Thesis for the Degree of Doctor of Philosophy

PROACTIVE MAINTENANCE FOR EVOLVING LINKED DATA USING CHANGE-AWARE QUERY CACHING

Usman Akhtar

Department of Computer Science and Engineering
Graduate School
Kyung Hee University
South Korea

February 2021

PROACTIVE MAINTENANCE FOR EVOLVING LINKED DATA USING CHANGE-AWARE QUERY CACHING

Usman Akhtar

Department of Computer Science and Engineering
Graduate School
Kyung Hee University
South Korea

February 2021

PROACTIVE MAINTENANCE FOR EVOLVING LINKED DATA USING CHANGE-AWARE QUERY CACHING

By

Usman Akhtar

Supervised by

Prof. Sungyoung Lee

Submitted to the Department of Computer Science and Engineering and the Faculty of Graduate School of Kyung Hee University in partial fulfillment of the requirements of the degree of Doctor of Philosophy

Dissertation Committee:

Prof. Tae-Seong Kim _____

Prof. Seungkyu Lee _____

Prof. Yong-IK Yoon _____

Prof. Eui-Nam Huh _____

Prof. Sungyoung Lee _____

For the dreams of my father,

the efforts of my mother,

the trust of my wife,

and

the support of my brother and sisters,

I succeed, I prevail, I achieve

For their endless love, support and encouragement, thank you for encouraging me in all my pursuits and inspiring me to follow my dreams. I am especially grateful to my parents and wife, who supported me in all hard time to keep in the position to complete my Ph.D. degree.

Along with my all respected, hardworking and supportive teachers.

Abstract

The recent emerging technology such as the Internet of Things (IoT) bring internet connectivity to the physical objects and allow more resources to connect the internet and provide services on the Web. Due to the expansion of the digital system, the way information is spread has a great impact on the business. The Web of data provides an interlinked data space and allows search, querying, and reasoning. Efficient discovery of and querying data on the Web remains a key challenge due to the limited capability and high latency of searching on the Web. Semantic Web Technology promotes common data formats on the Web as a Resource Description Framework (RDF), and to query these data the SPARQL endpoints are available that allow searching and re-using of these data. The proliferation of semantic big data has resulted in a large amount of content published over the LOD. Semantic Web application consumes these data by issuing SPARQL queries.

Recently, Linked Data has emerged as one of the best practices to represent and connect these repositories, also allowing the exchange of information in an interoperable manner. Linked Data not only supports the integration of multiple data from diverse sources but also provides a way to query these datasets. The vast amount of data, especially in the field of Bio-medical are being published as Linked Data. As the Linked Open Data (LOD) cloud is a global information space and offers a wealth of structured facts, which are useful for a wide range of usage scenarios. Whenever there is public data available, data consumers want to query it, and nothing is more compelling than querying the vast amount of the Linked Data. With over 800 million triples are currently stored in DBpedia, the search of the specific resources has never been this high before.

The Linked Data cloud handles a large number of requests from applications consuming the data. Many of the current data analytic solutions require continuous access to these data sets. The major challenge faced by querying the Linked Data cloud on account of the inherently distributed

nature of Linked Data is its high search latency and lack of tools to connect SPARQL endpoints. Accessing Linked Data cloud at query time is prohibited due to high latency in searching the content and limited capability of tools to connect to these databases. Therefore, the performance of retrieving data from LOD repositories is one of the major challenges. In spite of the increased performance of SPARQL endpoints the main problem remains due to the low availability of these endpoints as on average the downtime of SPARQL endpoints is more than 2 days each month. A survey conducted on 427 public SPARQL endpoints registered on the DataHub shows the low efficiency of these endpoints with an availability rate above 90%. Although the Linked Data cloud supports SPARQL queries to access data from its publicly available interfaces, a central problem is the lack of trust regarding these endpoints due to network instability and latency. Therefore, the typical solution is to dump the data locally and maintain endpoints to process these data. The data stored at the local endpoints are not up-to-date and require constant updates, therefore, accurately hosting the endpoints requires expensive infrastructure support.

Existing research shows that the SPARQL server with high demands is often hard to host and which is further complicated as the endpoints are publicly hosted due to the unpredictable workloads. The current way of querying the SPARQL endpoint is to utilize the HTTP that is implemented on top. The client sent the request through these endpoints and the server returns the request. Due to the massive data involved server needs to execute a significant amount of work. The SPARQL query processing is different than the regular HTTP processing as the query-based partitioning of resources occurs. Therefore, regular HTTP caching strategies can not be fully applied to the Linked Data scenario. In the recent past, many efforts have been made to improve the performance of effectively querying Linked Data as compared to querying a triple stored in a relational database, querying a triplestore is still slower by 20%. To circumvent the problem of effectively querying Linked Data caching is the most popular technique to reduce query time by serving the requests from a cache. The idea of query caching is to reuse the previously issued queries. In the distributed setting the benefits of caching are more evident, as the previously issued queries are stored on the remote source. The caching will immediately improve the robustness of the system where the remote resources are not available due to the network insatiability. Existing caching approaches consider caching of entire query result which means that similar queries can

not be served from cache.

To cope with the above challenge, we argue in this thesis, it is advantageous to maintain a local cache for efficient querying and processing. We build a decentralized cache in which the client can take care of their own processing and maintain the access patterns of similar queries. Therefore, the client solves the complex queries and only allow the server to request simple data retrieval operations. Due to the continuous evolution of the LOD cloud, local copies have become outdated. In order to utilize the best resources, improvised scheduling is required to maintain the freshness of the local data cache. In this thesis, we have proposed an approach to efficiently capture the changes and replace the cache. Our proposed approach, called Application-Aware Change Prioritization (AACP), consists of a change metric that quantifies the changes in LOD, and a weight function that assigns importance to recent changes. We have also proposed update policies, called Preference-Aware Source Update (PASU), which incorporates the previous estimation of changes and establishes when the local data cache needs to be upgraded. The aim of this work to accelerate the overall query processing of the LOD cloud. Our work alleviates the burden on the SPARQL endpoint by identifying subsequent queries learned from client historical query patterns and caching the result of these queries.

Acknowledgement

First and foremost, I deliver my humble and sincere thanks to the **Almighty and great Merciful ALLAH** for showering His blessings upon me. He gave me patience, courage, strength, and introduced to me all those people who made my studies a pleasant experience. Without His mercy and help, I could not have made any step forward.

I would like to thank my advisor, Respected Professor Sungyoung Lee, and co-advisor Professor Eui-Nam Huh for providing me the opportunity to pursue my education further and become a Ph.D. His guidance, support, and encouragement have been the key to my success and achievements. Above all, his kindness towards me as a father-figure is what I cherish the most. I am nothing but grateful. He has a great role in polishing my skills such as technical soundness, thinking, and creativity which are key ingredients for high-quality research. I appreciate my dissertation evaluation committee for their valuable observations and insight recommendations during the dissertation defense. These comments enhanced the presentation and contents of the dissertation.

I am also would like to acknowledge my colleagues and seniors, especially Dr. Bilal Amin and Dr. Wajahat Ali Khan for their guidance. I would like to thank all the members of the Ubiquitous Computing Laboratory for their kind support and for providing a beautiful working environment. I would like to thanks Muhammad Asif Razzaq, Hafiz Syed Muhammad Bilal, Syed Imran Ali, Ubaid ur Rehman, Dr. Shujaat Hussain, Dr. Jamil Hussain, Dr. Mugahed A. Al-antari, Dr. Maqbool Ali, Dr. Taqdir Ali, Dr. Maqbool Hussain, Dr. Muhammad Afzal, Dr. Maqbool Ali, Musarrat Hussain, Fahad Ahmed Satti, Hua-Cam Hao, Muhammad Sadiq, Asim Abbas, Muhammad Zaki Ansaar. Also, I would like to thank Mrs. Kim, Dr. Taeho, and Dr. Jaehun. Bang, for being available and helping me in many regards and always been kind when asked for help. I also appreciated all of my current and former Kyung Hee University fellows for their kind support to my personal

and academic life in Korea, especially Uman Khalid, Aunas Manzoor, Muhammad Saad Qasir, Muhammad Asad Ullah, Dr. Aftab Alam, Dr. Oresti Banos, Dr. Kifayat Ullah, Dr. Ahsan Raza Kazmi. This journey would have been quite difficult without their support.

I am thankful to my MS advisor Dr. Mehdi Hassan in Air University, Islamabad for his motivation to continue further education and grow as a researcher. Last but not the least, I would like to express my sincere gratitude to my childhood friend especially Mr. Qasim Bilal.

Last but not the least, I would like to express my sincere gratitude to my parents (Akhtar Ali Tanvir and Nasim Akhtar), Wife (Hashmoonah Ali), sisters (Faryal Akhtar, Saira Akhtar, Ayesha Akhtar), and brother (Hassan Akhtar) for their endless love, support, prayers, and encouragement. Especially, my wife Hashmoonah Ali, her support and encouragement while living in Korea has made this dissertation possible. To my family, thank you for believing in me, I always knew that you believed in me and wanted the best for me. Finally, my parents, Akhtar Ali Tanvir (father) Nasim Akhtar (mother), who deserves my title more than me, thank you for teaching me that my job in life was to learn, to be happy, and to know and understand myself; only then could I know and understand others. Wish, I could be half as good as you.

Usman Akhtar

February 2021

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

The Web often contains valuable information, but often poorly manage and inaccurate information can be found. The Web of data is often interlinked data space and allows search, querying, and reasoning. Semantic Web Technology promotes common data formats on the Web as a Resource Description Framework (RDF), and to query these data the SPARQL endpoints are available that allow searching and re-using data. As the Web of data evolves and information is being added and removed, therefore local copies of Web sources need to be updated from time to time to ensure the quality and consistency of data. Due to the evolution of data many Web data application maintains local copies and naive approach for detecting changes in the Web of data is to download the arbitrary-size RDF data and compare them [1–3]. Whereas most research has been conducted in generating and publishing the RDF data, this thesis proposed novel approaches on efficient management of the evolution of Web data to achieve optimal accuracy and efficient use of available computational resources.

The World Wide Web is based on a graph that is connected through edges and hyperlinks. The content of the Web pages is normally connected with the Hypertext Markup Language (HTML) and transferred using the Hypertext Transfer Protocol (HTTP). Tim Berners-Lee describes the key features of LOD as to provide the unique Uniform Resource Identifier (URI) for referencing entities, allow accessing those URIs through HTTP and use the standard to represent information [4]. Therefore, URIs and HTTP are the essential features of LOD to access the data on the Web. The main goal of the LOD is to retrieve and interpret information as an HTML document only provides rendering information. Whereas LOD resources can be utilized for complex reasoning. Many famous projects utilized LOD for example DBpedia [5] aims at publishing the structured

contents from the Wikipedia pages. Other projects that focus on publishing the specific resources is LinkedGeoData [6] that extract the information from the OpenStreetMap[1] (a community project). This project is providing more than one billion resources. Recently, domain-specific projects are popular that utilized LOD such as LinkedCT [7] which contains information about clinical trials. Currently, thousands of LOD resources are available that offer no restriction on using and altering the content that enables users to freely utilize these resources. Nowadays, the Semantic Web community promotes Linked Open Data (LOD) to address interoperability and sharing issues for online datasets [8]. The LOD cloud offers a wealth of information including geo-location facts[2] and cross-domain information. Currently, it is estimated that more than thirty billion facts have been published over LOD [9]. The format of LOD is encoded as RDF,[3] which consists of a subject, a predicate, and an object that is stored in Triplestore[4]. RDF is widely used as an information model for vast semantic data. However, in the RDF data model, the querying complexity is higher than the relational data model. Currently, widely used RDF datasets such as DBpedia [10], handle abundant requests from diverse applications [11].

In the recent past, massive amounts of data are available publicly and these data are produced at an alarming rate in all domains of medical sciences, creating what we today call ”Big Data”. The era of big data generates new opportunities for the research community to build solutions to search and integrate life sciences data effectively [12]. Most of these datasets have varied formats or lack any format and are stored in disparate locations that are poorly linked, which hinders the operator from efficiently searching and retrieving data. Recently, Linked Data has emerged as one of the best practices to represent and connect these repositories, also allowing the exchange of information in an interoperable manner. Linked Data ontologies not only support the integration of multiple data from diverse sources but also provide an efficient way to query these datasets. Healthcare systems and recent technologies have realized the importance of acquiring and preserving Big data streams for decision making. Therefore, with the evolution of big data in healthcare, the research trends have shifted from massive storage to efficient analysis of data [13].

