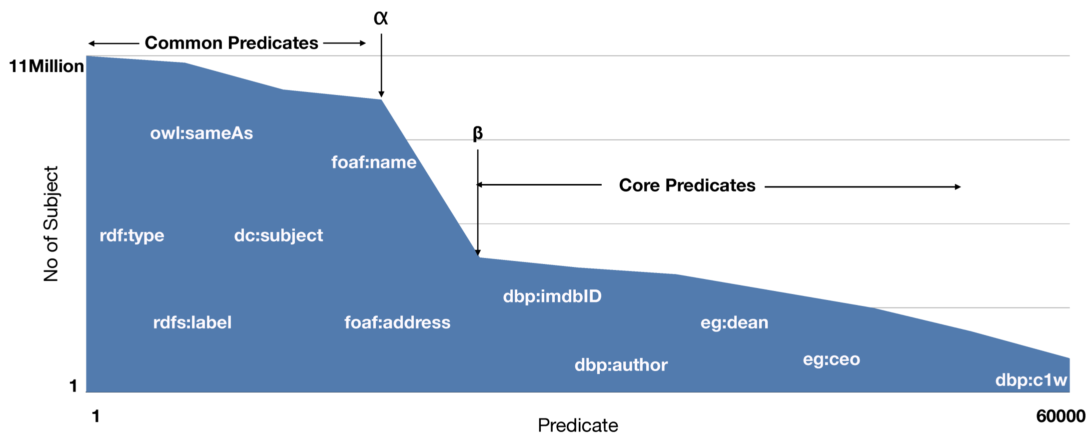


Figure 3.2: Slope from common to core predicates

3.3.2 Bilevel *nexus*-based hierarchical agglomerative clustering

Consider two subjects, '`Harvard University`' and '`Stanford University`', as shown in Figure 3.3. Here, predicates '`Director`', '`Dean Administration`' and '`has Department`' are the primary core predicates, and the predicates '`Resea-rch Interest`', '`Publication`', '`Office`', '`Professor`', '`Student`' and '`Organises`' are the secondary core predicates. '`Director`' is an incoming predicate, and '`has Department`' is an outgoing predicate. The subjects are considered similar only if the primary- and secondary-level core predicates are similar. First the similarity of the primary core predicates is checked, and the similarity of the secondary core predicates is checked only if the primary core predicates are similar.

The incoming predicates are labelled as '`I`' predicates, and the outgoing predicates are labelled as '`O`' predicates. In Figure 3.3, '`Director`' and '`Dean Administra-tion`' are '`I`' predicates, while '`has Department`' is the '`O`' predicate. The second-level predicates are labelled '`II`', '`IO`', '`OI`' and '`OO`', as in Table 3.1, where '`I`' and '`O`' represent the incoming and outgoing predicates, respectively.

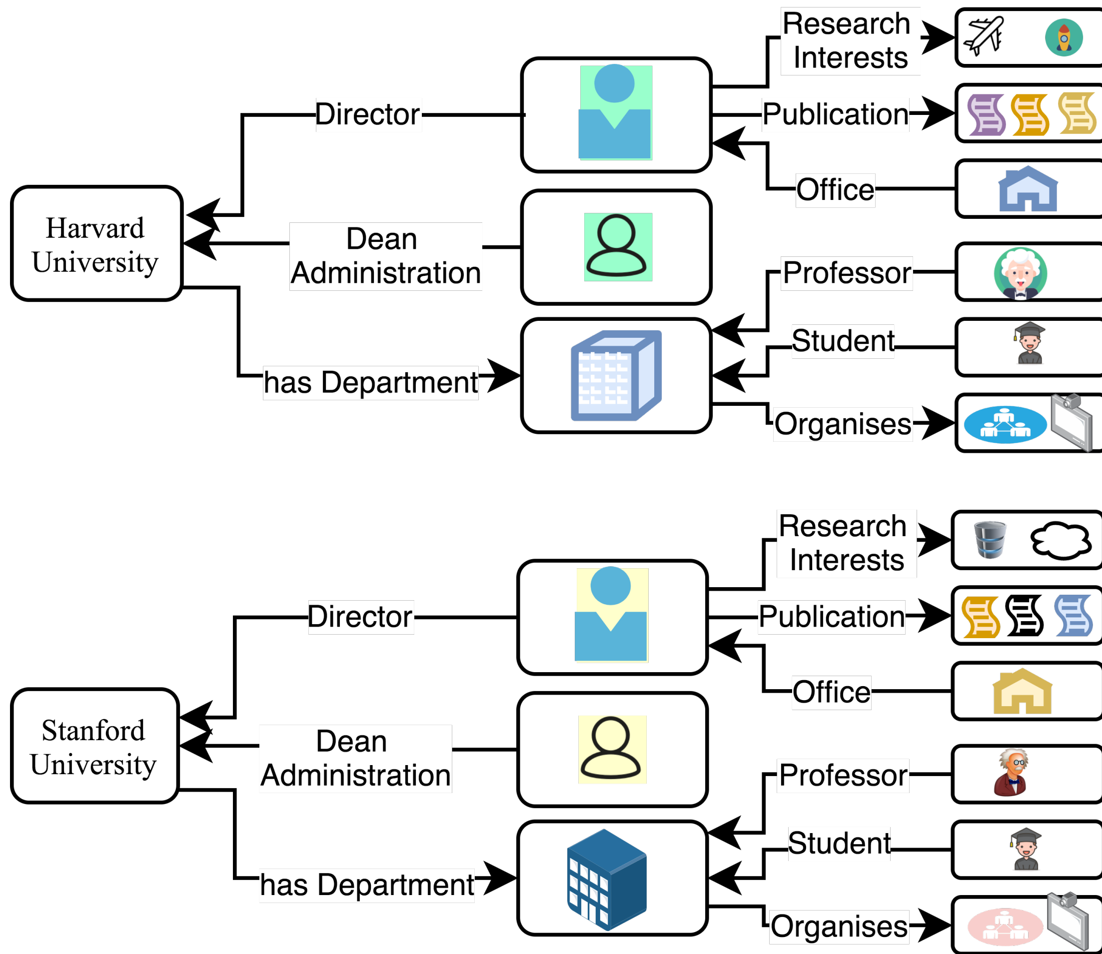


Figure 3.3: Example clustering

In Figure 3.3, '`Research Interest`' and '`Publication`' are the '`IO`' predicates; '`Office`' is the '`II`' predicate; '`Professor`' and '`Student`' are the '`OI`' predicates; and '`Organises`' is the '`OO`' predicate.

Table 3.1: Notation of secondary-level predicates

Notation	**First level**	**Second level**
II	Incoming	Incoming
IO	Incoming	Outgoing
OI	Outgoing	Incoming
OO	Outgoing	Outgoing

3.3.3 Nexus clustering algorithm

The Algorithm 5 is the *nexus* clustering algorithm. The inputs to the *nexus* clustering algorithm are the linked data graph (`G`) and the expected number of clusters (`n`). The set of subjects (S_a) and predicates (P_a) are extracted from the linked data graph (`G`). The common predicates (CP_p) and semi-common predicates (SP_p) are determined using the procedure discussed in Section 3.3.1. Predicates that occur only once (O_p) are eliminated, as they have no use in finding similarity.

Algorithm 5: Nexus clustering algorithm

```
   Input: RDF graph G;
1  n // Number of clusters
   Output: C_x = {C_1, C_2, ..., C_m} // Set of clusters
2  S_a = {S_1, S_2, ..., S_p} // Set of all subjects
3  P_a = {P_1, P_2, ..., P_q} // Set of all predicates
4  O_p = {O_1, O_2, ..., O_r} // Set of predicates with COUNT=1
5  CP_p = {CP_1, CP_2, ..., CP_s} // Set of common predicates
6  SP_p = {SP_1, SP_2, ..., SP_t} // Set of semi-common predicates
7  P_y = {{P_a} − {CP_p + SP_p + O_p}} // ORDER BY ASC on COUNT(P_s)
8  C_x = S_a // Initially, each subject is a cluster
9  γ = 100; // Similarity score
10 while ((γ > 0) && (COUNT(C_x) > n)) do
11     for (p_i : P_y) do
12         C_j = getClusters(p_i, C_x);
13         for all pairs of clusters (c_x, c_y) from C_j do
14             P_pcx = getPrimaryPredicates(c_x);
15             P_pcy = getPrimaryPredicates(c_y);
16             PrimarySimilarityScore = getPrimarySimilarity(P_pcx, P_pcy);
17             if PrimarySimilarityScore >= γ then
18                 P_scx = getSecondaryPredicatesSet(c_x);
19                 P_scy = getSecondaryPredicatesSet(c_y);
20                 SecondarySimilarityScore =
                    getSecondarySimilarity(P_scx, P_scy);
21                 if SecondarySimilarityScore >= γ then
22                     C_x = MergeCluster(c_x, c_y);
23             end
24         end
25     end
26     γ − −;
27 end
28 return(C_x);
```

The core predicate is the difference between all predicates (P_a) and common predicates (CP_p), semicommon predicates (SP_p) and predicates occurring once (O_p). Because the number of core predicates is much smaller than the number of subjects, the cluster similarity search is constrained with the subjects corresponding to every predicate[5]. The method of clustering using the core predicates minimises the comparison space and reduces false positives while clustering. Each subject is initially assigned as a cluster, and the loop continues while the similarity score is greater than zero and the cluster count is greater than (n).

For every core predicate p_i, all clusters C_j containing the predicate p_i are obtained using the '*getClusters*' method. For every pair of clusters c_x and c_y in C_j, the similarity in the first-level predicates is found using the '*getPrimarySimilarity*' method. The '*getPrimarySimilarity*' method returns the number of `I` and `O` predicates that are similar between any two clusters at the first level. If the primary-level predicates are found to be greater than or equal to γ percentage, then the secondary-level predicate similarity is found between the clusters using the '*getSecondarySimilarity*' method. The '*getSecondaryPredicatesSet*' method is used to obtain the '`II`', '`IO`', '`OI`' and '`OO`' predicates for the given clusters. The '*getSecondarySimilarity*' method returns the number of similar predicates '`II`', '`IO`', '`OI`' and '`OO`' between the given clusters. If the secondary-level similarity score is greater than or equal to γ percentage, then the clusters c_x and c_y are merged using the '*MergeCluster*' method. Finally, the algorithm returns the clusters of subjects.