In various healthcare disciplines, the use of large datasets is becoming popular and the data is

[1] http://www.openstreetmap.org/
[2] https://linkedgeodata.org/
[3] https://www.w3.org/RDF/
[4] https://jena.apache.org/

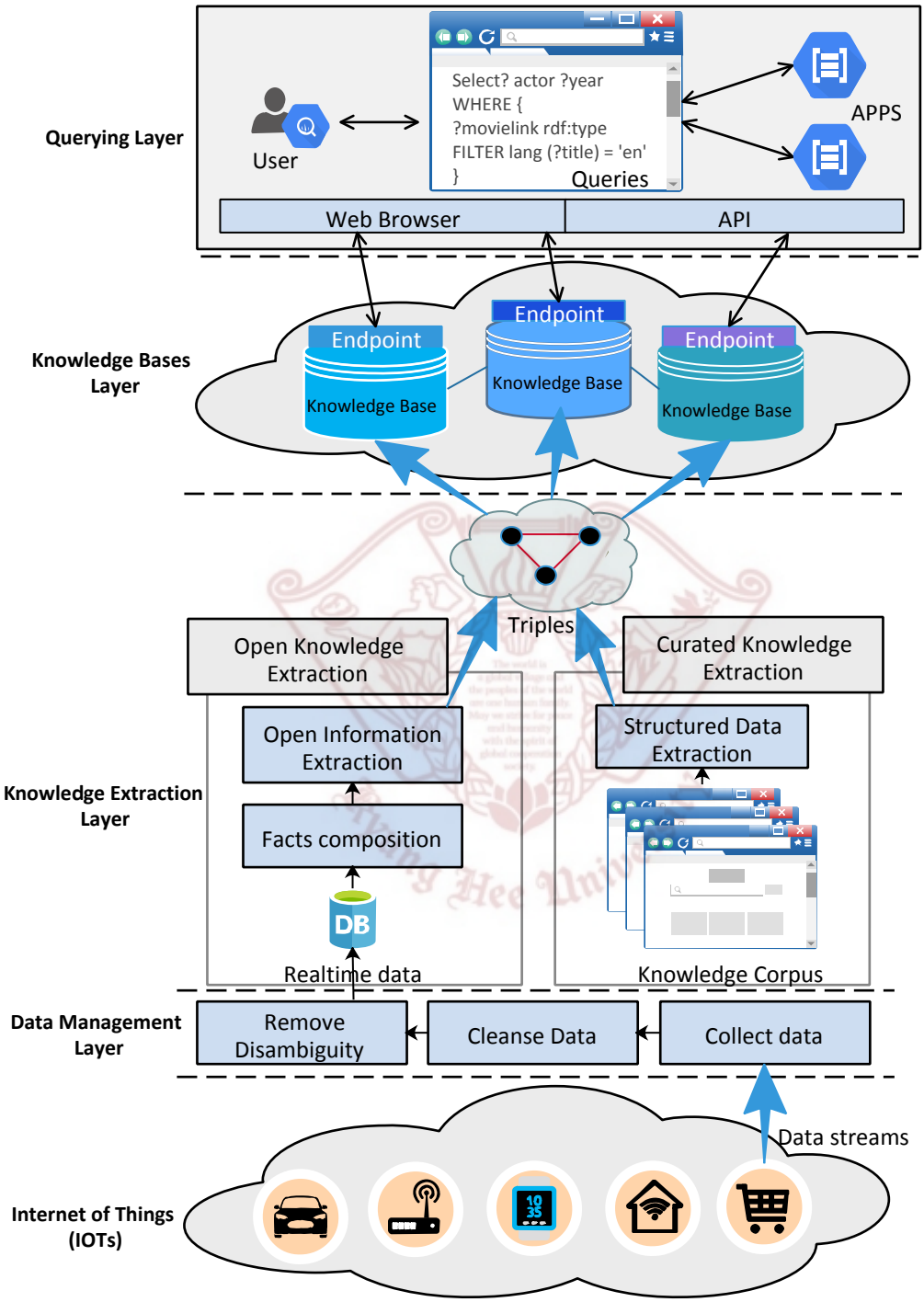


Figure 1.1: Process of Storing and Querying Linked Open Data (LOD)

shared on the cloud. Modern stream analytics applications [14,15] are exploiting the fusion of data streams and Linked Data publicly available on the Web. Recent investigations [16–22] have shown that the content of Linked Data is dynamic in nature and keeps on evolving. Linked Data content changes over time and some of the data stored in the local cache becomes outdated. Therefore, knowledge about what has changed in the database is important for analytics applications [1]. Analytics applications need constant updates to guarantee the quality of service in maintaining the local cache. To the extent of our knowledge, there is limited work addressing the problem of maintaining local views (or caches) up-to-date [1,9,19,23–27]. Normally a maintenance policy is needed to determine what to update and when. In the literature, three kinds of conventional maintenance policy have been reported for data analytics systems [28]: immediate (update immediately after data arrives), deferred (no execution is performed on current query evaluation), and periodic (update the local views on regular bases). However, these conventional policies fail to effectively optimize local views for linking data due to slow response time. Figure 1.1 illustrates the overall process of querying Linked Open Data, where the knowledge base layer uses the RDF as a data representation model and the querying layer is responsible for conveying the SPARQL to interface in order to return the results.

Recent investigations [16–22] have shown that the content of LOD is dynamic over time and continuously evolving. The accuracy of the index drops [29] due to the continuous evolution of LOD. Knowledge about the changing behavior of the LOD cloud is extremely important for applications consuming these data [1]. The LOD application pre-fetches the data and stores it in its cache for future information needs. The LOD application utilizes caching techniques to leverage the query processing and serve the requests from its cache (also called *cache hits*) [30]. Many caching techniques have been developed for relational databases such as LRU [31] and LFU [32]. The underlying structure of the LOD cloud is different from the relational databases. These caching algorithms designed for relational databases are not applicable in the LOD scenario [33]. Linked Data applications need constant updates to guarantee the quality of service and maintain up-to-date copies of the data. In the ideal case, an application is needed to visit all the data sources. However, in the real world scenario, this is not feasible due to limited computational resources. To the extent of our knowledge, we believe, there is limited work addressing the

problem of updating local LOD caches [1, 9, 19, 23–27].

Existing approaches [1, 16, 34] that are dedicated to capture changes in LOD utilize *HTTP header information*. The header information provides information concerning when the source in the LOD cloud was changed last. Recent analyses [29, 35] have shown that application relying on the *last-modified HTTP header information* is inappropriate and susceptible to drawing incorrect conclusions. Alternative strategies [23, 24] have utilized scheduling and explore different features such as such as *Age* [36] and *Size* [37] to assign the preference in order to visit the resources. However, existing scheduling methods [23, 24] are not as effective as they do not consider the importance of the preference score while conducting the cache update. Similarly, the cache mechanism of existing methods is computationally expensive; instead of only replacing with changed items, the existing methods perform a full cache replacement. Recently [23], a crawling strategy called Triple Linear Regression (TLR) has been proposed for RDF documents. However, due to the poor accuracy of this approach, it can not be utilized in a real-world LOD application. The opening chapter will contain the main motivation of this thesis in Section 1.1, the problem statement along with the research question in Section 1.2, an overview of the existing approaches in Section 1.3, and the proposed solution in Section 1.4. Finally, the key contributions of this thesis in Section 1.5.

1.2 Motivation

The heap of structured data published over the Internet is increasing i.e., Linked Data [20]. Linked Data is a global information space for representing and connecting data structurally. The format of Linked Data is encoded as RDF[5] which consists of the subject, predicate, and an object and is stored in the Triplestore[6]. RDF is widely used as an information model for vast semantic data. However, in the RDF data model, the querying complexity is higher than the relational data model. As the SPARQL[7] is standard language to query RDF dataset. To access the data, the SPARQL service is deployed on each knowledge base which uses the HTTP bindings as shown in Figure 1.2.

[5]https://www.w3.org/RDF/
[6]https://jena.apache.org/
[7]https://www.w3.org/TR/rdf-sparql-query/

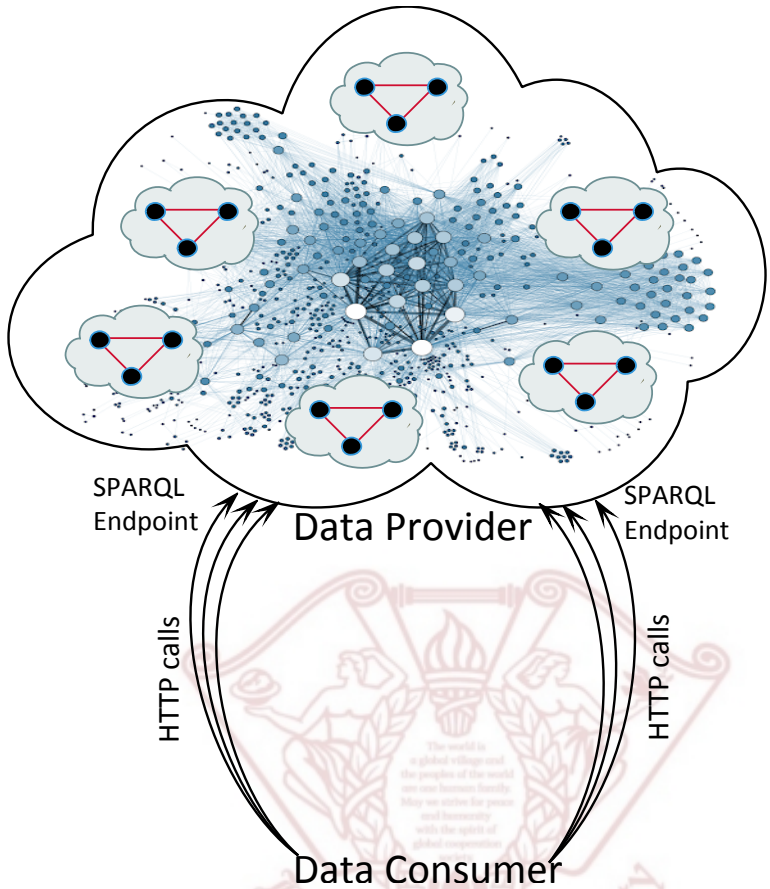


Figure 1.2: Showing the Process of Querying Linked Open Data, where Server perform all the Processing

The main part of the SPARQL language is Web Services Description Language (WSDL)[8] that describe the means for conveying queries and results to the processing service. Currently, widely used RDF datasets such as DBpedia[9] produces abundant request from diverse applications [11]. Nowadays, the amount of semantic data is growing rapidly, therefore for efficient query processing and caching [11] is required. So caching is used to leverage the query processing on the Triplestore and the data is present in its cache the request is sent immediately (also called *cache hits*) [30].

Many caching techniques have been developed such as LRU [31] and LFU [32] for relational databases. The underlying structure of the big semantic data is different from the relational databases. In recent years, a lot of non-relational Triplestore [38] are emerging. The caching al-

[8]https://www.w3.org/TR/rdf-sparql-protocol/
[9]https://www.dbpedia.org/

gorithm design for relational databases does not apply to Triplestore [33]. In the RDF triplestore, some of the records are ”hot” (frequently accessed by the application) and others are ”cold” or seldom accessed. The performance depends on the number of factors such as hot records in the cache or residing in the memory for fast access [10]. The Linked Open Data (LOD) cloud is dynamic the content of these LOD changes frequently. As mentioned previously, the existing approaches are unable to cope with the dynamic and evolving nature of the Linked Open Data (LOD) cloud. To utilize the best resources, it is essential to improvise an effective scheduling strategy for updating the local data cache. However, LOD applications that consume these data need to be aware of the changes. As the content stored locally is outdated and needs to ensure the most recent version of the data. In the ideal case, the index needs to be updated continuously to maintain up-to-date. However, in the real world scenario, the LOD applications must deal with the limitation of the computational resources. These limitations imply to prioritize which data sources need to be updated. Our work is motivated by the need for efficient query processing in the Triplestore [39]. These considerations drive our research:

(1) Access Workload: The performance of the Triplestore is a major challenge in real-world practical application. The workload exhibits considerable access skew, for example, the product description in the online store exhibits natural skew as most of the items are popular and frequently accessed than others [40].

(2) Overhead in caching: The major problem of cache is high overhead due to the proactive fetching [41, 42]. For example, cache policy such as LRU encounters 25% overhead on every record access [33].

There are numerous applications that are related to the change detection and notification services such as [43], which estimate the change frequency of the web and to improve the incremental web crawler. Identified solution strategies such as *HTTP Meta-data Monitoring* [25] uses the header timestamps to detect the change in the data. Another solution strategy to estimate the change is Dynamic Linked Data Observatory (DYLDO) [44] which is fetching the entire content and determining locally. Hence, the existing solution strategies are unable to cope with the scalability and dynamics of the LOD cloud in an effective way.