3.4 Experimental setup and evaluation

3.4.1 Experimental setup

LDSF was developed using Java with the support of the Apache Jena[6] package. The SPARQL endpoint availability, performance, discoverability and interoperability were collected from SPARQLES (Vandenbussche et al., 2017). Initially, 776 SPARQL endpoints were found across the Internet[7,8]. Some of the SPARQL endpoints, such as

[5] In DBpedia, the no. of predicates is 60,649 & the no. of subjects is 10,721,963; Accessed on 11 November 2018.

[6] https://jena.apache.org

[7] https://www.w3.org/wiki/SparqlEndpoints, Accessed on 10 March 2018.

[8] http://sparqles.ai.wu.ac.at/api/endpoint/list

"*http://upenn.eagle-i.net/sparqler/sparql*" and "*https://eagle-i.itmat.upenn.edu/sparqler/sparql*", contain the same data. These SPARQL endpoints were queried for their availability, and only 275 SPARQL endpoints containing distinct data were found[9]. Table 3.2 shows the total amount of data available from the 275 SPARQL endpoints.

Table 3.2: Data available from the SPARQL endpoints

Type	**Total count**
Triples	26.5 billion
Subjects	0.4 billion
Predicates	1.7 billion
Object as values	12.6 billion

Although the final goal of LDSF is to allow users to access data from 275 SPARQL endpoints, due to resource limitations, a test dataset was created to evaluate the cluster based partitioning. 77,000 subjects were sampled from DBpedia belonging to eight categories: Film, Artist, Director, Producer, Writer, Song, Musician and Singer. The bilevel nodes and edges of the 77,000 subjects were retrieved from the 275 SPARQL endpoints. The '`skos:exactMatch`' property is used to obtain the different URIs used by the other SPARQL endpoints used to represent the same data (Halpin et al., 2010). The dataset is a gold standard for cluster evaluation because the sampled categories are closely related and share some properties.

3.4.2 Evaluation metrics

Figure 3.4 is the sample clustering of the eight classes using properties. Here, all clusters share common properties such as `rdf:type` and `owl:sameAs`. The semi-common properties are shared among certain clusters. In Figure 3.4, the properties of '`Person`' such as `foaf:name` are semi-common as they are shared among the Artist, Director, Producer, Writer, Musician and Singer clusters.

Although the proposed clustering algorithm computes the similarity based on the occurrence of the core properties, called the core occurrence ratio (COR), the results are also tested for the normal occurrence ratio (OR) and matching ratio (MR).

[9] Queried on 10 March 2018.

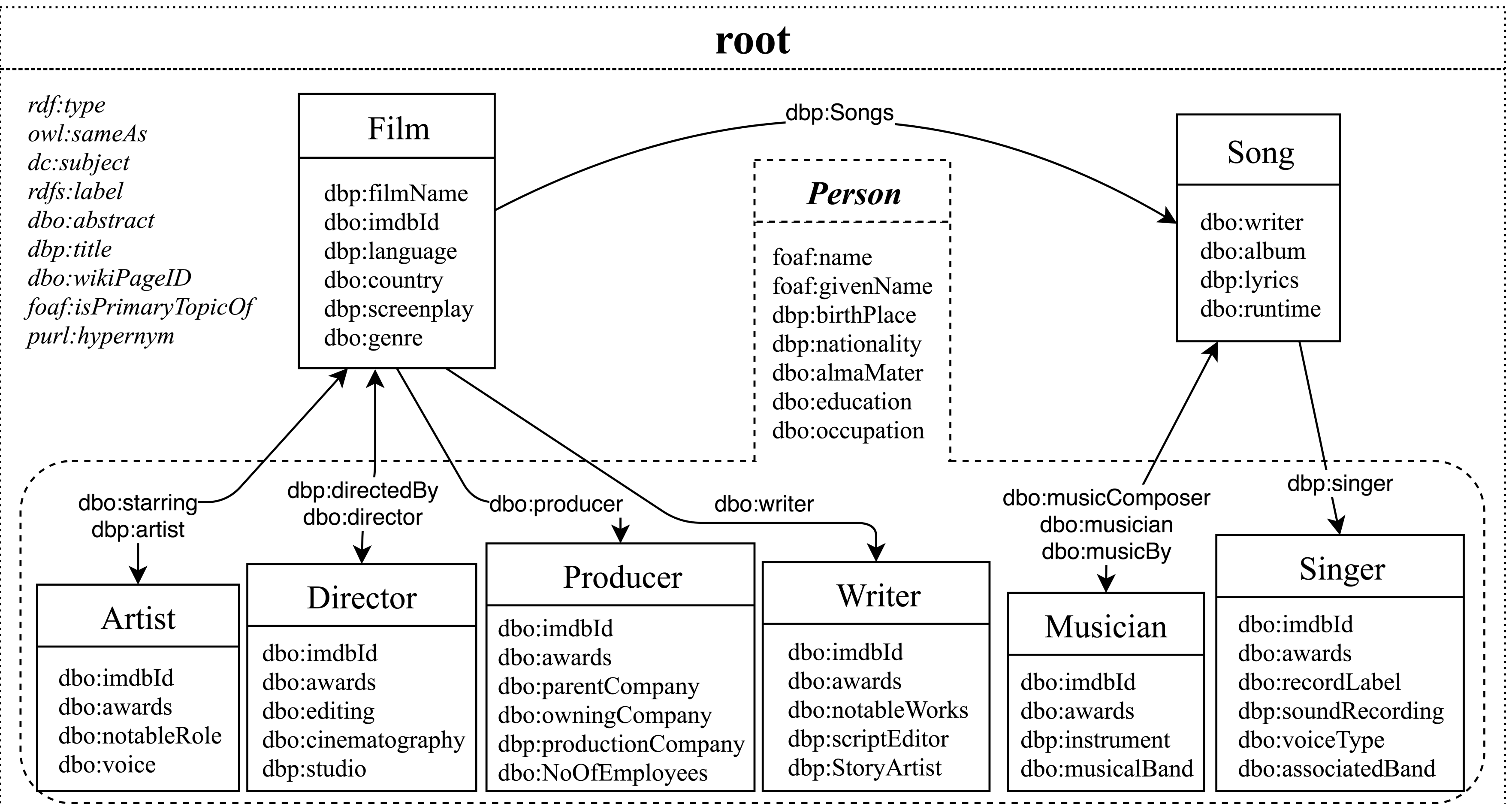


Figure 3.4: Clustering sample

1. **Matching ratio (MR):** The matching ratio is the ratio of the number of distinct predicates matching between the clusters to the total number of distinct predicates in the clusters. Let 'MR_t' be the total number of distinct predicates in the clusters c_x and c_y, and let 'MR_m' be the number of distinct predicates matching between the clusters c_x and c_y. Then, the matching ratio '***MR***' similarity between the clusters c_x and c_y is calculated as the ratio of 'MR_m' to 'MR_t' as in Formula 3.1.

$$Matching\ ratio\ (MR) = \frac{MR_m}{MR_t} \tag{3.1}$$

2. **Occurrence ratio (OR):** The occurrence ratio is the ratio of the total number of occurrences of the predicates between the clusters to the total occurrences of the predicates in the clusters. Let 'OR_t' be the total occurrences of the predicates in the clusters c_x and c_y, and let 'OR_m' be the total number of occurrences of the predicates between the clusters c_x and c_y. Then, the occurrence ratio '***OR***' similarity between the clusters c_x and c_y is calculated as the ratio of 'OR_m' to 'OR_t' as in Formula 3.2.

$$Occurrence\ ratio\ (OR) = \frac{OR_m}{OR_t} \tag{3.2}$$

3. **Core occurrence ratio(COR):** The core occurrence ratio is the occurrence ratio calculated using the core predicates. This COR is used to calculate the similarity score in the proposed nexus clustering algorithm. Let 'COR_t' be the total number of occurrences of the core predicates in the clusters c_x and c_y, and let 'COR_m' be the total number of occurrences of the core predicates between the clusters c_x and c_y. Then, the core occurrence ratio '***COR***' similarity between the clusters c_x and c_y is calculated as the ratio of 'COR_m' to 'COR_t' as in Formula 3.3.

$$Core\ occurrence\ ratio\ (COR) = \frac{COR_m}{COR_t} \tag{3.3}$$

3.4.3 Experimental results

Table 3.3 shows the clustering precision calculated using MR, OR and COR for the primary incoming and outgoing predicates (I, O) and for the secondary incoming and outgoing predicates (IO, OI, II). No secondary outgoing outgoing (OO) predicates were found in any of the test data. The precision was calculated using Formula 3.4.

$$Precision = \frac{True\ Positive}{True\ Positive + False\ Positive} \tag{3.4}$$

Table 3.3: Clustering precision at various levels

Data	I			O			IO			OI			II		
	MR	**OR**	**COR**	**MR**	**OR**	**COR**	**MR**	**OR**	**COR**	**MR**	**OR**	**COR**	**MR**	**OR**	**COR**
Film	0.446	0.536	0.877	0.586	0.529	0.920	NULL	NULL	NULL	NULL	NULL	NULL	0.532	0.530	0.914
Artist	0.428	0.512	0.831	0.657	0.530	0.832	0.487	0.526	0.712	0.790	0.536	0.857	0.361	0.500	0.626
Director	0.442	0.503	0.823	0.612	0.527	0.794	0.423	0.524	0.696	0.704	0.528	0.767	0.305	0.499	0.563
Producer	0.476	0.482	0.827	0.611	0.519	0.774	0.411	0.509	0.728	0.684	0.526	0.760	0.316	0.501	0.607
Writer	0.434	0.510	0.859	0.630	0.532	0.879	0.484	0.526	0.750	0.766	0.533	0.879	0.337	0.503	0.721
Song	0.727	0.534	0.991	0.789	0.526	0.989	0.552	0.538	0.990	NULL	NULL	NULL	0.439	0.520	0.984
Musician	0.468	0.498	0.811	0.648	0.526	0.832	0.443	0.524	0.789	0.707	0.531	0.822	0.353	0.505	0.733
Singer	0.431	0.508	0.819	0.549	0.527	0.826	0.474	0.525	0.781	0.732	0.532	0.841	0.339	0.508	0.743
Average	0.481	0.510	0.855	0.635	0.527	0.856	0.468	0.525	0.778	0.730	0.531	0.821	0.373	0.508	0.737

Figure 3.5 shows the line graph plotted for the mean clustering precision at various levels. The core occurrence ratio (COR) achieves higher precision than the matching ratio (MR) or occurrence ratio (OR).