There are various implementations that looked into the characteristics of the LOD cloud. Some

have conducted the structural analysis in order to obtain the characteristics of the data [45]. In literature, there are various works that have investigated on the characteristics of LOD and estimation of the change. Researchers in [46,47] have estimated the change frequency of data to improve the web crawlers, web caches and to help the data mining tasks. Other works are related to change in content of a RDF documents that is crawled for the period of 24 weeks. According to the author Etag and Last-modified HTTP header were applied to typically indicate the change. There is a variety of existing works related to the change detection of the query results on dynamic data sets. Among them the most prominent work on the query caching is [11], but the working implementations are rare. Currently, there is a limited research that focuses on the impact of the cache on LOD. Most of the available work is rich in database literature where query cache occurs; however cache with the SPARQL engine is not considered relevant [48].

Since, LOD cloud is a global information space and it is structurally connect data items. The distributed web based nature of data motivates many application to keep local copies of the data. Due to the dynamic nature of the linked data many applications need to keep updating the local copy of the data. The main problem is when to perform the updates. To solve this problem researchers have investigated scheduling update strategies to periodically update the LOD caches. We also investigate update strategies that are proposed in the literature for updating linked data caches. Another worth mentioning work by Magnus Knuth et al. [26] discusses the problem of scheduling refresh queries for large number of registered SPARQL queries. They have investigated various scheduling strategies and compared them experimentally. The main contribution of their work is an empirical evaluation on the real world SPARQL queries.

The issues related to the exploitation and maintenance of local views received considerable attention in the research community over recent years. Most of the maintenance algorithms proposed in the literature are based on *eager (or deferred)* maintenance. Eager maintenance protocols update the view periodically whenever a change occurs in the cloud. Colby et al. [49] proposed deferred view maintenance to reduce the view downtime and perform incremental maintenance to keep the local views up-to-date. However, all these algorithms divide the maintenance task into smaller steps, which is not as effective as combining the small maintenance task to improve the efficiency of the overall maintenance process. Data warehouses have also used a *lazy mainte-*

nance [50] policy that delays maintenance until downtime rather than the peak query time. Also worth mentioning are the maintenance policies for stream monitoring that examine the problem to schedule multiple continuous queries. They are able to manage incoming data streams using deferred maintenance for updating local data caches. However, their policies mainly focus on ETL (Extract, Transform, and Load) operations and they have difficulties in managing latency and accuracy when a large-scale fusion of linked datasets is given.

Semantic caching was originally proposed for Database Management System (DBMS) [51] to reduce the overhead when retrieving data over a network. The idea is to maintain has been extended to previously issued queries to facilitate future requests. Nowadays, the semantic caching technique triplestores [11]. Godfrey et al. [52] proposed a notion of semantic overlap and introduced a caching approach that utilized client-server systems. To extend this idea, Dar et al. [51] proposed a semantic region-based caching and introduced a distance metric to update the cache where the far regions are discarded from the cache. However, these approaches can only handle *SELECT* SPARQL queries. Martin et al. [11] proposed an idea to apply cache on the SPARQL processor. The benefit of this work is to cache both the triple query result and the application result. However, this work does not consider identical and similar queries for cache replacement. To extend this work, Shu et al. [53] proposed a content-aware approach that utilized *query containment* to estimate whether the queries can be answered from the caches. The containment checking approach produces a lot of overhead, therefore this approach is not widely utilized by the Semantic Web community. Yang et al. [54] proposed an approach to decompose the query into basic graph patterns and cache intermediate results. But they do not consider cache replacement to maintain the freshness of the cache. In summary, only a few works have been reported that deal with the problems related to view maintenance. Moreover, the existing solutions are unable to provide a proactive maintenance policy therefore reducing query time.

There is a need of user-friendly techniques to represent, query, and visualize the linked data. Healthcare organization such as World Health Organization (WHO), publish health related data online. However, most of the data are either available in Excel or PDF format by the portal. On the other hand, linked data provides an efficient way to publish the data and alleviate many challenges.

Figure 1.3, shows the typical architecture of LOD based application which provides the flex-

ibility and usability where the data consumer such an expert is able to explore and visualize the data. Whenever the change occurs in the sources the scheduling strategies should keep the local copy of the data up-to-date. The main core of the architecture is the service and the model layer where the external sources are converted into the RDF triples and stored in the triplestore. As the data keep on changing the effective change estimation is needed to update the local copies of the data considering the limitation on the bandwidth. The purpose of this work is to evaluate the effectiveness of these scheduling strategies.

In contrast with the above-mentioned approaches, this thesis presents change-aware scheduling for effectively updating LOD cache. In contrast to existing approaches [23, 24], our novel scheduling utilized a change metric together with a weight function that assigns more importance to recent changes in the dataset. Moreover, our proposed update policy incorporates the previous estimation of changes and establishes when the local data cache needs to be updated. The motivation behind the change-aware scheduling approach is to maintain the local data cache up-to-date for faster querying and processing. In order to achieve these goals, this study was undertaken with the following objectives (i) to select the change metric that quantifies the changes for the LOD cloud (ii) to select scheduling that assigns a preference in order to visit LOD sources (iii) to keep the local data cache up-to-date by replacing it with changed items.

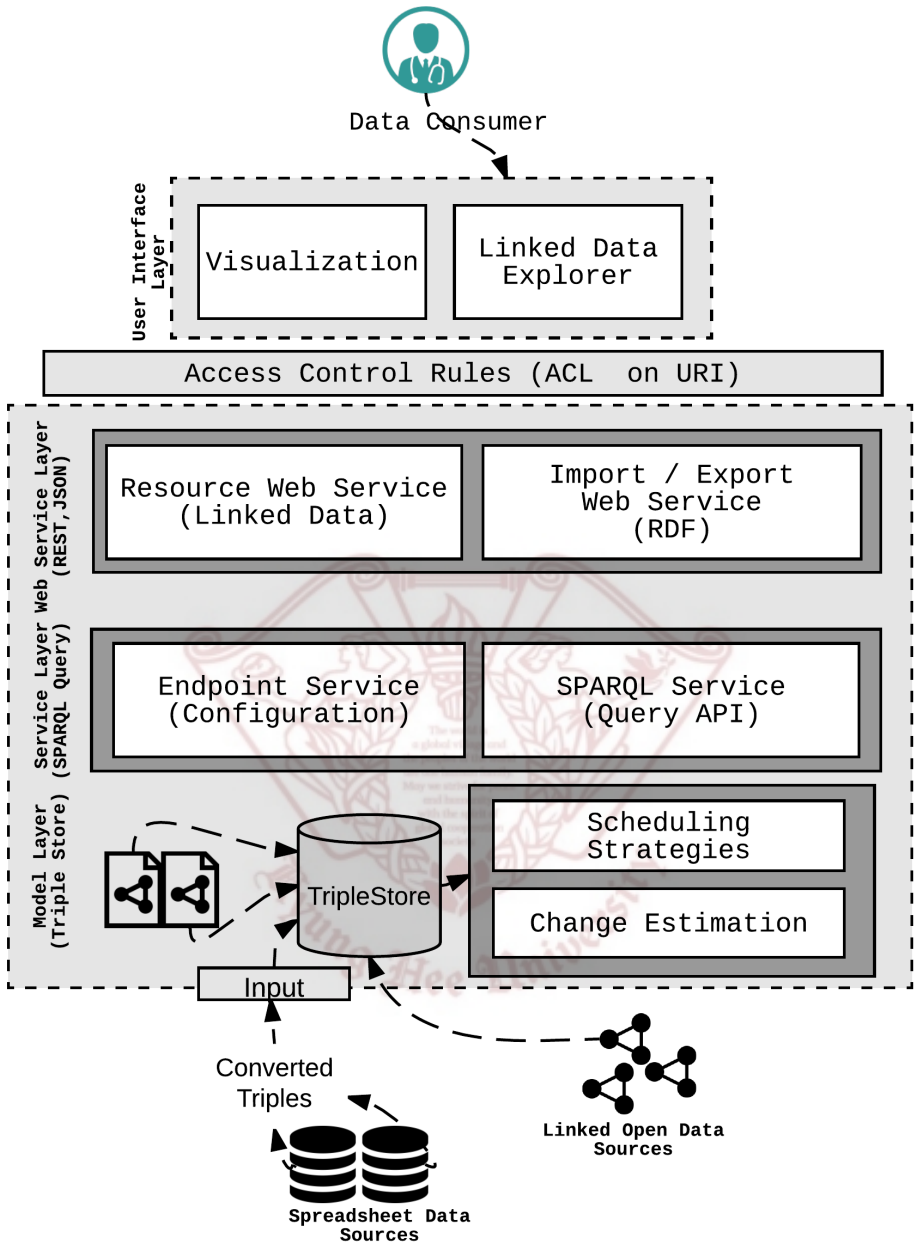


Figure 1.3: Typical Workflow Diagram of Medical LOD application from Querying, Conversion and Visualization

1.3 Problem Statement

The proliferation of semantic big data has resulted in a large amount of content published over the Linked Open Data (LOD) cloud. Semantic Web applications consume these data by issuing SPARQL queries. The problem with querying the LOD cloud is its high search latency and slow performance of retrieving data from its repositories. The rapid expansion of LOD use in academia and industry evidences the efficient retrieval of data as one of its major challenges. Although every LOD cloud supports SPARQL queries to access data from its publicly available interfaces, a central problem is the lack of trust regarding these endpoints due to network instability and latency. Therefore, the typical solution is to dump the data locally and maintain endpoints to process these data. The data stored at the local endpoints are not up-to-date and require constant updates, therefore, accurately hosting the endpoints requires expensive infrastructure support. Whenever there is public data available, data consumers want to query it, and nothing is more compelling than querying the vast amount of the Linked Data. Recently, Linked Data has emerged as one of the best practices to represent and connect these repositories, also allowing the exchange of information in an interoperable manner. Linked Data not only supports the integration of multiple data from diverse sources but also provides a way to query these datasets. With over 800 million triples are currently stored in DBpedia[10], the search of the specific resources has never been this high before [11].

In spite of the increased performance of SPARQL endpoints the main problem remains due to the low availability of these endpoints as on average the downtime of SPARQL endpoints is more than 2 days each month [55]. A survey conducted on 427 public SPARQL endpoints registered on the DataHub shows the low efficiency of these endpoints with availability rate above 90% [55]. Oftentimes, SPARQL endpoints are not processed the specific workload as efficiently. The major challenge faced by querying the SPARQL endpoint on account of the inherently distributed nature of Linked Data is its high search latency and lack of tools to connect SPARQL endpoints. Accessing Linked Data cloud at query time is prohibited due to high latency in searching the content and limited capability of tools to connect to these databases. Therefore, the performance of retrieving data from Linked Data repositories is one of the major challenges. Although the Linked Data cloud

[10]https://www.dbpedia.org/

supports SPARQL queries to access data from its publicly available interfaces, a central problem is the lack of trust regarding these endpoints due to network instability and latency. Therefore, the typical solution is to dump the data locally and maintain endpoints to process these data. The data stored at the local endpoints are not up-to-date and require constant updates, therefore, accurately hosting the endpoints requires expensive infrastructure support [56].

Existing research shows that the SPARQL server with high demands is often hard to host and which is further complicated as the endpoints are publicly hosted due to the unpredictable workloads [57]. The current de-facto way of querying the SPARQL endpoint is to utilize the HTTP that is implemented on top. The client sent the request through these endpoints and the server returns the request. Due to the massive data involved servers need to execute a significant amount of work [58]. The SPARQL query processing is different than the regular HTTP processing as the query-based partitioning of resources occurs. Therefore, regular HTTP caching strategies can not be fully applied to the Linked Data scenario. In the recent past, many efforts have been made to improve the performance of effectively querying Linked Data [59, 60], as compared to querying a triple stored in a relational database, querying a triplestore is still slower by 20% [11]. To circumvent the problem of effectively querying Linked Data [61, 62], caching is the most popular technique to reduce query time by serving the requests from a cache. The idea of query caching is to reuse the previously issued queries. In the distributed setting the benefits of caching are more evident, as the previously issued queries are stored on the remote source. The caching will immediately improve the robustness of the system where the remote resources are not available due to the network insatiability. Existing caching approaches consider caching of the entire query result which means that similar queries can not be served from cache. These research questions are individually tackled in this thesis as follows.

1. **Access Patterns**. How to identify the access patterns of similar structure queries to identify the common access strategies? At the same time, numerous research have been conducted on LOD [63,64], but little work addresses the problem of discerning Linked Data access patterns. Recently, query logs are increasingly used to harvest query results from available RDF data repositories. These logs are useful to identify the previously issued queries. To evaluate the access patterns, we have contributed to the notion of Linked Data query similarity. As

a vast amount of data, especially in Biomedical research [63, 65], are being published as Linked Data. Being able to analyze these data is essential for creating new knowledge and decision making. Many of the current LOD based analytic applications require continuous access to these data sets. The problem with continuous access is that it requires a lot of computational resources which makes it unfeasible to perform in system peak time. In order to utilize the best resources, improvised scheduling is required to maintain the freshness of the local data cache. As cache has limited space, it is important to fill it with valuable content by replacing unnecessary content. To reduce this overhead cost, modern database systems maintain a cache of previously searched content. The challenge with Linked Data is that databases are constantly evolving and cached content quickly becomes outdated.