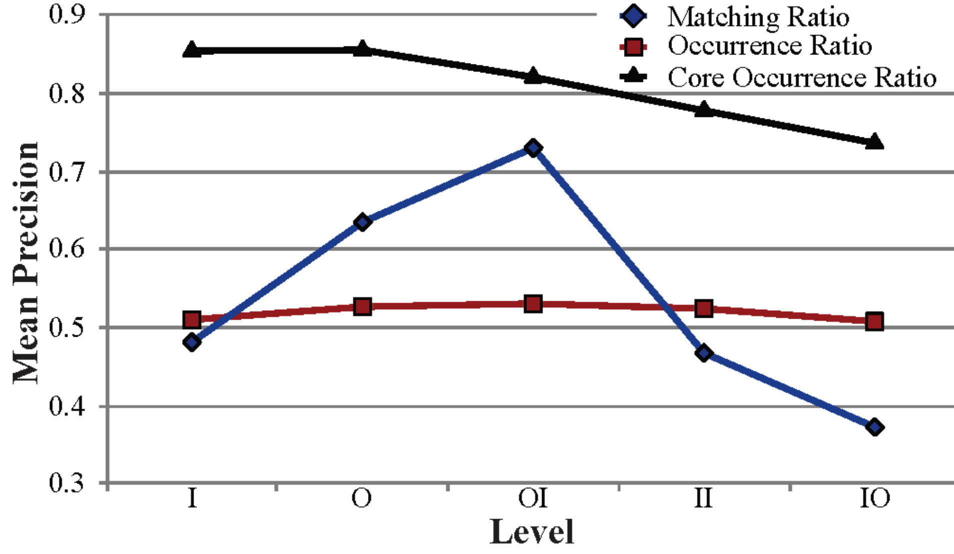


Figure 3.5: Mean precision at different levels

In LDSF, the COR similarity was used to determine the cluster of each subject. Table 3.4 shows the overall clustering precision, recall and F-measure computed using the COR similarity. The recall and F-measure were calculated using formulas 3.5 and 3.6, respectively.

$$Recall = \frac{True\ Positive}{True\ Positive + False\ Negative} \tag{3.5}$$

$$F - measure = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{3.6}$$

Although methods such as dimensional clustering (Nentwig et al., 2016) and holistic clustering (Lee et al., 2014) achieve high precisions of 0.993 and 0.71, respectively, they require the manual definition of the training data from the user to achieve this high clustering accuracy. In practice, manually defining training data for linked open data is impractical due to the huge data volumes. In the study conducted by Gong et al. (2018) on seven different automatic clustering techniques using 13 different measures, the highest F-measure (Case 1) of 0.432 was achieved by single-linkage clustering in

Table 3.4: Nexus clustering evaluation using the core occurrence ratio

Data	**Precision**	**Recall**	**F-measure**
Film	0.974	0.896	0.907
Artist	0.959	0.844	0.898
Director	1.000	0.849	0.945
Producer	0.824	0.846	0.835
Writer	0.763	0.846	0.802
Song	0.902	0.991	0.879
Musician	1.000	0.858	0.995
Singer	0.988	0.863	0.921
Average	**0.924**	**0.875**	**0.899**

the N-gram approach. Although the highest precision (Case 2) was achieved by complete linkage clustering on the overlap of the property values and N-gram, this method achieved a very low recall of 0.039, with an F-measure of 0.075. Table 3.5 shows the performance comparison of the proposed nexus clustering evaluated with the dataset used by Gong et al. (2018). The proposed nexus-based clustering achieves an F-measure of 0.929, with a precision and recall of 0.987 and 0.877 respectively, making it better than current automated clustering approaches.

Table 3.5: Cluster comparison

Clustering	**Precision**	**Recall**	**F-Measure**
Case 1: Single-linkage clustering in the N-gram approach	0.534	0.363	0.432
Case 2: Complete linkage clustering on overlap of property values and N-gram	0.971	0.039	0.075
Proposed Nexus clustering	0.987	0.877	0.929

3.5 Summary

This chapter presents a cluster-based partitioning of linked data subjects that is suitable for an information retrieval framework. The term ‘*nexus*’ is redefined according to linked data as the “*set of subjects belonging to the same type*”. The proposed method clusters the linked data using a novel *nexus* clustering algorithm. The key idea is to identify the two-level core predicates of the subjects and use them for clustering. The main benefit of our method is that it is automated and can cluster linked data from multiple SPARQL endpoints without the support of training data. The proposed algorithm partitions the gold standard test data with a precision of 98.7% and recall of 87.7%.

Chapter 4

Indexing

4.1 Introduction

The performance of information retrieval is highly dependent on the physical organisation and indexing of the linked data. Indexing is a way to optimise the speed and performance of querying the linked data by minimising the number of disk access cycles. Without an index, the system must scan the entire RDF graph, which requires very large amounts of computation resources and time. An inverted index is a popular indexing technique used by web search engines. The inverted index maps the contents of the web document, such as words or numbers, to the corresponding webpage in a table.

Substantial research on permutation-based indexing of linked data has been reported by surveys conducted by Faye et al. (2012), Wylot et al. (2018), Kaoudi & Manolescu (2015) and Ali et al. (2020). These exhaustive indexes use the permutations of the subject-predicate-object of the triples and are built on the hypothesis that the queries are basic graph pattern (BGPs). However, the schema-free structure of the linked data allows expressive SPARQL queries to be executed. These expressive SPARQL queries are complex in terms of representing graph patterns. A permutation-based index performs better only on BGP queries in a single SPARQL endpoint. However, in an information retrieval framework, because of the necessity of indexing the linked data from all the SPARQL endpoints, permutation-based indexing will outrun storage.

In this chapter, indexing based on an inverted index and a hybrid data structure named *'trist'* is proposed. The structure of trist is inspired by the structure of WordTrie, which is discussed in Chapter 2, Section 2.4. *Trist* is formed by linking a tree and a doubly linked list. The proposed indexing method benefits from the fact that an inverted index facilitates a relatively fast search. In contrast to the reported permutation-based indexing methods, the *trist*-based indexing method has a smaller storage size. The proposed *trist*-based indexing supports incremental indexing, i.e., the insertion and deletion of data do not require the complete rebuilding of the index. The time complexity for searching for a URI in the proposed *trist*-based indexing is $\theta(m)$, where 'm' is the length

of the URI.

The remainder of this chapter is organised as follows. Section 4.2 discusses the related work. Section 4.3 provides a detailed description of the proposed indexing technique. Section 4.4 presents the experimental results. Finally, Section 4.5 summarises the work.

4.2 Related work

Permutation-based indexing was initially proposed by Weiss et al. (2008), called Hexastore. Hexastore creates an index using all six possible combinations of the subject (S), predicate (P) and object (O) of the triples. The six combinations of Hexastore are SPO, OPS, SOP, OSP, PSO and POS, where S, P and O are the subject, predicate and object of the triples. Later, variations of this permutation-based indexing began to attract attention. The recent work on permutation-based indexing is summarised in Table 4.1. The S, P and O in Table 4.1 are the subject, predicate and object of the triples. The underscores used by Hu et al. and Oh et al. denote an unknown attribute passed by the query. In the five indexing models proposed by Chen et al., class (C) denotes the URI of the subject and object, and relation (R) denotes the URI of the predicate.

Table 4.1: Permutation based indexing of linked data

Index	**Author**	**Permutation**
Eight Indexes	Papailiou et al. (2014)	Six core index same as Hexastore, Two aggregate index on each of the core index.
Six Indexes	Hu et al. (2016)	SP_O, SO_P, PO_S, P_SO, O_SP, S_PO
Five Indexes	Chen et al. (2015)	C (Class), R (Relation), CR, RC, CRC
Four Indexes	Zeng et al. (2013)	SPO, OPS, PS, PO
Three Indexes	Xu et al. (2015), Punnoose et al. (2015)	SPO, POS, OSP
	Harbi et al. (2015)	P, PS, PO
One Index	Oh et al. (2015)	Singe index containing O_PS and S_PO

An extensive survey of the existing methods of indexing the linked data conducted by Faye et al. (2012), Kaoudi & Manolescu (2015), Wylot et al. (2018) and Ali et al. (2020) highlights the following challenges.

(a) The main drawback of permutation-based indexing is the large index size. The same data are stored in multiple patterns, causing substantial storage overhead.

(b) Some permutation-based indexing employs compression of the index files to reduce the index size. However, the objective of indexing of improving the access time is negated by the time required to decompress the compressed index files.

(c) With increasing amounts of linked data, it is essential not only to create but also to maintain and update the index to obtain the latest data. Updating and deleting the data elements from the permutation-based index is complex.

(d) Limiting the index to the BGP query using the permutation will not fulfil the expressive power of SPARQL querying over a dynamic structure of linked data.

Considering the limitations in the existing systems, a novel method to index linked data is proposed in Section 4.3.