2. **Query prefetching**. How can we assist data consumers to efficiently retrieve the data from LOD and prefetches the additional facts related to the query? As discussed earlier, the LOD patterns have been addressed briefly in the existing literature. Therefore, there have been few efforts aiming to retrieve information related to the LOD [39, 66–68]. Query prefetching aims at discovering interesting related information based on the user request. There exist a number of projects that include implements the query prefetching for retrieving the data [69]. The main benefit of query prefetching is to increase the cache hit rate [70]. The idea behind query prefetching is to retrieve the results of the queries as they are requested by the user. The problem of identifying the subsequent queries is quite challenging, if the results of the queries are prefetched the user is served from the cache. This immediately corresponds to the reduce the result of the response time as more queries are served from prefetching the result of previously issued queries.

3. **Cache Replacement**. What are the current obstacles in replacing the cached content and how can they be alleviated? The volume of Semantic Big Data is unprecedented and keep on evolving. In recent years, many efforts have been devoted to the problem of effectively replace the cache in order to reduce the query time by serving the request from the cache [71]. As the cache is limited, therefore it is important to fill it with valuable content. Currently, a lot of existing research is replacing the content based on time. To the extent of our knowledge, we believe, there is very limited work addressing the problem of efficiently replac-

ing the cached content. Data on the LOD frequently changes and applications relying on the local copies need constant update their cache. Therefore, instead of visiting all LOD sources, a maintenance policy helps to know when data from the LOD cloud need to be updated [23, 30].

The solution to the problems and challenges is to adopt client-side caching as it is a domain-independent approach that does not require underlying knowledge of the LOD. Typically, the queries issued by the end-user are repetitive and follow similar patterns that only differ in a specific element. The major challenge of this task is to find similar queries, as it is possible that two queries are structurally similar but may differ in content. To tackle this problem, we propose the use of a bottom-up matching approach to find similar queries. For the structural similarity, we first compute the distance between the triple patterns and prefetch the results of similar queries to be placed in the cache. A cache has limited space, therefore it is advantageous to replace it with frequently-accessed data. In this work, our cache replacement utilizes exponential smoothing forecasting to calculate the frequency of the accessed data and replace content based on access frequencies. More specifically, we propose a full-record replacement strategy, in which at every new query the hit frequency of accessed triples is calculated using exponential smoothing, and the cache is replaced with the highest access queries. The motivation behind adaptive cache replacement is to improve the querying efficiency and reduce the burden on the SPARQL endpoint. Repeated queries are cached locally, and the results of these queries are immediately answered to the user. Our approach optimizes the results of predicted potential queries and less-valuable queries are replaced from the cache. Our approach is based on the idea that the clients who processed similar queries are likely to process similar queries in the future. The aim is to search and gather content possibly requested at the future queries by identifying the concepts of the previously issued queries. The main benefit is to reduce the transmission overhead and improve the hit rate and query time by retrieving contents for future queries at once. We prefetch all contents that were used to answer the queries since those queries are more likely to be requested by the same user in the future. For cache replacement, we serve each query according to the estimated frequency, the query with the highest frequencies are kept in the cache. Figure 1.4 illustrates the overall flow of the proposed approach. When a client sends a new query the cache manager first check if an identical query has been cached. In this

case, the results are immediately sent to the client. In case of the cache miss, new queries are sent to the query prefetching where similar queries are suggested and the result of these queries are retrieved from the SPARQL endpoints. As an offline process, the result of a similar query is placed in the cache for future access queries. The cache replacement process is triggered when the cache is full and it runs on a separate thread that does not affect the query answering process. The cache replacement is based on the frequency, the higher accessed queries are placed in the cache.

1.4 Key Contributions

The goal of this research is to provide a methodological framework for utilizing cache to query Linked Open Data (LOD) cloud. In particular, the main objectives of this research is to: develop a change metric that is utilized by the scheduling to update the local data caches, formulate the approach to cater to structural and content-wise changes in the LOD cloud, prepare methodology to handle change detection, change collection a change formulation and finally establish the criteria when the LOD cache need to be updated. We have prioritized the recent changes using a weight function, that assigns importance to the recent changes. We have considered a behavior change of LOD as an essential factor and utilized the weight function to assign importance to recent changes. Our cache replacement incorporates the previous estimation and establishes when the local data cache needs to be upgraded.

When a user sends a query to the available endpoint, the system will check in case of the previous cache result. In case of similar queries, the request is immediately returned from the cache module. After recording the query, the offline process starts to group similar structure queries. The result of frequently access queries is placed in the cache. As cache has a limited size, therefore, cache replacement is executed when the number of cache queries is reached, and based on the frequency of the accessed data it replaces the content of the cache. We proposed a frequency-based cache replacement policy to update the cache. The query record will keep track of the accessed queries and send for offline analysis to calculate the access frequencies by using exponential smoothing. As compared to the existing approaches [23, 24], our proposed approach has the flexibility to incorporate other change metric [72]. In this approach, we have considered the behavior change of LOD as an essential factor and utilized the weight function to assign importance

to recent changes.

1.4.1 Dynamic Change Model

The LOD is constantly evolving and the applications using the LOD may face several issues such as outdated data. Therefore, we have proposed a change model to capture the changes and update the local data caches. As compared to the existing approaches, we explore the notion of the prioritization regions for change detection with the aim of optimal accuracy for the efficient use of the computational resources. Our model captures the information of both changed resources and triples in linked datasets.

1.4.2 Structure and Content based Query Similarity

In the LOD cloud, most often users request similar resources and the queries are almost similar. The aim of a query similarity is to compute structure similarity based on the distance score. For the structure similarity, we proposed to utilize the bottom-up query matching approach to detect changes between triple patterns occurring in consecutive queries.

1.4.3 Query Augmentation for Exploratory Query Prefetching

Instead of issuing similar queries, our approach is based on prefetching. We proposed exploratory prefetching to retrieve additional content related to central concepts. The aim of the query prefetching is to reduce transmission overhead and retrieve useful information for the future information need. Therefore, our prefetching retrieves additional data related to future queries. The goal of this phase is to reduce the overhead of similar queries run over KBs.

1.4.4 Frequency-based Cache Replacement

The cache replacement aims to replace the less valuable cache items. However most of the existing approaches produce additional overheads while replacing the cache contents. We proposed frequency-based cache replacement that scan the access log to replace the cache. Exponential Smoothing (SE) is applied to rank each query according to the estimated frequency, highest accessed frequency are kept in cache. Thus, our work benefit the triplestore in replacing the cache.

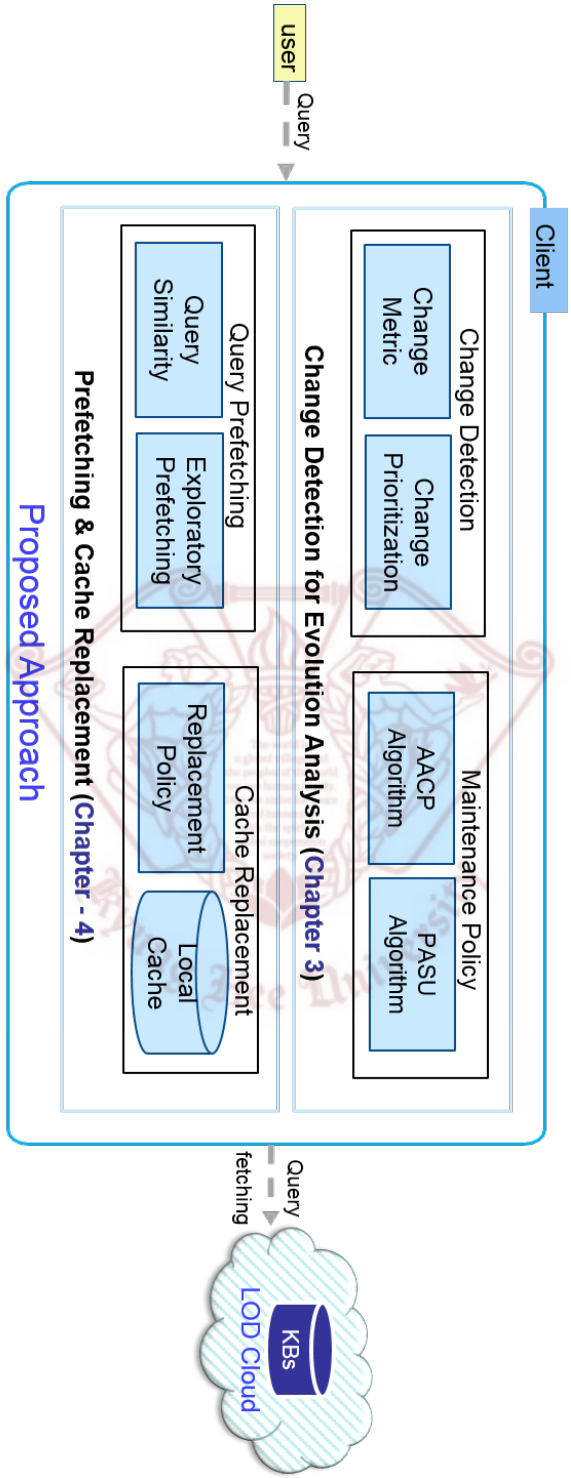


Figure 1.4: Idea Diagram of the Proposed Research Studies with Chapters Mapping

1.5 Thesis Organization

This dissertation is organized into chapters as following.

- **Chapter 1: Introduction.** Chapter 1 provides the brief introduction of the research work on utilizing caching to query LOD cloud. Therefore, instead of running repeated queries, results are provided from the local views thus improving overall response time. It focuses on the problems in the areas, the goals to achieve these problems, and finally the objectives achieved in this research work.

- **Chapter 2: Background and Related Work.** A background detail is provided in this chapter about dynamic optimization of cache replacement in querying LOD cloud. Finally, it provides a comparison of these systems with the proposed system of the research thesis to reflect the limitations of current systems addressed by the proposed system.

- **Chapter 3: Change Detection for Evolution Analysis.** A proposed solution in the form of the change metric is provided to deal with the structure and content-wise changes in the LOD cloud. The main benefit of this change metric is that instead of issuing the similar structure queries this metric will keep the similar structure queries in the cache and serves the answer from its local cache.

- **Chapter 4: Prefetching and Cache Replacement.** we propose a proactive maintenance policy to update the local view by issuing the maintenance jobs during system idle time. It achieves the desired query performance by doing the maintenance ahead of query evaluation time. This module works as an optimizer to accelerate the overall query processing by prefetching the data and storing it in the cache for future queries. Our maintenance policy postpones the update until the system has a freecycle. Therefore, the result of these similar structure queries is placed in cache to increase hit rates, while alleviating the burden on the querying endpoints.

- **Chapter 5: Evaluation and Results.** The results and evaluation of different techniques used in the proposed framework are highlighted in this chapter. We have evaluated the effectiveness of our approach based on the precision and recall using a real dataset.

- **Chapter 6: Conclusion and Future Directions.** This chapter concludes the thesis and also provides future directions in this research area. The main contribution of the thesis is also highlighted in this chapter.

Chapter 2

Background and Related Work

2.1 Semantic Web Foundations

According to Tim Berners Lee, the semantic Web is an extension of the current Web in which information is given in a well-defined format, that enables the computer and the people to work in cooperation. Based on the vision, the semantic web is considered as the next step in Web evolution that enables the machine to process and transform the data in a variety of ways. The traditional World Wide Web (WWW) is a global information space that provides links to documents. These Webs consist of a graph structure that includes the Web pages connected with the hyperlinks. Most of the Web page content is published in Hypertext Markup Language (HTML). The special way to referencing other documents by indicating with the HTML tag $< a >$ which provides the relationship with the other document of Web.