4.3 Proposed indexing technique

A novel indexing scheme to index the linked data from multiple SPARQL endpoints is proposed. URIs are the main components of the subjects, predicates and objects of the linked data. The method of using dictionaries to reduce the recurrence of URIs is discussed by Singh et al. (2018). This chapter proposes indexing using a new data structure called ***trist*** by linking the tree and doubly linked list. The URI and values of the linked data are parsed and stored in the *URI trist* and *value trist*, respectively. The *trist* shares the common higher-level nodes, and each URI and value are given an ID, which is used by the inverted index to map the ID to the nodes containing the respective URIs and values in the RDF graph.

The structure of the *trist* data is illustrated in Section 4.3.1. The techniques for storing the URIs and values in the *trist* data structure as *URI trist* and *value trist* are discussed in Sections 4.3.2 and 4.3.3, respectively. The method of mapping the URI and value ID to the RDF graph using the inverted index is briefly described in Section 4.3.4. Section 4.3.5 outlines the procedure for storing and accessing the RDF graph.

4.3.1 Trist data structure

A new data structure named ***trist*** is proposed by linking the tree and the doubly linked list. Figure 4.1 shows the structure of the *trist* for the words '`http`', '`https`', '`html`', and '`head`'. Here, the words are split into characters and stored as nodes of the tree, where words with the same starting substring (prefix) share nodes of the tree branch starting from the root. In Figure 4.1, all the words share the common first node '`h`' as

all these words begin with the character 'h'. The nodes of the tree containing the last character of the words are called '*fruit nodes*'. In Figure 4.1, the nodes 'p', 's', 'l', and 'd' are the '*fruit nodes*'. A unique ID is allotted in the doubly linked list for each word inserted in the tree. The word 'http' is alloted the ID '1' in the list. The pointer from the '*fruit node*' to the list is called the '*list pointer*'. The pointer from the doubly linked list to the '*fruit node*' is called the '*tree pointer*'.

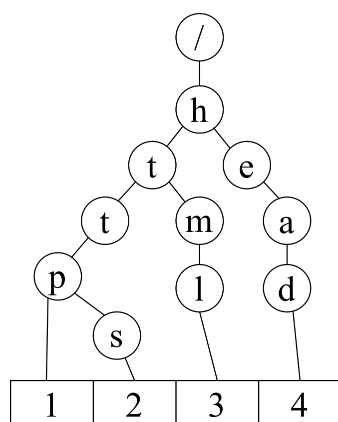


Figure 4.1: Trist structure

(a) **Insert new data in trist**: New data is inserted by adding the new word to the tree and allocating a new ID to the newly inserted word.

(b) **Delete existing data from trist**: A word is deleted by revoking the allocated ID for the word and removing the fragment of the word that is not shared with any other words.

4.3.2 URI trist

The *URI trist* is used to index the URIs used in the RDF graph. The URI of the RDF data follows the grammar

$$URI ::= protocol : //hostname/[path]^*/fragment$$

The URI is parsed to obtain the hostname, intermediate paths and fragments. The hostname and path are stored wholly as words; the fragment part of the URI is split into characters. The protocol is not stored in the *trist* as the protocol *http* or *https* do not alter the implication of the subject. The hostname is directly connected to the root

node. The intermediate paths are then connected to their corresponding hostnames. The fragment is character parsed, and the first character of the fragment is connected to the final path node. The last character node of the fragment from the *fruit node* is connected to the list using the list pointer. Consider Figure 4.2, where the two URIs "`http://dbpedia.org/ontology/city`" and "`http://dbpedia.org/ontology/campus`" share the hostname '`dbpedia.org`' and the intermediate path '`ontology`' in the *trist*. Additionally, the fragments '`city`' and '`campus`' start with the same character, '`c`', and share the character node, as in Figure 4.2. The *fruit node* of the fragment is linked to the list. The list has a unique ID allocated for every URI. In Figure 4.2, the pointer value '`0×5A`' is the unique ID for the URI "`http://dbpedia.org/ontology/city`".

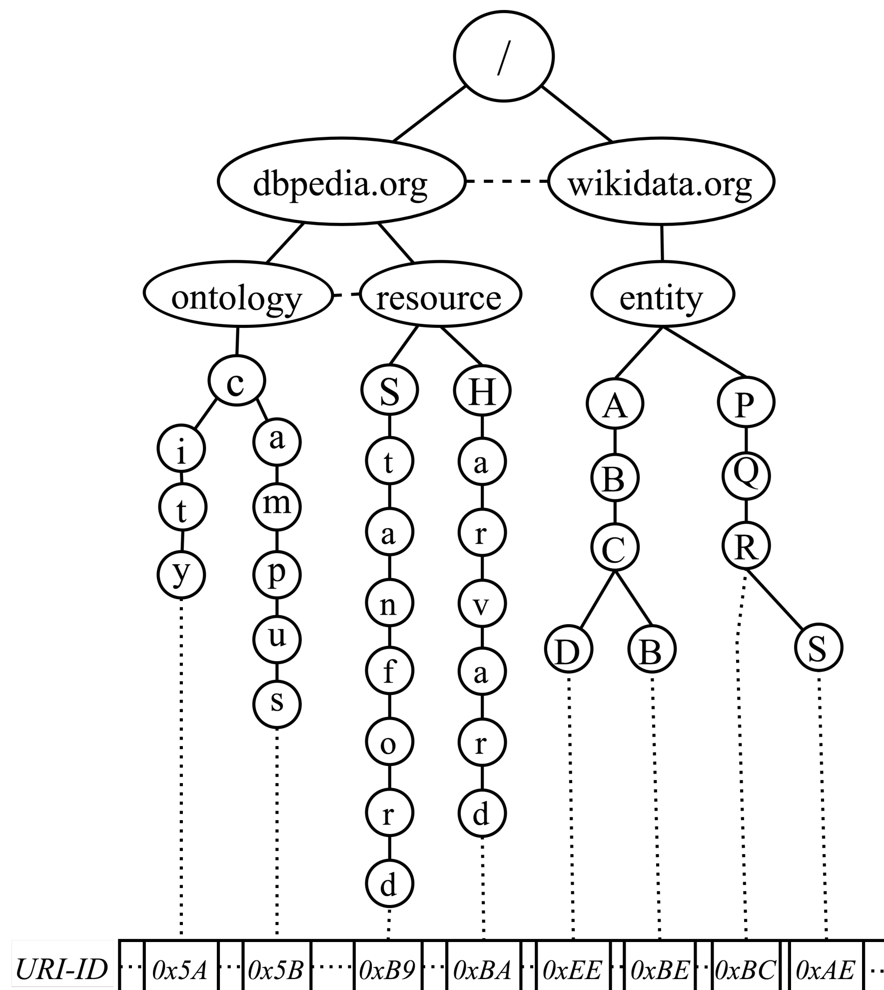


Figure 4.2: URI trist

4.3.3 Value trist

The *value trist* is used to index the objects of triples containing values. The values are grouped based on their data type in the *value trist* (V. Biron et al., 2004). Table 4.2 shows the grammar followed by some of the data types (Prud'hommeaux & Seaborne, 2007). Since the string values can contain a long string with tens and hundreds of words, they are split into unigram (1-gram) words using the stemming algorithm (Mayfield & McNamee, 2003). The stop words are removed from the string values because they do not contribute to information retrieval. The values are then parsed based on the grammar and stored in the *value trist* with the common higher-level nodes in the tree being shared.

Table 4.2: Object grammar

Data type	**Grammar**
integer	[0-9]+
float	[0-9]* [.][0-9]+
string_en	[a-z A-Z 0-9]+
date	[0-9][0-9][0-9][0-9][Jan-Dec][01-31]
time	[00-24][00-60][00-60]
date-time	[0-9][0-9][0-9][0-9][Jan-Dec][01-31][00-24][00-60][00-60]

Figure 4.3 is the sample structure of the *value trist*. In the *value trist*, the values are stored based on type. For example, the decimal value '`3.14`' is parsed based on character and stored under the float type. The value '`3.14`' is allocated the value ID *`0×D48`*. In Figure 4.3, the terms string-en, string-es and string-fr are the same notations used by RDF for representing English, Spanish and French strings, respectively. Consider the string "*California is a suburban city*" to be indexed using *value trist*. First, the stop words are removed from the sentence. The stopwords in the sentence are '*is*' and '*a*'. Then, the stemming algorithm is applied to obtain the unigram words from the sentence. The unigram words obtained from the string are '*California*', '*suburban*' and '*city*'. A unique ID is allocated for each of the unigram words stored in the *value trist*.

Consider the date-time value "*2017 April 8, 15:24:30*", as given in Figure 4.3. The first four levels in the *trist* represent the year, followed by the month, date, hour, minute and second. Finally, the *fruit node* containing the second value is linked to the list node with the unique value ID '*0×B55*'.

Figure 4.3: Value trist

4.3.4 Inverted index

The inverted index (Baeza-Yates & Ribeiro-Neto, 1999) is the most popular structure used by search engines. This index maps the content of the website to its location. In LDSF, the unique ID assigned to the URI and value by the *URI trist* and *value trist* is mapped to its location in the RDF graph using inverted index. Figure 4.4 shows the method of mapping the ID to the location. The inverted index contains only the mapping from the ID to its location in the RDF graph. The mapping of the URI or value to the ID in Figure 4.4 is shown for illustration and is actually stored by the *URI* and *value trist*. The URI "`http://dbpedia.org/ontology/city`" shortly written

as ‘dbo:city’ has the ID ‘0×5A’ assigned while constructing the *URI trist*. Since the RDF graph contains the URI ‘dbo:city’ in memory location ‘5’ and ‘11’, the ID ‘0×5A’ is mapped using an inverted index to its location ‘5’ and ‘11’. Similarly, Figure 4.4 contains the mapping of other IDs to their location in the RDF graph.

URI / Value	ID	Location		
dbo:city	*0x5A*	5	11	
dbo:campus	*0x5B*	9	6	500
dbp:Stanford	*0xB9*	1		
dbp:Harvard	*0xBA*	2		
suburban	*0xD79*	90	55	99
urban	*0xD89*	56	91	
California	*0xD99*	155	255	200
Cambridge	*0xE00*	120	140	

Figure 4.4: Inverted index

4.3.5 RDF graph

Even though the main objective of indexing is to improve the performance by reducing the number of disk access cycles, the proposed *trist* model also reduces the storage space at the expense of additional lookup time. The time-efficient method of indexing the RDF graph is discussed in Section 4.3.5.1. The method of optimising the storage space of the RDF graph is discussed in Section 4.3.5.2.

4.3.5.1 Time-optimised RDF graph

Figure 4.5 shows the sample data stored in the time-optimised RDF graph. Figure 4.5 contains two parts: the first with the address of the data and the second containing the actual data. The time-optimised RDF graph is the classic approach of storing the RDF graph with the URIs and values. The inverted index using the *URI trist* and *value trist* is used to accelerate the access of the RDF graph. This is the approach used by the LDSF for indexing the RDF graph.

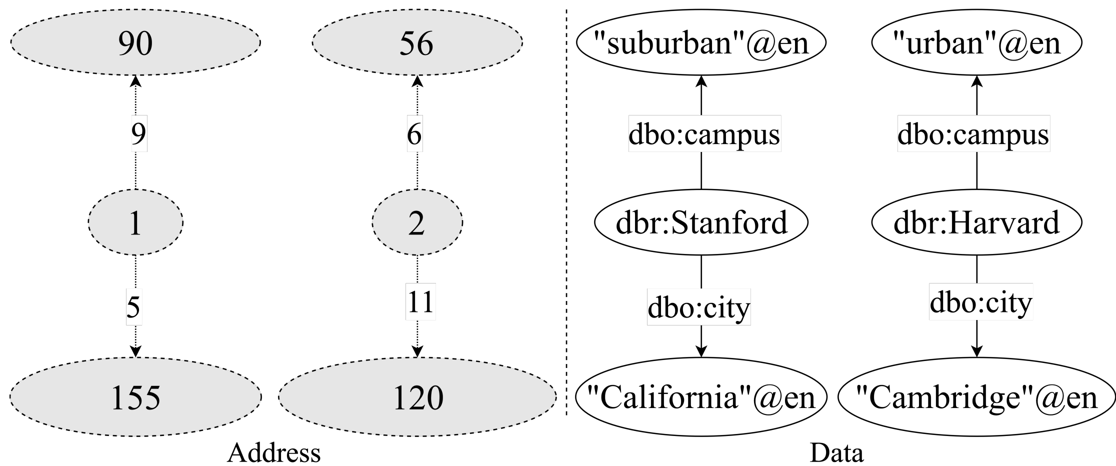


Figure 4.5: Time optimised RDF graph

4.3.5.2 Space-optimised RDF graph

Figure 4.6 shows the sample data stored in the space-optimised RDF graph. The space-optimised RDF graph occupies less storage space by storing the URI-ID instead of the actual URI. The values are not replaced with the ID as it is used in intermediate query processing, and replacing it increases the lookback time during query processing. The query processing is performed completely with the URI-ID, and the URIs of the respective ID are retrieved only while communicating the end results to the user. The URIs are retrieved from the URI-ID by traversing back the *URI trist*. This method consumes additional time to retrieve the URIs from the *URI trist*.

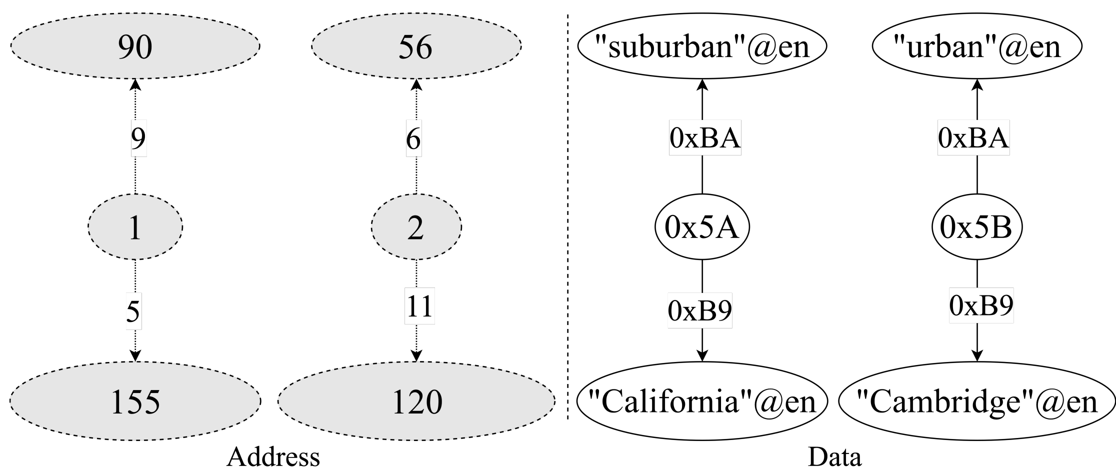


Figure 4.6: Space optimised RDF graph

4.3.5.3 Time complexity

The time complexity to search for a URI or value of '*m*' characters in an RDF graph of '*n*' nodes is $\mathcal{O}(n \times m)$. The time complexity to locate the URI or value in the RDF graph with *trist* indexing is $\mathcal{O}(m)$. The value of '*n*' is in the billions, and the value of '*m*' is in the tens. The time needed to locate the URI or value in the RDF graph with the *trist* is negligible compared to that of searching the entire RDF graph.

4.4 Experimental results

The indexing is evaluated in terms of search time savings and storage size reduction by the time-optimised and space-optimised RDF graphs, respectively. The results are evaluated using the test data generated in Chapter 3, Section 3.4.1. The performance of the access time using *trist* indexing in the time-optimised RDF graph is discussed in Section 4.4.1. The effectiveness of the space-optimised RDF graph is disclosed in Section 4.4.2.

4.4.1 Time-optimised RDF graph

Our experiments were performed on an Intel Core i5 CPU 2.9 GHz and 8 GB of memory running a 64-bit Linux kernel. The *URI trist* is constructed from 77,000 entities. A total of 52-word nodes and 1.4 million character nodes are generated in the *URI trist* for the 77,000 URIs in 1.68 seconds(s). The query to fetch each of the 77,000 URIs is executed ten times. The average query execution time in the RDF graph without indexing is 16.78 millisecond(ms). The average query execution time for the *trist* indexed RDF graph is 0.002 ms. Although the construction of the *trist* index for the 77,000 URIs requires 1.68 s, the use of the *trist* index reduces the search time from 16.78 ms to 0.002 ms.

4.4.2 Space-optimised RDF graph

Table 4.3 shows the results for the storage size comparison of the time and space-optimised RDF graphs. The 7.09 MB of storage occupied by the 77,000 URIs in the time-optimised RDF graph is reduced to 2.96 MB by the space-optimised RDF graph.

Table 4.4 shows the compression ratio and space savings of the time and space-

Table 4.3: Storage size comparison of RDF graphs

Data	Size of time-optimised RDF graph (KB)	Size of space-optimised RDF graph (KB)
Film	487.1	260.1
Artist	443.8	185.3
Director	450.5	193.2
Producer	328.5	146.0
Writer	454.2	198.8
Song	509.8	305.1
Musician	443.5	186.3
Singer	435.5	177.1
All	**7093.7**	**2962.1**

optimised RDF graphs on the test data. The compression ratio is computed as the ratio between the storage size of the time-optimised RDF graph and that of the space-optimised RDF graph. A 7.09 MB RDF graph compressed to 2.96 MB has a compression ratio of $7.09/2.96 = 2.4$, where the ratio of the storage size between the time and space-optimised RDF graphs is $2.4 : 1$. The space savings are obtained using one minus the ratio of the space-optimised RDF graph to the time-optimised RDF graph. Compressing 7.09 MB to 2.96 MB implies a space savings of $1 - (2.96/7.09) = 0.6$, where the space savings of the space-optimised RDF graph is 60% compared to the space occupied by the time-optimised RDF graph.

Table 4.4: Compression ratio and space savings

Data	Compression ratio	Space saving
Film	1.9	0.5
Artist	2.4	0.6
Director	2.3	0.6
Producer	2.2	0.6
Writer	2.3	0.6
Song	1.7	0.4
Musician	2.4	0.6
Singer	2.5	0.6
All	**2.4**	**0.6**

4.5 Summary

This chapter presents a novel data structure called *trist* for use in quickly locating data in an RDF graph without having to search the entire graph. The proposed *URI trist* and *value trist* are used to index the URIs and values of the linked data, respectively. Inverted indexing is used to map the URIs and values stored in the trist-to-RDF graph. Data insertion, updating and deletion are easy in the proposed *trist*-based indexing. The access speed of a trist-indexed RDF graph is up to 6000 times faster than that of a regular RDF graph on the test data. Moreover, the proposed space-optimised RDF graph achieves space savings of 60%.

Chapter 5

Ranking

5.1 Introduction

The process of ranking information is the core component behind any information retrieval framework. Unlike the keyword-based search in web search engines, the search in linked data is usually via SPARQL queries. Even searches based on keywords are executed with SPARQL queries. The method of viewing information related to a particular entity in linked data is called entity browsing. In a typical use case of entity browsing in DBpedia[1], the user would find an average of 180 facts (values) attached to each entity (subject). A user looking for the country "`United States`" from all SPARQL endpoints[2], will obtain 87 thousand values. These enormous amounts of data are too complex for humans to interpret without processing and ranking.

Searching and browsing large volumes of linked data requires complex ranking strategies to guide the user get the relevant information. Notable work on ranking includes triple ranking, resource ranking and property ranking. Substantial effort has been made to develop property ranking, as it is widely applied in entity browsing. Entity browsing in DBpedia uses alphabetical ordering on predicates. Most of the proposed ranking approaches are supervised, and those that follow unsupervised approaches produce poor results.