The Semantic Web is an extension of the traditional Web that facilitates the linking of the documents [73]. As described by Tim Berners-Lee [4], the Semantic Web enables the machine in such a way that data can be searched, interpreted, and reused. Linked Data is another important concept in the Semantic Web, enabling the machine to browse the Web of data, such as DBpedia[1]. Linked Data is collaboratively built from the Web corpus to represent knowledge in a structured format that greatly facilitates the sharing of information around the world. Linked Data describes the interconnection of Web pages*URLs(Uniform Resource Locator)*. Therefore, facilitating the linking of the data from diverse sources. The Semantic Web is mainly composed of, Knowledge Bases (KBs) such as Freebase [74], DBpedia [75] and Yago [76]. These KBs represent data as Linked Data according to a predefined schema, known as Resource Description Framework (RDF). The RDF is considered the standard representation for Linked Data, where relationships are represented in the

[1] https://wiki.dbpedia.org/

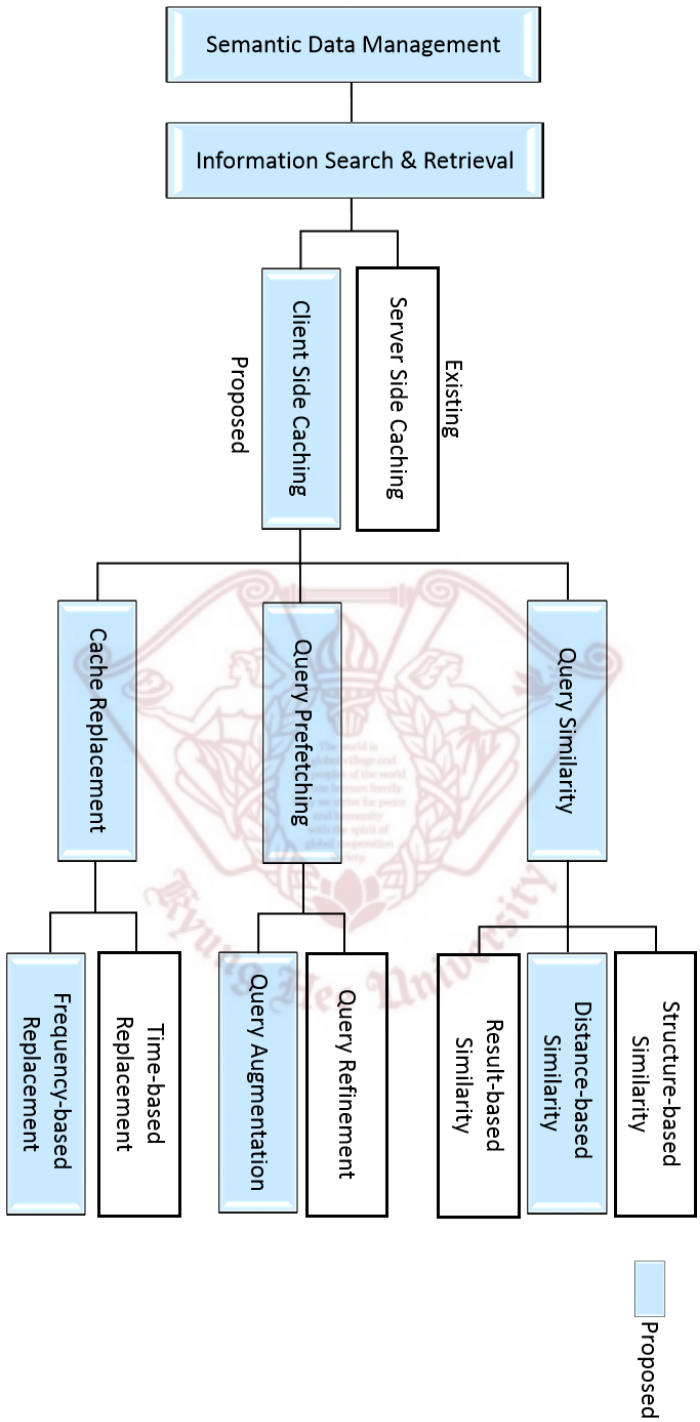


Figure 2.1: Showing the Research Taxonomy of the Proposed Work

form of triples, i.e., (*subject, predicate, and an object*).

SPARQL[2] is a widely used standard query language to retrieve and manipulate data that are stored in RDF format. SPARQL is a structured query language standardized by the W3C for querying RDF triplestores[3]. A SPARQL query can be further decomposed into Basic Graph Patterns (BGPs) and the results are represented as a hierarchical tree. The syntax of the SPARQL query contains different and disjoint query types such as SELECT, CONSTRUCT, ASK, and DESCRIBE. The Linked Data cloud also provides a SPARQL endpoint for their datasets. However, querying SPARQL endpoints is cumbersome due to network instability often the connection to these endpoints is temporarily lost, affecting query performance. These endpoints do not provide any information about dataset modification. Therefore, long-running data analytic applications must resubmit queries for keeping the local data cache up-to-date.

The benefit of the Semantic Web is that it provides data which is understandable by the machine as it can be reused and interoperable in many different Web sources. Moreover, it facilitates the discovery of the Web in a better way supports the knowledge of exchange between users. Clearly, the web is made of many documents, and metadata of the given Web document is the data that is used to describe the document. This includes the title of the document, the date of the document, and also the authors of the documents. It is not possible for automated agents to process all these data in a uniform way, therefore some metadata standard is required and this standard is used to describe the data. However, this metadata information is not displayed in the Web browsers. The overall flow of the research taxonomy is shown in Figure 2.1.

An increasing amount of the statistical data is published on the Linked Open Data (LOD) cloud. Getting insights from the data in more intuitive ways are becoming important. Systems for the Semantic Questions Answering (SQA) plays a vital role to connect with linked open data and provides an intuitive interface by translating natural languages queries into SPARQL syntax. Statistical data need more advanced querying methods to empowers non-experts users to draw their own conclusions. Semantic question answering is extremely important in the following application involving Linked Data to access public data sources.

— **Healthcare and Life Sciences (HCLS):** Statistical data in the form of the RDF data cubes

[2]https://www.w3.org/TR/sparql11-overview/
[3]http://www.w3.org/TR/rdf-sparql-query/

influences decisions in a domain such as health care and life sciences. Many clinical datasets are often composed of the numerical observations as well as statistical information such as clinical trial data which is often composed of patient attribute [63, 77, 78].

— **Biomedical Question Answering:** In biomedical, workers want to express their information needs in natural language. BIOASQ [79] encourages the participant to adopt semantic indexing as a mean to combine the information from the multiple sources of different types such as biomedical articles and ontologies. But, this system typically lacks supports for the RDF data Cubes, where clinical data represented in the form of multi-dimensional data [80].

We have motivated the need of the semantic question answering, where statistical data in the form of the RDF data cubes. However, there exist some important challenges that need to be tackled by the Semantic questioning answering system, including the following:

— **Lack of processing RDF cubes by SQA systems:** One of the major limitations of SQA system is a lack of processing of the statistical data in the form of the RDF data cute. Statistical data is different than other data and can not be queried by the existing linked open data querying approaches. However, the current SQA system provide translating natural language into SPARQL, which is a native language to query the RDF knowledge bases [81–83]

— **Enabling Access Over Statistical Data:** Current query federation approaches enables the integration of the multiple data sources but they do not consider the methods to access the statistical data while maintaining the good performance [79].

Although, there are a number of benefits to publishing data in multi-dimensional, such as statistics in Linked Open Data (LOD) cloud using LOD publishing principles. First, the data become web addressable and allow a consumer to annotate and link the data. Secondly, data can be flexible and combined with the other data using Linked data technology. Finally, data can be reusable and access by using the SPARQL, one of the example is linkedspending[4], which contains government spending from all over the world as linked data [84].

[4] `http://linkedspending.aksw.org/`

2.1.1 Resource Description Framework (RDF)

The Resource Description Framework (RDF)[5] is a basic building block for supporting the Semantic Web and it is used to establish the model for providing the information about the Linked Open Data resources. The basic rule for publishing the resources on the internet consist of representing the resources using the URIs and provide detailed information when looking up URIs [20]. The RDF is capable of proving the basic facts of any resource such as exchanging and reusing the structured metadata, as it supports the interoperability among the applications. The RDF allows to model facts and statements about the entities as:

$$RDF := (s,p,o)\varepsilon(U) * (U) * (U \cup L) \tag{2.1}$$

In Eq. 2.1, where U belongs to the URIs(HTTP) and L is the literals which can either be the string and the s,p,o is commonly named as the $subject, predicate$, and $object$. In the RDF, the resource is standard for metadata as it offers a way of specifying anything which in the RDF is called the resource. A resource, therefore, describe by the RDF expressions as could be a Web page, real-world object, or anything. The *property* is a resource that is used to describe a specific aspect such as the attribute or relation of a given resource. Finally, the statement is used to describe the RDF statement of the resources as the property value can be a string or literals such as $<$ $subject$ $>$ has property $<$ $predicate$ $>$, whose value is $<$ $object$ $>$. These statements are used to express the knowledge in a simplified way. Figure 2.2 shows the example of the RDF document in the XML deceleration and the XML declaration is followed by the $<$ $rdf:RDF$ $>$ that indicates that this is an RDF document. The $<$ $rdf:Description$ $>$ contains the elements that describe the resource. Therefore, the resources being described in this document is identified as an artist as RDF provides a predefined value to describe the particular resources.

The main goal of the RDF is to provide data in a way that machines can interpret. However, this information is often serialized in a way that it can be loaded triple store to further process. There are many serialization formats and XML is considered as a default format [85]. Additionally, an XML document can be cumbersome to parse due to the increasing size and especially if the user is trying to get the basic summary. Thus, there are several alternatives format have been proposed

[5] https://www.w3.org/RDF/

```
<?xml version="1.0"?>
<rdf:RDF
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:si="https://www.w3schools.com/rdf/">
<rdf:Description
rdf:about="http://www.recshop.fake/cd/Empire Burlesque">
  <cd:artist>Bob Dylan</cd:artist>
  <cd:country>USA</cd:country>
  <cd:company>Columbia</cd:company>
  <cd:price>10.90</cd:price>
  <cd:year>1985</cd:year>
</rdf:Description>
</rdf:RDF>
```

Figure 2.2: Example of RDF Document

such as N-Triple format [86]. In this format, a triple is separated with the white space, and each line of the syntax is terminated with the full stop. Where the end of each line indicates the end of the statement. This format is widely used as it is easy to add new facts in the corresponding line.

2.1.2 SPARQL Query Language

The SPARQL Protocol and RDF Query Language (SPARQL)[6] s considered as a structured query language standardized by the W3C for querying RDF triple stores[7]. and it is considered as a graph-matching query language. SPARQL can be used to access diverse data sources where the data is stored as RDF. The result of the queries issued in the SPARQL can be the result set of RDF graphs. The result of the SPARQL consists of a pattern that contains the RDF terms as $subject$, $predicate$, and $object$. The query result is matched with the given dataset and these terms are obtained from the matching process, however, to extract the values from the endpoints, *SELECT* is widely used.

The syntax of SPARQL consists of the five main parts: (i) prefix declaration that define the URI, (ii) dataset clause that considered the datasets which the query will run and provide the results, (iii) query forms which considered what type of the queries will run most common are *SELECT*, *CONSTRUCT*, *ASK* and *DESCRIBE*, (iv) Query clause that specifies that binds the data to the against the query, (v) and finally solution modifier that projects the ordering of the result. A brief description of these query forms are given below:

[6]https://www.w3.org/TR/sparql11-overview/
[7]http://www.w3.org/TR/rdf-sparql-query/

```
1  # PREFIX DECLARATIONS
2  PREFIX : <http://dbpedia.org/resource/>
3  PREFIX dbpedia2: <http://dbpedia.org/property/>
4  PREFIX dbpedia: <http://dbpedia.org/>
5  # DATASET CLAUSE
6  FROM <dbpedia: <http://dbpedia.org/>
7  # RESULT CLAUSE
8  SELECT ?philosopher1 ?philosopher2
9  # QUERY CLAUSE
10 WHERE {
11   ?philosopher1 foaf:name "Auguste_Comte" .
12   ?philosopher1 ?relationshipWith :Paris .
13   } UNION {
14   ?philosopher2 dbo:influenced ?philosopher1 .
15   # SOLUTION MODIFIER
16   OPTIONAL {
17   ?philosopher2 foaf:givenName "Jean-Baptiste_Say"
18   }
19   }
```

Figure 2.3: Showing the Example of a SPARQL Query

SELECT A *SELECT* is widely used to find out the bindings of the query patterns. This statement is widely used in RDF if in case of no bindings the result set of the *SELECT* statements is empty. *SELECT* statements followed by the $*$ give all the results of the variables in both the RDF graph. The result set is in the boolean either true or false.

ASK This statement is commonly used with *SELECT* queries in a special case where no bindings are returned. By issuing the *ASK* statement the querying engine will determine all the query patterns. The difference with the *SELECT* statement is that this will not require to contain any variable. The benefits of this query are that it will not incur high network traffic as it accesses offline whether the statements are incurred in the triple patterns or not.

CONSTRUCT When building an application, the user might require to retrieve some information from queried data to allow a more effective way to query from other data. *CONSTRUCT* statement composes entire RDF graphs as a result and discovered the bindings, but as compared with the *ASK* statement *CONSTRUCT* request many not contain any variables.