This chapter presents ranking factors for linked data and the method for applying the approach to the query results. The core elemental components of linked data are the endpoint, concept, predicate and value for which the ranking methods are proposed. The *nexus* proposed in Chapter 3 and the inverted index proposed in Chapter 4 are also used to support various ranking factors. The proposed ranking factors are unsupervised and can rank linked data from multiple endpoints, and the proposed ranking approach can rank any form of query results. An improved user interface (UI) for entity browsing obtained by replacing the URI with human-readable text and embedding multimedia content is also discussed.

[1] http://dbpedia.org/sparql, Accessed on 9th March 2018.

[2] 275 endpoints, discussed in Chapter 3, Section 3.4.1

The remainder of this chapter is organised as follows. Section 5.2 discusses the related work. Section 5.3 provides a detailed description of the proposed ranking technique. Section 5.5 presents the experimental results. Finally, Section 5.6 summarises the work.

5.2 Related work

Dessi & Atzori (2016) proposed a machine-learned ranking (MLR) approach to rank RDF predicates among entities. MLR uses a machine learning approach with nine different features. Specifically, the main feature of interest is ranking based on the frequency of the predicate. This model considers the frequency of the predicate across the data; for example, the count of '*dbo:birthDate*', which is used across almost all domains, will have a higher frequency. However, in some domains, such as '*Actor*', predicates with a low overall score, such as '*dbp:famousMovie*' and '*dbo:award*', are more important than '*dbo:birthDate*'. The other setback of this model is the feature '*IsEnglish*', where the results are biased toward English predicates. The training methods adopted by the MLR are supervised by semantic web experts and students. It would be more complex to train the data from all the endpoints, as the data are enormous and the domain is diverse, with DBpedia alone containing 4.8 lakh distinct domains identified using '*rdf:type*' property.

Lee et al. (2014) proposed a semantic search ranker (SSR) with three measures to rank semantic search results: "*Number of meaningful semantic paths*", "*Coverage of keywords*" and "*Discriminating power of keywords*". The semantic path weight is computed using the number of ways in which the entity can be reached in the RDF graph. For example, for a book entity, the predicate '*eg:hasTitle*' has only one title value; however, the predicate '*eg:writtenBy*' can have many author names assigned to it. This method computes a higher rank for '*eg:writtenBy*' and a lower rank for '*eg:hasTitle*'. However, SSR does not address alternative cases, such as a book having many single-valued predicates such as '*eg:title*', '*eg:publisher*', '*eg:publishingDate*', '*eg:volume*', and '*eg:issue*'. Additionally, SSR does not provide a method for ranking among cases with the same semantic path weight.

Ruback et al. (2017) proposed SELEcTor to find the similarity between two entities in linked data by ranking and comparing their features. SELEcTor generates ranked features with the help of SPARQL query, which matches the path pattern to obtain all the features and ranks based on the count. This approach has a high possibility of incorrectly marking the entities similar to some features, such as '*rdf:type*', '*dc:subject*', '*owl:sameAs*', '*rdf:label*', '*rdf:comment*', that are standard across entities belonging to different domains. Feature identification in SELEcTor is aided by a domain expert.

Axel-Cyrille et al. (2017) proposed a holistic ranking for RDF entities (HARE) to rank resources, predicates, literals and triples. HARE ranks predicates by considering them a resource that is general and common across the domain. However, in the proposed approach, the rank of the predicates is specific to the domain (nexus).

Arnaout & Elbassuoni (2018) proposed a framework to rank the triple pattern in relevance to keywords of the RDF graph. If the exact keyword did not match a triple pattern, they used a query relaxation technique.

Marx et al. (2016) proposed RDF ranking using real user query logs. However, an actual user query is related mostly to the concept and has minimal consideration of predicates. This ranking method is ideal for ranking concepts, but the user query logs alone are not sufficient for predicates.

Noia et al. (2016) proposed a semantic path-based ranking named SPrank for ranking concepts. SPrank uses a machine learning approach for user interactions, such as clicks, purchases, and video watching, to understand the likes and dislikes of the user and, based on this information, to recommend concepts in which users may be interested.

Motivated by these techniques, different approaches to ranking linked data become the focus.

5.3 Proposed ranking technique

This chapter presents a holistic approach to rank the linked data from multiple SPARQL endpoints. For ranking, the edge of the RDF graph in the LDSF-Index is annotated with the endpoint from which the triple is retrieved. The ranking of linked data is based on the following four components.

1. **Endpoint:** The endpoint from which the data are retrieved by LDSF.
2. **Concept:** The node in the RDF graph containing URI.
3. **Predicate:** The edge connecting two nodes in the RDF graph.
4. **Value:** The node in the RDF graph containing the value.

Section 5.3.1 presents the approach of ranking the triples from multiple endpoints. The factors for ranking the linked data concepts are explained in Section 5.3.2. Section 5.3.3 discusses the technique for ranking predicates. Section 5.3.4 explains the technique used to rank values. The method of combining individual ranking factors to rank any form of query result is discussed in Section 5.3.5. Finally, Section 5.4 explains the user interface used for entity browsing.

5.3.1 Endpoint ranking

The ranks of endpoints are calculated based on *nexus* defined in Chapter3, Section 3.1. The rank of an endpoint in a *nexus* is proportional to the number of triples contributed by the endpoint to the *nexus*. For example, the endpoint '*BBC-Music*' contributes more triples to the "*Music-Nexus*" than '*DBpedia*'. If the user searches for music data, then the triples from '*BBC-Music*' are given higher rank than the triples from '*DBpedia*'.

5.3.2 Concept ranking

The concepts of the linked data are ranked based on the following factors:

1. **Keywords:** Table 5.1 shows the list of properties used for describing the concepts along with its priority. The lower the priority value, the higher the importance of the property. The rank of a concept is proportional to the number of search terms that appear in the values of the properties describing the concept.
2. **Country:** Concepts with a state or country tag matching the user location are ranked higher.

Table 5.1: Properties describing concepts

Priority	Property	Use
1	dc:title or dbp:title	A name given to the concept.
2	foaf:name	The name given to the URI, if the URI is a thing or agent.
3	rdfs:label	Provides a human-readable version of the URI.
4	skos:prefLabel	Preferred label.
5	skos:altLabel	Alternative label.
6	rdfs:comment	Provides human-readable description of the URI.

3. **Core *nexus* properties:** Concepts with a higher number of core properties matching the *nexus* of the query are ranked higher.

4. **Endpoints:** The rank of the concepts is proportional to the number of endpoints containing the concept. The concepts from different endpoints are aligned using the '`skos:exactMatch`' property.

5. **Inbound Links:** The rank of a concept is proportional to the number of concepts from other endpoints linked to that concept (similar to page rank).

5.3.3 Predicate ranking

An entity contains an average of 41 predicates[3] and a maximum of 294 predicates[4]. Extensive analysis of the linked open data from multiple SPARQL endpoints revealed that "*the importance of a predicate is directly proportional to its familiarity among its nexus*". The *nexus* generated while clustering triples in Chapter 3, Section 3.3.3 is used to rank the predicates. Let '`p`' be the predicate belonging to *nexus* '`n`'. Let '$Count_{p,n}$' be the count of the predicate '`p`' in *nexus* '`n`'. Let '$Count_{a,n}$' be the count of all the predicates in *nexus* '`n`'. The rank of predicate '`p`' in *nexus* '`n`' is computed using Formula 5.1.

$$Rank_{p,n} = \frac{Count_{p,n}}{Count_{a,n}} \tag{5.1}$$

Note: The same predicate will have a different rank in a different *nexus*, i.e., the rank of the same predicate in a different *nexus* varies as its importance in the *nexus* differs.

[3]Results of DBpedia on 12 Dec 2018.
[4]Results of entity dbr:First_Geneva_Convention on 12 Dec 2018.

5.3.4 Value ranking

The ranking of values is computed using the mapping of the inverted index proposed in Chapter 4, Section 4.3.4. The value rank is proportional to the number of its mapping elements in the inverted index. A value with a higher number of mapping elements in the inverted index is given a higher rank.

5.3.5 Aggregate ranking approach

The most common applications of linked data are

1. **Concept listing (C):** Concept listing is the listing of concepts matching the query. Here, concepts are ranked using the concept ranking factors discussed in Section 5.3.2.

2. **Entity browsing (PO):** Entity browsing is the most common use case of linked data. Entity browsing lists the predicate and object related to a concept. First, the predicates are ordered based on the predicate ranking score proposed in Section 5.3.3; then, under each predicate, the values are ranked using the mapping of the inverted index, as proposed in Section 5.3.4.

3. **Higher order listing:** For queries with three or more result components, the ranking is done from the first component to the last component. For example, consider the following sample query.