DESCRIBE The SPARQL *DESCRIBE* statement does not return resources matched by the query patterns but describe those resources. Sometimes, the query client wants to know the structure of the RDF, this statement is helpful in discovering the information and bind the result.

Here is the brief example of the SPARQL query illustrated in Figure. 2.3. The query pattern contains *SELECT* statement that limits the projection to the certain variables used in the query such as ?*philosopher*1 and ?*philosopher*2. This query used the *UNION* and *OPTIONAL* as basic operations for modifying the content of the SPARQL query. The first line defines the prefix of the particular resource and the dataset clause is defined by the *SELECT* which determines the execution of the query. The result clause states that what kind of data should be returned by the query, in our example the result of the Dbpedia resources with term ?*philosopher* is returned by the query. The query clause matches the graph patterns and finally the solution modifier that limit the number of results from the query. In order to reduce the retrieved results without restricting the scope of the query, the *LIMIT* keyword is used.

LOD cloud also provides SPARQL endpoint for their datasets. However, querying SPARQL endpoints is troublesome due to network instability, and the connection to these endpoints can be temporarily lost, which affects the query efficiency. These endpoints do not provide any information about dataset modification. Therefore, long-running applications that use a cache must resubmit the queries to keep the local data cache up-to-date. In order to query the RDF SPARQL endpoints are available. LOD cloud also provides a SPARQL endpoint for their datasets. However, querying SPARQL endpoints has problems due to network instability he connection to these end points can be temporarily lost, which affects query efficiency. These endpoints do not provide any information about dataset modification. Therefore, long-running applications that use cache must resubmit the queries for keeping local data cache up-to-date. The decomposition of the SPARQL query is a recursive process and to find a similar query, there is a need to find the distance between the patterns as shown in the Equation 2.2.

$$d\left(PQ_1, PQ_2\right) = d\left(P_{BGP}, P'_{BGP}\right) + d\left(P_{UNION}, P'_{UNION}\right) \tag{2.2}$$

Where PQ_1 and PQ_2 denotes the two queries are structurally the same by calculating the basic patterns between them. Graph Edit Distance (GED) [87] is normally applied to measure the structural similarity between the SPARQL queries. Although, the GED is computationally very expensive, and the structural similarity is not enough. It is possible that two SPARQL queries are structurally the same but differ in the result.

2.1.3 The Ontology Web Language

The Web Ontology Language (OWL)[8] is considered as a standard ontology language for the Semantic Web. However, OWL is more expressive than RDF and utilized the core RDF vocabulary. In the Linked Data context, these terms assist in grasping the full semantics of RDF documents [88, 89]. The OWL identifies the key properties of the RDF and provides a key overview of the entities. The main purpose of OWL is to give support to the content of information instead of just presenting information to humans. The OWL ontology is the RDF graph that contains the set of the triples but it can be written in a different syntax form. The meaning comes from the RDF graph. The information in OWL is gathered into ontologies which are stored as a document.

2.1.4 Linked Open Data: Principles and Best Practices

The LOD approach represents more expressive semantics when referencing resources from the Semantic Web. The World Wide Web combines all technologies to create, link, and consume hypertext documents. These documents are machine-readable records. As described by Tim Berners-Lee [4], the Semantic Web enables the machine in such a way that data can be searched, interpreted, and reused. Linked Open Data (LOD) is another important concept in the Semantic Web, enabling the machine to browse the web, such as DBpedia[9]. LOD refer to the best practices and principles by which the Semantic Web Standards such as RDF and SPARQL can be effectively deployed on the Web. The main purpose of LOD is to provide easy access to structured data. Therefore, LOD provides a method for publishing and connecting the structured data on the web using the standard protocols and these data are continuously evolving [4, 90]. Thus, the following are the four main key features of LOD.

- LOD provides the Uniform Resource Identifier (URI) for referencing the entities.
- LOD allows accessing those URI using the HTTP so that the user may consume these entities.
- LOD utilizes the standards for presenting the information about the entities and these are

[8] https://www.w3.org/TR/owl-ref/
[9] https://wiki.dbpedia.org/

consumed by using their URI.

- LOD interlink with related datasets by using the URI, so it becomes more useful through semantic queries.

Hence, the URI and HTTP is the most fundamental way of identifying and accessing the LOD. The main goal is to make the data machine-readable so that agents can retrieve, parse, and interpret the information. However, the HTML document only rendering the page. But in the case of the LOD, it provides more complex reasoning to infer the relationships between them. There are several projects established in the context of LOD. For example, Freebase [74], DBpedia [75, 75] and Yago [76] which is is collaboratively built from web corpus and represents knowledge in structured form. With the current evolution of the Semantic Web 3.0, the LOD enables data to be linked between sources, as shown in Fig. 2.4. The LOD cloud contains almost 570 datasets from different domains that are interlinked with each other.

Massive amounts of data are being produced and currently available in life sciences. However, these data are unstructured and difficult to integrate. Therefore, the Linked Data paradigm is currently suitable for publishing and connecting life science datasets to improve access and use. The Linked Data cloud contains almost 570 datasets [91] from different domains that are interlinked with each other as shown in Figure 2.4. A considerable portion of Linked Data is comprised of life sciences data, significant contributors include the *MEDLINE* [10], *Bio2RDF* [92] and *DrugBank* [11]. *MEDLINE* is an American national medicine bibliographic database that contains more than twenty-four million references to journal related to bio-medicine. *Bio2RDF* is built over the Semantic Web to provide biological databases interlinked with life science data. *DrugBank* contains data related to *Bioinformatics* and *cheminformatics* with comprehensive target information. These databases contain the protein sequences that are linked to the drug entries that target the protein data [46, 78].

[10] https://www.ncbi.nlm.nih.gov/pubmed
[11] https://www.drugbank.ca/

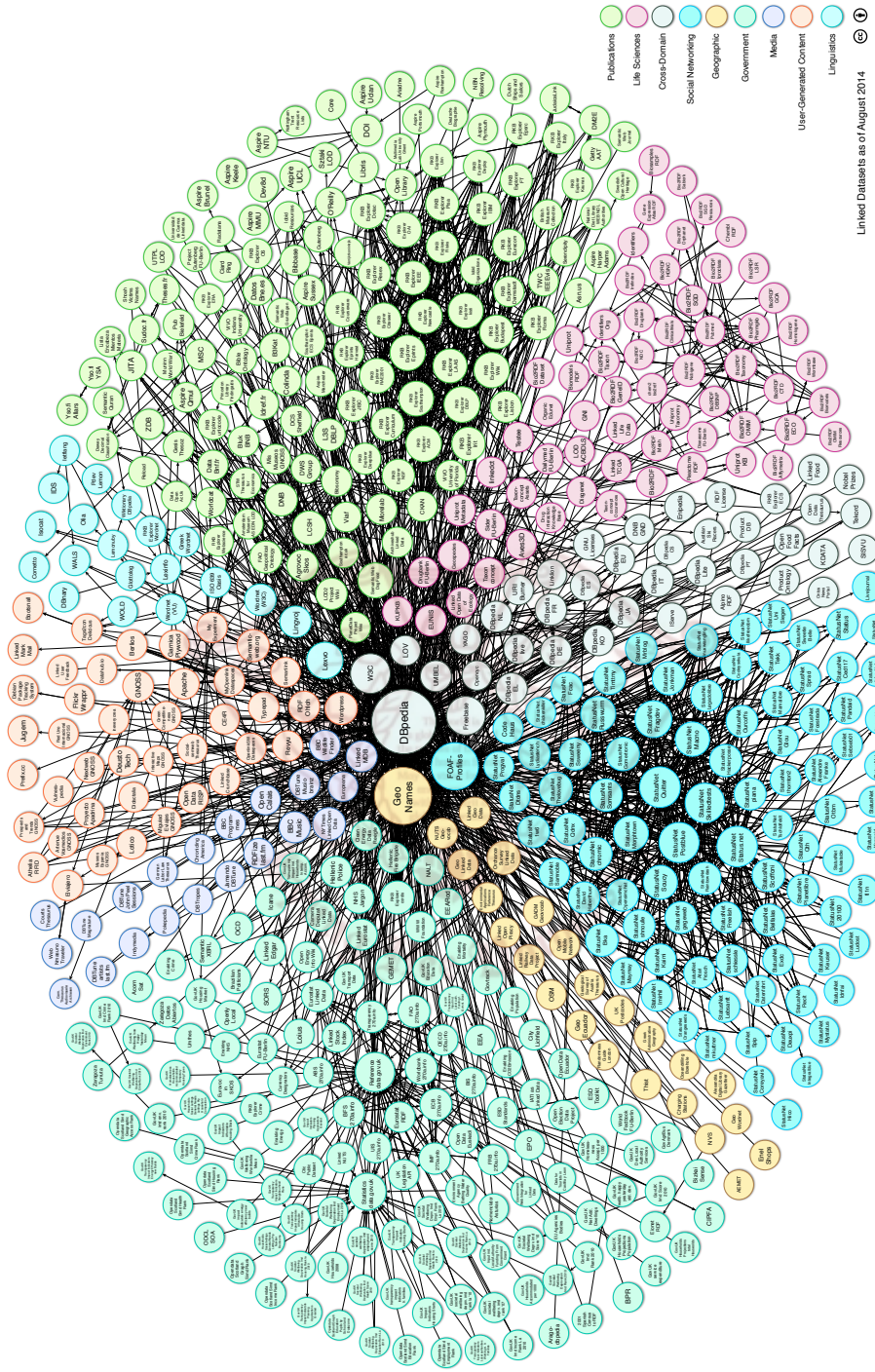


Figure 2.4: Showing the LOD diagram containing interlinked data from multiple domains.

2.2 Evolution of Linked Open Data

LOD is highly dynamic, contents get added or removed. As an application using the dynamic LOD, the highly dynamic nature incurs problems such as broken links or outdated data [93, 94]. These issues are related to the dataset dynamics [95], which investigates the approaches that detect changes in LOD and explore the new path towards exploration and design of query related to the dynamic data. However, the LOD community currently lacks a high-quality platform to maintain the snapshots of the data over a period of time. There is only a domain-specific platform to store the data of weekly monitoring of LOD datasets, among the most popular is DyLDO (Dynamic Linked Data Observatory) [1], which focus on monitoring the evolution of the LOD sources. The example of the snapshot of DYLDO is shown in the Table2.1, where X_t represents as a set of triples captured at a point in t and the set of all the snapshots captured at different points in time t is denoted as $X = \{X_{t_1}, X_{t_2}, X_{t_3}, ..., X_{t_N}, \}$ where N represents the total number of snapshots.

To analyze the LOD dynamics, Umbrich et al. [95] proposed a Multicrawler framework that extracts the LOD documents from the sources, there framework extracted the LOD documents on weekly basis resulting in 650,000 RDF documents. Similarly, Kafer et al. [1] utilize the breadth-first crawling utilizing the 220 URIs to extract the document from the DataHub[12] repository. After extracting those documents Kafer et al. then monitored these documents to see the dynamics of the LOD. However, after the analysis of the Umbrich et al. and Kafer et al., they noticed that almost 65% of documents did not change and half of the documents had a change frequency not more than 3 months. Similarly, Dividino et al. [96] monitored the dynamics of the LOD over 19 months and concluded that half of the LOD sources are highly dynamic. Based on these findings, Nishioka et al. [23], Dividino et al. [24], and Kafer et al. [1] proposed a scheduling strategy to keep the LOD data up-to-date based on the change rate.

2.2.1 Accuracy of the Index Model

Nowadays, RDF data is ubiquitous and thanks to the linked data movement where billions of data available online. The basic principles behind the linked data is to model, interlink and publish at ease. The evolution of these data are scale-free and requirements are not handled by the centralized

[12] https://datahub.io/

Table 2.1: Showing the example of the snapshots of Linked Data Cloud

X_{t_1}: a snapshot at time t_1		
Subject	Predicate	Object
db:Pierre Peloso	db:location	db:Green Village
db:Pierre Peloso	db:works	db:Green University
db:Green Village	db:population	264324
X_{t_2}: a snapshot at time t_2		
db:Pierre Peloso	db:location	db:Green Village
db:Pierre Peloso	db:works	db:Green University
db:Pierre Peloso	foaf:knows	db:Mark Thomas
db:Green Village	db:population	224123
X_{t_3}: a snapshot at time t_3		
db:Pierre Peloso	db:location	db:Green Village
db:Pierre Peloso	db:works	db:Green University
db:Green Village	db:population	264324

environment. Although, most of the data are published in distributed way and consumers are using RDF and HTTP to integrate into the existing applications. However, growing of these sources post significance challenges such as the accuracy of the index model over evolving data. The main reason of the inaccurate view of the indices are mainly due to the decentralized nature of the linked data publishing. However, indices are used to fast lookup of the data. Since, the growth of the linked data post significance challenges on the index structures as outdated indices are often provide the wrong information. However, more bandwidth are utilized when maintaining the indices updates. Thus, the major problem is archiving polities is the scalability problem when managing the evolving RDF data [97–99].