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT
?Concept-1 ?Concept-2 ?Property ?Value
WHERE{
?Concept-1 rdfs:subClassOf ?Concept-2 .
?Concept-2 ?Property ?Value .
}
```

   Here, '`Concept-1`' is ranked first, followed by '`Concept-2`', then '`Property`' and finally '`Value`'.

5.4 User interface: Case of entity browsing

Although LDSF allows the user to execute arbitrary SPARQL queries, a unique characteristic of UI is proposed for entity browsing. The employed UI model is inspired by Wikipedia[5] to define the title, abstract and image, similarly to Wikipedia, using

[5]Website: www.wikipedia.org, Accessed on 22 March 2019.

the properties '*rdfs:label*', '*dbo:abstract*' and '*dbo:thumbnail*', respectively. The URIs enable the machine to understand the linked data. In entity browsing, the URIs are replaced with human-readable text from the following properties.

- dc:title
- dbp:title
- foaf:name
- rdfs:label
- skos:prefLabel
- skos:altLabel

The values are selected using the language tag attached to the value based on user selection. Multimedia files, such as images, audio and video, represented using URLs in linked data are embedded in the page for insight and to reduce the time needed by the user to access this information separately. First, the predicates are ordered based on the predicate ranking score proposed in Section 5.3.3; then, under each predicate, the values are ranked using the mapping of the inverted index, as proposed in Section 5.3.4.

5.5 Experimental results

The rankings are evaluated in terms of Spearman's rank correlation coefficient (ρ) (Spearman, 1904) for entity browsing. Ten entities are selected randomly in each category from the test data generated in Chapter 3, Section 3.4.1. The entities are ranked based on properties and values, as discussed in Section 5.4. The ranked properties and values are compared with the ordering of Wikipedia. The pages on Wikipedia are contributed by many users around the world, and the ordering of the text on Wikipedia is considered readable by most users. The ordering of the properties was extracted from Wikipedia[6] articles.

Spearman's rank correlation coefficient (ρ) Spearman (1904) was used to assess the similarity between the two ranking systems. ρ ranges from -1 to $+1$, and the value is

[6]https://www.wikipedia.org Accessed on 16 Feb 2019.

large high when the rankings are similar and small when the rankings are dissimilar. ρ is computed using Formula 5.2:

$$\rho = 1 - \frac{6 \sum d_i^2}{n(n^2 - 1)} \tag{5.2}$$

where '*d*' is the difference between the ranks and '*n*' is the number of observations.

ρ was used to evaluate the ranking of the entries in DBpedia and LDSF based on the ordering of Wikipedia, as shown in Table 5.2. DBpedia, by default, uses alphabetical ordering of the predicates. The proposed LDSF ranking is 50.5% more similar to the ordering of Wikipedia than the results of DBpedia.

Table 5.2: Spearman's rank correlation coefficients of DBpedia and LDSF

Data	**DBpedia (ρ)**	**LDSF (ρ)**
Film	0.199	0.838
Artist	0.525	0.849
Director	0.489	0.955
Producer	0.452	0.922
Writer	0.340	0.990
Song	0.134	0.920
Musician	0.433	0.807
Singer	0.525	0.851
Average	**0.387**	**0.892**

Table 5.3 compares the ranking of LDSF with that of related works. Although previous approaches have evaluated their rankings based on a small number of users, the proposed approach extends the number of users to a large, unbiased crowd (Wikipedia). The ranking correlation of LDSF is better than that of the other methods.

Table 5.3: Comparison of LDSF ranking with related work

Method	**Year**	**Spearman's (ρ)**
Ranking DBpedia Properties (Atzori & Dessi, 2014)	2014	33.3 %
DBtrends (Marx et al., 2016)	2016	37 %
Machine learning approach (Dessi & Atzori, 2016)	2016	74 %
Holistic and Scalable Ranking (Axel-Cyrille et al., 2017)	2017	30 %
Proposed LDSF ranking	2020	89.2 %

5.6 Summary

This chapter discusses unsupervised methods for ranking linked data from multiple endpoints. Ranking factors based on endpoint, concept, predicate and value are explained, and the application to query results is discussed. The ranking of predicates based on the *nexus* and the ordering of the values based on the inverted-index is widely used in entity browsing. An improved UI for entity browsing obtained by replacing the URI with human-readable text and embedding multimedia content is also discussed. The recommended ranking for entity browsing achieves a Spearman rank correlation coefficient of up to 99% with the ordering of Wikipedia. The ranking approach is one factor to consider, but users are the end consumers of the information. Typically, users have to select and filter data to find relevant information.

Chapter 6

Conclusions

This work discusses LDSF, a framework for storing, partitioning, indexing and ranking linked data. An advancement over the existing character-based encoding method is proposed to store linked data. The proposed system, named 'WordCode', is used to efficiently store the entire community of text data (including linked data). WordCode is an extension of Unicode with the additional capability to encode words. WordCode can encode up to 67.9 billion words, irrespective of language, each with a maximum size of five bytes. Additionally, the structure of the code page is upgraded from the regular table-based storage used by character encoding to a customised trie model named 'WordTrie'. Because WordCode-encoded files are smaller than Unicode-encoded files, machines handling text data (including linked data) with WordCode encoding have a reduced workload compared to machines processing text data with Unicode encoding.

The term *nexus* is redefined according to linked data as the "*set of subjects belonging to the same type*". Linked data are partitioned based on a bi-level *nexus* clustering algorithm using the core predicates. The key idea is to identify the two-level core predicates of the subjects and use them for partitioning. The proposed algorithm partitions the gold standard test data with a precision of 98.7% and recall of 87.7%.

The linked data are indexed using a novel data structure called *trist* to accelerate RDF graph access. The URIs and values are indexed using *URI trist* and *value trist*, respectively. Inverted indexing is used to map the URIs and values stored in the trist-to-RDF graph. The access speed of a trist-indexed RDF graph is up to 6000-times faster than that of the regular RDF graph on the test data. Moreover, the proposed space-optimized RDF graph achieves a space savings of 60%.

An unsupervised methods for ranking linked data from multiple endpoints is proposed. Ranking factors based on endpoint, concept, predicate and value are explained, and the application to query results is discussed. An improved UI for entity browsing obtained by replacing the URI with human-readable text and embedding multimedia content is also discussed. The recommended ranking for entity browsing achieves a Spearman rank correlation coefficient of up to 99% with the ordering of Wikipedia.

Overall, the proposed framework for information retrieval from linked data incorporating storing, partitioning, indexing and ranking is more efficient than the existing systems.

In future, this work can be extended in the following ways:

1. **Temporal linked data:** Any data changes over time. For example, the population of a country changes over time. The method of indexing and presenting the view of data over a period of time to the user might be a promising future direction of research. This kind of information will allow ranking based on time.

2. **Publisher Information:** Tagging the publisher information to the linked data will help to rank the trustworthiness of the data. Additionally, allowing the user to provide feedback on incorrect results will improve the quality of linked data and make linked data successful in the long run.

References

Ali, W., Saleem, M., Yao, B., & Ngomo, Axel Cyrille, N. (2020). Storage, Indexing, Query Processing, and Benchmarking in Centralized and Distributed RDF Engines: A Survey. *Preprints*.

Aluç, G., Özsu, M. T., & Daudjee, K. (2014). Workload matters: Why RDF databases need a new design. *Proceedings of the VLDB Endowment*, *7*(10), 837–840.

Aoe, J.-I., Morimoto, K., & Sato, T. (1992). An efficient implementation of trie structures. *Software: Practice and Experience*, *22*(9), 695–721.

Arnaout, H. & Elbassuoni, S. (2018). Effective searching of RDF knowledge graphs. *Journal of Web Semantics*, *48*, 66 – 84.

Atzori, M. & Dessi, A. (2014). Ranking DBpedia properties. In *IEEE 23rd International WETICE Conference, 2014*, 441–446. IEEE.

Axel-Cyrille, Ngomo, N., Hoffmann, M., Usbeck, R., & Jha, K. (2017). Holistic and scalable ranking of RDF data. In *IEEE International Conference on Big Data, 2017*, 746–755.

Azad, M. A. K., Sharmeen, R., Ahmad, S., & Kamruzzaman, S. M. (2005). An efficient technique for text compression. In *International Conference on Information Management and Business (IMB), 2005*, volume 1009.4981, 467–473.

Baeza-Yates, R. & Ribeiro-Neto, B. (1999). Modern information retrieval. ACM press New York.

Berners-Lee, T., Fielding, R., & Masinter, L. (2005). Uniform resource identifier (URI): Generic syntax. RFC 3986, IETF.

Berners-Lee, T., Hendler, J., & Lassila, O. (2001). The Semantic Web. *Scientific American*, *284*(5), 34–43.

Chen, X., Chen, H., Zhang, N., & Zhang, S. (2015). SparkRDF: Elastic discreted RDF graph processing engine with distributed memory. In *2015 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*, volume 1, 292–300. IEEE.

Cheng, G., Ge, W., & Qu, Y. (2008). Falcons: Searching and browsing entities on the semantic web. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, 1101–1102., New York, NY, USA. ACM.

Clark, K. G., Feigenbaum, L., & Torres, E. (2008). SPARQL protocol for RDF. *World Wide Web Consortium (W3C) Recommendation*.

d'Aquin, M. & Motta, E. (2011). Watson, more than a semantic web search engine. *Semantic Web*, *2*(1), 55–63.

Dessi, A. & Atzori, M. (2016). A machine-learning approach to ranking RDF properties. *Future Generation Computer Systems*, *54*, 366–377.

Ding, L., Finin, T., Joshi, A., Pan, R., Cost, R. S., Peng, Y., Reddivari, P., Doshi, V., & Sachs, J. (2004). Swoogle: A search and metadata engine for the semantic web. In *Proceedings of the Thirteenth ACM International Conference on Information and Knowledge Management*, CIKM '04, 652–659., New York, NY, USA. ACM.

Faye, D. C., Curé, O., & Blin, G. (2012). A survey of RDF storage approaches. *Revue Africaine de la Recherche en Informatique et Mathématiques Appliquées*, *15*, 11–35.

Ferrara, A., Genta, L., Montanelli, S., & Castano, S. (2015). Dimensional clustering of linked data: Techniques and applications, 55–86. Springer Berlin Heidelberg.

Fredkin, E. (1960). Trie Memory. *Communications of the ACM*, *3*(9), 490–499.

Galárraga, L., Hose, K., & Schenkel, R. (2014). Partout: A Distributed Engine for Efficient RDF Processing. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14 Companion, 267–268., New York, NY, USA. Association for Computing Machinery.

Gong, S., Hu, W., Li, H., & Qu, Y. (2018). Property clustering in linked data: An empirical study and its application to entity browsing. *International Journal on Semantic Web and Information Systems (IJSWIS)*, *14*(1), 31–70.

Grabowski, S. & Swacha, J. (2010). Language-independent word-based text compression with fast decompression. In *Proceedings of VIth International Conference on Perspective Technologies and Methods in MEMS Design (MEMSTECH)*, 20–23. IEEE.

Halpin, H., Hayes, P. J., McCusker, J. P., McGuinness, D. L., & Thompson, H. S. (2010). When owl:sameAs Isn't the Same: An Analysis of Identity in Linked Data. In *The Semantic Web – ISWC 2010*, 305–320., Berlin, Heidelberg. Springer Berlin Heidelberg.

Harbi, R., Abdelaziz, I., Kalnis, P., & Mamoulis, N. (2015). Evaluating SPARQL queries on massive RDF datasets. *Proceedings of the VLDB Endowment*, *8*(12), 1848–1851.

Harbi, R., Abdelaziz, I., Kalnis, P., Mamoulis, N., Ebrahim, Y., & Sahli, M. (2016). Accelerating SPARQL queries by exploiting hash-based locality and adaptive partitioning. *The VLDB Journal*, *25*(3), 355–380.

Hogan, A., Harth, A., Umbrich, J., Kinsella, S., Polleres, A., & Decker, S. (2011). Searching and browsing Linked Data with SWSE: The Semantic Web Search Engine. *Web semantics: science, services and agents on the world wide web*, *9*(4), 365–401.

Hu, C., Wang, X., Yang, R., & Wo, T. (2016). ScalaRDF: a distributed, elastic and scalable in-memory RDF triple store. In *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*, 593–601. IEEE.

Huang, J., Abadi, D. J., & Ren, K. (2011). Scalable SPARQL querying of large RDF graphs. *Proceedings of the VLDB Endowment*, *4*(11), 1123–1134.

ISO/IEC 10646 (2017). Universal Coded Character Set (UCS).

Kalajdzic, K., Ali, S. H., & Patel, A. (2015). Rapid lossless compression of short text messages. *Computer Standards & Interfaces*, *37*, 53–59.

Kaoudi, Z. & Manolescu, I. (2015). RDF in the clouds: A survey. *The VLDB Journal*, *24*(1), 67–91.

Knuth, D. E. (1985). Dynamic Huffman Coding. *Journal of Algorithms*, *6*(2), 163–180.

Lee, J., Min, J.-K., Oh, A., & Chung, C.-W. (2014). Effective ranking and search techniques for web resources considering semantic relationships. *Information Processing & Management*, *50*(1), 132–155.

Lee, K. & Liu, L. (2013). Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning. *Proceedings of the VLDB Endowment*, *6*(14), 1894–1905.

Ma, Z., Capretz, M. A. M., & Yan, L. (2016). Storing massive Resource Description Framework (RDF) data: A survey. *The Knowledge Engineering Review*, *31*(4), 391–413.

Marx, E., Zaveri, A., Moussallem, D., & Rautenberg, S. (2016). DBtrends: Exploring Query Logs for Ranking RDF Data. In *Proceedings of the 12th International Conference on Semantic Systems*, 9–16. ACM.

Mayfield, J. & McNamee, P. (2003). Single n-gram stemming. In *Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*, 415–416.

Mirizzi, R., Ragone, A., Di Noia, T., & Di Sciascio, E. (2010). Ranking the Linked Data: The Case of DBpedia. In *Web Engineering*, 337–354. Springer Berlin Heidelberg.

Nentwig, M., Groß, A., Möller, M., & Rahm, E. (2017). Distributed holistic clustering on linked data. In *On the Move to Meaningful Internet Systems. OTM 2017 Conferences*, 371–382. Springer International Publishing.

Nentwig, M., Groß, A., & Rahm, E. (2016). Holistic entity clustering for linked data. In *2016 IEEE 16th International Conference on Data Mining Workshops (ICDMW)*, 194–201.

Noia, T. D., Ostuni, V. C., Tomeo, P., & Sciascio, E. D. (2016). SPrank: Semantic Path-based Ranking for Top-N recommendations using linked open data. *ACM Transactions on Intelligent Systems and Technology (TIST)*, *8*(1).

Oh, H., Chun, S., Eom, S., & Lee, K.-H. (2015). Job-optimized map-side join processing using mapreduce and hbase with abstract RDF data. In *2015 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*, volume 1, 425–432. IEEE.

Özsu, M. T. (2016). A survey of RDF data management systems. *Frontiers of Computer Science*, *10*(3), 418–432.

Pan, Z., Zhu, T., Liu, H., & Ning, H. (2018). A survey of RDF management technologies and benchmark datasets. *Journal of Ambient Intelligence and Humanized Computing*, *9*(5), 1693–1704.

Papailiou, N., Tsoumakos, D., Konstantinou, I., Karras, P., & Koziris, N. (2014). H2RDF+: An Efficient Data Management System for Big RDF Graphs. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, 909–912., New York, NY, USA. Association for Computing Machinery.

Prud'hommeaux, E. & Seaborne, A. (2007), SPARQL Query Language for RDF.

Punnoose, R., Crainiceanu, A., & Rapp, D. (2015). SPARQL in the cloud using Rya. *Information Systems*, *48*, 181–195.

Ruback, L., Casanova, M. A., Renso, C., & Lucchese, C. (2017). Selector: discovering similar entities on linked data by ranking their features. In *2017 IEEE 11th International Conference on Semantic Computing (ICSC)*, 117–124. IEEE.

Sinaga, A., Adiwijaya, & Nugroho, H. (2015). Development of word-based text compression algorithm for Indonesian language document. In *2015 3rd International Conference on Information and Communication Technology (ICoICT)*, 450–454.

Singh, G., Upadhyay, D., & Atre, M. (2018). Efficient RDF dictionaries with B+ trees. In *Proceedings of the ACM India Joint International Conference on Data Science and Management of Data*, 128–136.

Spearman, C. (1904). The proof and measurement of association between two things. *American journal of Psychology*, *15*(1), 72–101.

Stevens, R., Lord, P., Malone, J., & Matentzoglu, N. (2019). Measuring expert performance at manually classifying domain entities under upper ontology classes. *Journal of Web Semantics*, *57*, 100469.

The Unicode Consortium (2015). The Unicode Standard, Version 8.0.

Tummarello, G., Cyganiak, R., Catasta, M., Danielczyk, S., Delbru, R., & Decker, S. (2010). Sig.ma: Live views on the web of data. *Journal of Web Semantics*, *8*(4), 355 – 364. Semantic Web Challenge 2009 User Interaction in Semantic Web research.

Tummarello, G., Delbru, R., & Oren, E. (2007). Sindice. com: Weaving the open linked data. In *The Semantic Web* 552–565. Springer.

V. Biron, P., Permanente, K., & Malhotra, A. (2004), XML Schema Part 2: Data types Second Edition.

Vandenbussche, P.-Y., Umbrich, J., Matteis, L., Hogan, A., & Buil-Aranda, C. (2017). SPARQLES: Monitoring public SPARQL endpoints. *Semantic Web*, *8*(6), 1049–1065.

Waidyasooriya, H. M., Ono, D., Hariyama, M., & Kameyama, M. (2014). Efficient data transfer scheme using word-pair-encoding-based compression for large-scale text-data processing. In *2014 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, 639–642.

Weiss, C., Karras, P., & Bernstein, A. (2008). Hexastore: Sextuple indexing for semantic web data management. *Proceedings of the VLDB Endowment*, *1*(1), 1008–1019.

Witten, I. H., Witten, I. H., Neal, R. M., Neal, R. M., Cleary, J. G., & Cleary, J. G. (1987). Arithmetic coding for data compression. *Communications of the ACM*, *30*(6), 520–540.

Wylot, M., Hauswirth, M., Cudré-Mauroux, P., & Sakr, S. (2018). RDF Data Storage and Query Processing Schemes: A Survey. *ACM Computing Surveys*, *51*(4).

Xu, Z., Chen, W., Gai, L., & Wang, T. (2015). SparkRDF: In-Memory Distributed RDF Management Framework for Large-Scale Social Data. In *International Conference on Web-Age Information Management*, 337–349. Springer.

Zeng, K., Yang, J., Wang, H., Shao, B., & Wang, Z. (2013). A distributed graph engine for web scale RDF data. *Proceedings of the VLDB Endowment*, *6*(4), 265–276.

Ziv, J. & Lempel, A. (1977). A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, *23*(3), 337–343.

Ziv, J. & Lempel, A. (1978). Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, *24*(5), 530–536.

Publications

Patent

1. Ananthanarayana V. S., Sakthi Murugan R., Inventor; National Institute of Technology Karnataka, Assignee. “Method and System for Dynamic Word Encoding and Compression of Data.” India Patent Application Number 201641025735, applied on July 27, 2016, **(Patent received examination report)**.

Journal

1. Sakthi Murugan R., and Ananthanarayana V. S. (2019), ”WordCode using WordTrie”, *Elsevier Journal of King Saud University - Computer and Information Sciences*, DOI: https://doi.org/10.1016/j.jksuci.2019.05.011.

2. Sakthi Murugan R., and Ananthanarayana V. S. (2019), ”LDSF: A large-scale linked data search framework”, *International Journal on Semantic Web and Information Systems (IJSWIS)*, (**Under review**).

Conference

1. Sakthi Murugan R., and Ananthanarayana V.S., ”NPRank: Nexus based Predicate Ranking of Linked Data”, *5th International Conference on Data Science and Engineering (ICDSE 2019)*, held at IIT Patna, India,
DOI: https://doi.org/10.1109/ICDSE47409.2019.8971791.

Curriculum Vitae

Mr. Sakthi Murugan R
Part-Time Research Scholar
Department of Information Technology
National Institute of Technology Karnataka
P.O. Srinivasanagar, Surathkal
Mangalore-575 025

Permanent Address

Sakthi Murugan R
Door No. 97, Vellala Street
Pondicherry -605001
India.
Email: sakthimuruga@gmail.com
Mobile: +919994298296.

Professional Experience

1. Senior Consultant (Technical) in Kudzu Infotech - MicroFocus, from Aug'2019 to Mar'2020.
2. Software Engineer in HCL Technologies, Chennai, from 25th May'2011 to 29th June'2012.

Academic Records

1. M.Tech. in Network and Internet Engineering from Pondicherry University, Pondicherry, India, 2014.
2. B.Tech. in Computer Science and Engineering from Sri Manakula Vinayagar Engineering College, Pondicherry, India, 2011.

Research Interests

Linked Data
Semantic Web
Information Retrieval
Word Encoding