Large scale data applications are widely used on the cloud environment. The major challenge when managing the Big Data is to provide a fast and reliable query executions. Similarly, key-value store needs to effectively determine the existence of the elements without having exhaustively search. The indexing technique that are proven to be work on the traditional system are not able to work on the big data systems. The approaches that keeps the full index in a memory will face a hug difficulty when dealing with the large datasets. Due to the lots of the addition and deletion it is not feasible to maintain logical structure in memory as the amount of the data grows whole index need to be sorted causing the disk access problem.

In general, we argue with the current indexing scheme to deal with the large evolving data. The existing solution strategies need a strong scalable approach to ever deal with the increasing amount of the data and strong space efficiency in term of maintain the indices. The query search performance is another desirable factor that need to be considered during the scale-up process. Similar, query accuracy to provide to provide reliable results are missing in the existing solutions. Hence, there is a need of the strong element location and deletion capability to allow recycle of the unused space is needed for the big data.

In the field of big data indexing. The main approaches of the big data are categorized in the clustered and non-clustered indexing. In the big data computing, Apache Hadoop [100] is well-known big data processing platform that utilized the MapReduce programming model and HDFS for storage. However, it lacks to provide the efficient response and researchers have proposed an indexing frameworks in an non-invasive way i.e., they do not require to change the core part of underlying framework. However, Mapreduce programming model do not performs well on the traditional data systems. HadoopDB [101], which is the effort to integrate the traditional relational database system on the top of the Hadoop. Due to the high complexity in HadoopDB, the idea of such iteration is not well acknowledged. Hadoop++ [102] provides the more efficient solution to introduced the Trojan indexing which introduce the concepts of the data sorting similar of database cracking [103]. Although Hadoop++ increases the search performance of the Hadoop framework but it also suffers from the two main problems. Although, the index creation time of the Hadoop++ is very expensive as compared to the Hadoop built-in full scan. To deal with the index creation process HAIL [104], utilized the data replication features to increase the number of the indexes. Although the number of the indexes depend on the replication factor, this scheme also suffers for the large datasets due to no independent solution from data replication.

To overcome the problems exists in the clustered based approaches, non-clustered based approaches have been proposed. These approaches are categorized as tree-based, hashing and bitmap indexes. Unlike clustered based non-clustered approaches are effective in term of the query execution and data retrieval [105]. This approach is fast in term of creating the indexes and computational cost is less i.e., B-Tree is more feasible on growing datasets [106]. Similarly, Apache Lucene implements inverted indexes to retrieve the data [107]. However, Apache Lucene requires signif-

icant size in the memory for high volume of big data. Similarly, wise selection of the indexing model reduces the overhead.

Most of the index support the specific task for the application. Technical, linked data consist of the N-Quads [13], which consist of the (s, p, o, c), which corresponds to the subject, predicate, object and context. Let G_t represent the LOD snapshot that consists of all the NQuads taken in time t. The set of all the snapshot is denoted as $\mathbb{G} = \{G_{t_1}, G_{t_2}, G_{t_3}, ..., G_{t_N}, \}$, where N represents the number of the snapshots. Hence, a quad (s, p, o, c) ε G_t consist of the RDF triples along with the context, where the quad was retrieved. In most of the cases index model do not store all the information but rather a specific information in the NQuads. To define the abstract model we have a LOD over a data set R of (s, p, o, c) and D is a derived set which constitute the restriction of the quads to the smaller set of the tuples. Most of the index model define a key elements κ which is used for lookup the elements from the data. These key elements are used for the selection function $\sigma : \kappa \rightarrow \rho(D)$ to select the data items from the index. Hence, the abstract index data model is defined as a tuple which consist of the (D, κ, σ), where D is a data items and σ is the key elements that is used to lookup the items and finally the selection function σ that is used to retrieve the data items from the index [108–110].

2.2.2 Overview of the Change Detection approaches

The LOD are constantly evolving and the applications using these data face severe The LOD is constantly evolving and the applications using these data face severe challenges such as addition and deletion of the data and broken links problem [93]. To overcome these issues, detection of the changes in LOD is a main crucial step [111]. As the data evolve continuously the detection of the changes should also be performed at regular intervals [112, 113]. The use of a different kind of change information supports different types of scenarios. In the case of structural interlink, maintenance requires information of LOD to maintain the structurally broken links that are not accessible. Similarly, in the case of the semantic interlink maintenance require information on the resources that are changed during the evolution. Finally, the synchronization of the local copies requires information on the addition and deletion of the triples. Research in [111, 114] have

[13] https://www.w3.org/TR/n-quads/

identified that the majority of the LOD cloud is static in nature as no changes occur frequently, while some of the resources are more frequently changed than others. The notion of region [115] based change behavior is also an active research area that utilizes the regions where the changes are more frequent.

2.2.3 Using Provenance Information to Detect Changes in LOD

The LOD cloud changes frequently and applications require constant prefetching to keep the local data caches up-to-date. The main technique of detecting the changes in LOD sources is the use of *HTTP* header information such as *Last-Modified* field that shows when the resources have been changed in the past. The use of provenance information supports the decision process of determining which sources from the LOD need to be updated. Provenance information captures such as changes in the author, its timestamp, and software agent requesting for some changes [116]. The most intuitive way of detecting the changes in the LOD is to exploit the HTTP header information. According to the LOD principles, the resources are modeled using the HTTP and the SPARQL endpoints are available for particular URI to respond to the request from the agent. Usually, this information makes the use of the HTTP protocol that contains the provenance information about particular resources such as owner, creation time, and date. However, the information that contains the HTTP headers may be inaccurate or wrong [117]. Therefore application relies on this information are likely to draw the wrong conclusions. As LOD evolves over time, therefore graph change as information is added and removed to ensure the quality or freshness of these sources needs to be periodically updated. The naive solution for detecting the changes in the resources of LOD is to download the LOD resources and compare them [2, 3].

The LOD cloud is composed of various data servers that enable data access via the HTTP protocol. When the user request is sent to the server an HTTP GET message is received [35, 118]. The main body of the message consists of headers that contain the code along with the message body. The individual message contains the numeric codes and the header line. Kjernsmo et al., [48] investigate the HTTP header information that examines the availability of the resources in the LOD cloud. The header fields gave the information to the server to process further access to the resource requested by the URI. Following are the message body of the response:

- *Content-Language* This language describes the size of the entity-body in a decimal number.
- *Content-Type* This indicates the media type of the resource sent.
- *Content-MD5* mainly for the end-to-end message integrity check.
- *Data* It represents the date and time at which the message is originated from the sever.
- *Expires* it shows the detail of the stale response.
- *Last-Modified* Shows the date and time at which the resources from the LOD was last modified.

There is various work published in the existing literature that shows the characteristics of the LOD cloud. The main purpose of this research is to perform the structural analysis of the LOD cloud in order to obtain the characteristics of the data [45, 119]. In addition, there is a related work that shows the best guideline and model to publish the LOD [95]. However, this work does not consider the dynamics of the LOD cloud. Ding et al., [120] perform the experiments to crawl the 300 million triples from LOD and perform the analysis to extract the last-modified time from the LOD. The result of their experiments also shows that the LOD keeps on evolving, which indicates that the content is added and removed periodically. Monitoring tools [121] also provide information about the availability of LOD. DSNotify [93] and Semantic Pingback [122] are often considered generic frameworks. DSNotify uses the time blocking technique to detect and fix broken links between resources. Semantic Pingback uses a notification to establish new links. However, both of these frameworks are mainly focused on static resources. The main problem of these frameworks is that their main focus mainly on static resources. Dividino et al. [24, 35] investigated the availability of the *last-modified* field in the HTTP header and revealed that only 7% of the documents provide the correct information about the last change.

2.2.4 Capture the Dynamics of LOD

Most of the existing work provides the characteristics of the LOD but does not consider the dynamics of the cloud, how it evolved over the period of time. Recent investigations have shown that data published in the LOD cloud is updated frequently. Therefore, local copies become outdated.

The dynamic function helps to capture the evolution of the LOD sources in order to check for the changes. The LOD applications need to permanently visit the sources in a brief interval. State-of-the-art metrics [123] mainly quantifies the changes between the two datasets, but they do not consider the evolution of the LOD sources. Data on the LOD cloud changes frequently and applications replying to that data need to constantly update the cache. Instead of visiting all the sources a good scheduling strategy only visits the changed sources and updates the local caches. So far, only a limited work addressing the scheduling of the LOD source up-to-date. The most well know scheduling strategy to maintain the indices of the LOD documents such as the PageRank algorithm and metrics for quantifying the changes of LOD sources [124].

In order to analyze the dynamics of the LOD, Umbrich, et al., [9] mentioned that changed had a frequency of more than 3 months. Dynamic Linked Data Observatory (DYLDO) [1] is a monitoring framework for a Linked Data source, which monitors the changes based on the collection of the regular crawl on LOD sources. The authors give insights about the availability of documents. In the DYLDO study, they utilized the HTTP header information to detect the change. The study of the evolution of the Web and its dynamics are first studied by Bray et al., [125]. Therefore, based on the large collection of the documents authors discovered that the changed behavior of the Web is closely related to a Poisson distribution [126–128].

In summary, a variety of the existing work that is focused on change detection [129] only focuses on the structural analysis of the Linked Data cloud in order to obtain the characteristics of the data [18,45] most prominently on the query caching [11,48,95], but the working implementations are rare.

2.3 Preservation of Linked Open Data

On the Semantic Web, a large amount of work on managing and preservation of LOD has been quite diverse. Many surveys have been published which highlights the foundations, challenges, and opportunities of managing LOD cloud [115,130]. A number of works have dealt with issues related to query LOD. Linked Open Data (LOD) changes over time and it creates a need to maintain the history of the datasets [131]. Hence, preservation of the archive has been an active area related to LOD. Similarly, in the area of data analytic, there is an ongoing demand to maintain the history

of the datasets. Therefore, such archives are important to look up the data at a certain point. With the continuous evolution of the LOD, archiving has been an issue for RDF [132]. A recent survey on archiving the LOD [131] suggested the improvement of versioning capabilities of the existing approaches. Currently, there is no solution exist to query between different versions of the RDF datasets.

2.3.1 Existing RDF Archiving system

RDF storage systems utilize the indexing approaches to reduce the query run time. The existing archiving systems are typically based on the relational or document storage such as RDF-3x [133] is based on the clustered B+ Tree to maintain the indexes. Hexastore [134] is another RDF storage system that is based on the idea of indexing the RDF data in a multiple index framework and utilize the dictionary encoding to compress the common triple components. Triplebit [135], a fast and compact system for large-scale storage of RDF data based on the two-dimensional storage matrix including the design of the bit matrix storage structure and compression for storing the huge RDF graphs. K^2-Triples [136] is another RDF storage technique that utilizes indexing of the K^2-tree structure as it allows SPARQL queries on the memory without decompression. The result of this storage outperforms the traditional query processing system. Another RDF storage system RDFCSA [137] that utilize the compact self-index that store the data with its index. This system results in less storage space as compared to the raw storage but allows the pattern based search on it. HDT [138] is another binary RDF representation system of real-world RDF data that is based on the set of metrics called Header Dictionary Triples (HDT) that organized all the identifiers in the RDF graph and provide a dictionary to provide the catalog of the RDF terms.

The preservation of the Linked Open Data (LOD) emerges as a novel research area to assure the quality of datasets over time. However, many issues arise when archiving the evolving data. The main efforts addressing the challenging of RDF archiving are categorized into three main storage strategies: *Independent Copies (IC)*, *Change-based (CB)* and *time stamp-based (TB)* approaches [131]. The *IC* approach manage each version as a different and isolated dataset. But this approach often faces scalability issues as information may b duplicated on each version of the RDF datasets. Therefore, querying over the *IC* can be cumbersome due to the processing efforts.

Table 2.2: Overview of the existing RDF arching system

Name	Features	Limitation	Archiving Strategy
SemVersion [2]	Introduces the RDF-centric versioning approach for structure and semantic versioning	Not a scalable and not space efficient	Independent copies
Cassidy et al. [139]	Proposed a version control for RDF stores	The proposed solution do not ensure the overhead in terms of additional space and processing time	Independent copies
R&WBase [140]	Proposed a versioning layer for existing triple store	Overhead cause by accumulating the deltas for each version	Changed-based approach
R43ples. [141]	Proposed an approach using named graphs to semantically store the difference between versions	Only based on the semantics	Changed-based approach
RDF-3x [133]	A generic system that include the support of the RDF version	Only triples of given time are return, exhaustive and not a space efficient approach	time-based approach
RDF-TX [142]	Proposed an in-memory query engine that is based on the B+tree.	only work in-memory based query engine	time-based approach
v-RDFCSA [143]	Provide a version based queries based on the compressed RDF archives	Incurs high computational cost	time-based approach
Hexastore [134]	Indexing framework for managing RDF data	Worst-case increase in the index space	time-based approach
Triplebit [135]	Fast and compact system for accessing RDF data	Not efficient storage space	time-based approach

Currently, SemVersion [2] utilize the *IC* approach to manage different versions of ontologies, however, the implementation details of their system are still unknown. To address the scalability issues, *CB* approach only store the changesets between different versions. The query mediator of *CB* requires the additional computational cost to process the changed sets. Cassidy et al. [139] proposed a version control for RDF triple stores. The implementation is based on the Redland python that provides the wrapper called *record* to store addition and deletion from the triple store. Im et al. [144] proposed a framework for RDF version management in relational databases, as it stores the original and delta version. However, this approach is not effective in terms of storage overheads and slow query performance. R&WBase [140] is another *CB* that build on the principles of the distributed version control. Here the triples are stored in *Quad-store* and reducing the number of stored triples in a single graph. Graube et al. [141] introduce a novel way of dealing with version control for LOD, it represents *R43ples* as an approach using named graphs to store different version of changesets as separate named graphs. In time stamp-based *(TB)* approaches, Hauptman et al. [145] introduce a delta based storage approach based on the temporal validly of the facts. The implementation of this approach is based on the Sesame [146] and Blazegraph [147]. However, this approach response is slower than the *CB* based approaches, but storage is better than CB. X-RDF-3X [148] is an extension of RDF-3X [133] that enable versioning support using the *TB* approach. Their system supports the storage of creation and deletion of the triples, where SPARQL queries retrieve the triples at the given time. RDF-TX [142] is an in-memory query engine that is based on the B+Trees that outperforms other systems and store the changesets using the timestamp-based approach. v-RDFCSA [143] provide version based queries on top of compressed RDF archives. The results suggest that v-RDFCSA reduces a space up to 35 times over the state-of-the-art systems. A similar storage platform Dydra [149] is an RDF graph storage service that stores and retrieves the content from the versioned RDF.

2.4 The use of Caching in Semantic Web Applications

The performance of the triplestores is one of the major obstacles for deploying the large-scale development of semantic technologies [58,150]. In the recent past, many efforts have been made to improve the performance of effectively querying LOD [59,60], however, as compared to querying

a triple stored in a relational database, querying a triplestore is still usually slower by 20% [11]. In recent years, many efforts have been made to circumvent the problem of effectively querying LOD [61,62], among these, caching is the most popular technique to reduce query time by serving the requests from a cache. Therefore, the success of the Semantic Web application for storing and retrieving the RDF is gaining importance. But traditional RDF systems lack optimization for query processing and also lack the indexing structure for speeding the access of triples [151,152]. The idea of query caching is to reuse the previously issued queries. For example, a possible query is to retrieve all papers about the 'Semantic Web' written by the researchers of the Kyung Hee University, South Korea. So, the processing of these queries can be done more effectively if we already know the results of the paper about the 'Semantic Web', we can save the computational cost of computing the join. In the distributed setting the benefits of caching is more evident, as the previously issued queries are stored on the remote source instead of running all the queries the result is immediately sent from the cache. The caching will immediately improve the robustness of the system where the remote resources are not available due to the network insatiability. Most of the caching approaches utilize *Syntactic query caching* to reduce the network cost and utilize when to avoid the complete information of the result as the information is already available in the cache [151]. In order to reduce the system check whether the object is already available in the cache. Another approach is *Semantic query caching* when processing the new query, the system determines whether the useful results present in the cache, in case the system did not find any results the query is evaluated in the common way [153].

2.4.1 Query suggestion

Recently, query suggestion has been introduced into SPARQL processing. It plays a Recently, query suggestion has been introduced into SPARQL processing. It plays a vital role in improving the overall processing of the query. The suggestion is made based on the mining of similar queries from logs. Graph Edit Distance (GED) [87] is normally applied to measure the structural similarity between SPARQL queries. However, GED is very computationally expensive, and the use of structure similarity is insufficient. It is possible that two SPARQL queries are the same but differ in their result. To overcome this drawback, Shu et al. [53] proposed a content-aware ap-

Table 2.3: A comparison of related work.

Work	Advantages	Disadvantages	Method	Query similarity	Cache Replacement	Pre-fetching
Martin et. al [11]	Cache complete triples query results; introduced the proxy layer between the application and SPARQL endpoint to cache repeated query results	Only considers repeated queries	Structure-based similarity	×	✓	×
Chun et al. [154]	Proposed maintenance policy that update the cache prior to query execution	Only update the local cache at the system idle time	Content-based similarity	×	✓	×
Nishioka et. al [23]	Proposed a prediction model whether change will occur in the future	Do not consider the query similarity while predicting the future queries	Result-based similarity	×	✓	×
Godfrey et al. [52]	Define a general framework in logic for semantic query caching	The proposed algorithm has exponential time complexity	Content-based similarity	×	×	×
Yang et al. [54]	Adaptive cache to store intermediate results of a SPARQL query	No cache policy was introduced in their work	Result-based similarity	✓	✓	×
Dar et al. [51]	Proposed semantic region-based caching and a distance measure to update cache	Only considers a semantic region rather than tuples	Distance-based similarity	×	✓	×

Table 2.3: continued

Work	Advantages	Disadvantages	Method	Query similarity	Cache Replacement	Pre-fetching
Shu et al. [53]	Introduced *query containment* which evaluated whether a query can be answered from the cache or not	Containment checking is a computationally expensive task	Content-based similarity	✓	×	✓
Papailiou et al. [40]	Work-load adaptive caching to reduce the SPARQL query response time; introduced canonical labelling for optimal join execution plan	No policy for cache replacement was introduced in their work	Result-based similarity	✓	×	×
Lehman et al. [5]	Proposed a machine learning approach to leverage the query processing over KBs; no knowledge of underlying schema is required	The Feature modeling method introduced in their paper is time-consuming	Structure-based similarity	×	×	✓
Fernández et al. [155]	Proposed an archiving system to efficiently retrieve data from evolving RDF	Unable to replace the cache during system idle time	Structure-based similarity	×	✓	×
Proposed	Proposed an adaptive cache replacement that accelerates the overall processing of querying over earlier works	Prefetching the previously issued queries degrades the performance of the system	Structure and content-based similarity	✓	✓	✓

proach that utilized *query containment* to estimate whether the queries can be answered from the caches. However, this approach is not widely utilized by the semantic web community since the containment checking approach produces very significant overhead. Lorey et al. [156] proposed a query augmentation approach to alter SPARQL queries to detect frequently recurring patterns. The benefit of their approach is to answer the query from the cache without accessing the LOD. However, the major limitation is that it considers only the queries requested by the same agent and the hit rate of the template-based approach is only 39% [157]. In contrast to the aforementioned query suggestion methods, our approach considers both content-wise and structural similarities based on a simple distance score, which results in higher hit rates, shorter query time, and less spatial overhead.

2.4.2 Semantic caching

Semantic caching was originally proposed for the Database Management System (DBMS) [11, 51] and the purpose of the DBMS is to reduce the overhead of retrieving data from the cloud. Godfrey et al. [52] proposed the notion of semantic overlaps and introduced a caching approach that utilized client-server systems. To extend this idea, Dar et al. [51] proposed a semantic region-based caching technique and introduced a distance metric to update the cache such that the cold (e.g., less frequently access) regions are removed from the cache. Martin et al. [11] propose to selectively invalidate cache objects on updates of the knowledge base by identifying the affected query results. However, their work does not consider query similarity for cache replacement. Yang et al. [54] proposed server-side caching to decompose the query into the basic graph patterns and cache their intermediate results. To prefetch similar-structured queries, Lehman et al. [5] proposed a supervised machine learning approach that performed analysis on the user's previously issued queries. Their approach filters the range of possible answers and utilizes a learning technique to ensure that no prior knowledge of the underlying schema of LOD is required. Nishioka et. al [23] proposed a periodic crawling strategy that predicts whether the change occurs in RDF triples. However, Lehman et al. [5] and Nishioka et. al [23] did not consider the system overhead as their performance measure. Recently, a proactive policy for maintaining local cache is proposed by Chun et al. [154] that alleviates the expensive job of copying the LOD at idle time. In summary,

only a few works have been reported to deal with the problems related to the semantic caching for SPARQL queries. We propose a client-side adaptive strategy to utilize caching for SPARQL query processing. The goal of our research is to keep track of the access queries and evicts the less valuable content from the cache in an overhead-efficient way and regardless of system idle time.

2.5 Summary

In this chapter, first, we studied semantic web foundations in which information is given in a well-defined format, that enables the computer and people to work in cooperation. In literature, various works have investigated the characteristics of LOD and estimation of the change. Since the LOD cloud is a global information space and it structurally connects data items. The distributed web-based nature of data motivates many applications to keep local copies of the data. Due to the dynamic nature of the linked data many applications need to keep updating the local copy of the data. The main problem is when to perform the updates. Most of the existing work discusses the problem of scheduling refresh queries for a large number of registered SPARQL queries. They have investigated various scheduling strategies and compared them experimentally. The main contribution of their work is an empirical evaluation of the real-world SPARQL queries. We identified the limitations that the healthcare organizations are publishing data publicly, but consuming data is difficult due to the rapid growth of the linked data cloud. Most of the applications that are consuming linked data suffer from challenges such as change estimation and accuracy of the index for keeping the data fresh for visualization.

Chapter 3

Change Detection for Evolution Analysis

3.1 Overview

In this section, we present our proposed approach to estimate the change in the linked data together with the general update policy when a data source should be fetched. First, we discuss the change metric for the linked data sources. To prioritize the recent changes that occur in the Linked Data, we have proposed an algorithm assigns weight to the recent changes that occurred in the linked data.

The overall logic of our proposed approach is shown in Fig. 3.1. The proposed approach consists of a change metric that identifies the addition and deletion of triples and thus quantifies the evolution of the LOD sources. In most of the cases, the application requires only the latest changes AACP algorithm utilizing a weight function to give importance to the recent changes. Based on the previous estimation, our proposed update policy called PASU keeps the local data cache up-to-date. PASU also assigns a preference score to the highly dynamic sources. Our proposed method helps the application to identify the changes and update the local copies of the LOD. The following are the steps involved in our proposed approach:

- First, we have applied the change metric Δ on previously captured triples at point t. Δ maps the addition and deletion of the triples with the real number.
- Second, the LOD sources continue to evolve and the rate of change is not explicitly known. Thus, it is insufficient to apply only Δ. Therefore, we have utilized Dynamic Characterizing Sets (DCS) that identify the changes in the triple-level in each snapshot. An example of DCS is shown in Table 3.1.
- So far, we are able to quantify the changes using the proposed change metric. To prioritize

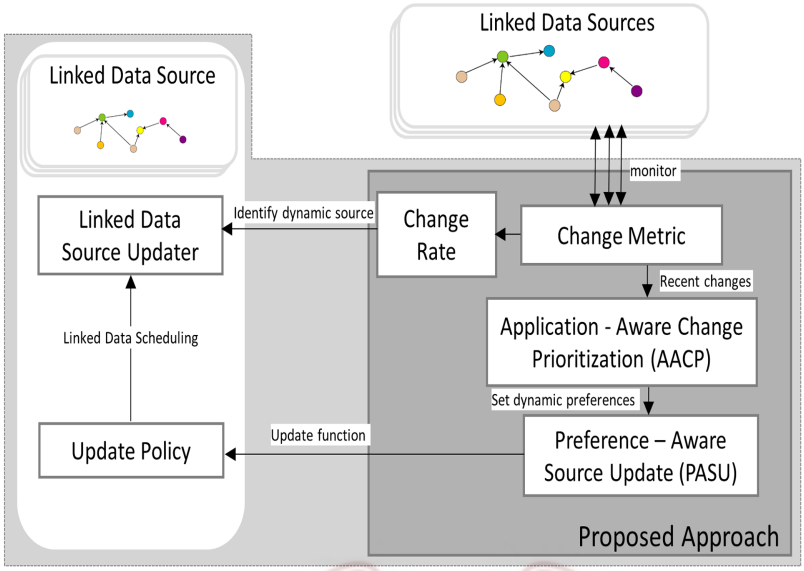


Figure 3.1: Overall Architecture of the Proposed Method.

the recent changes higher than the older changes, we have applied a weight function to the overall flow of the algorithm, as shown in Algorithm 1.

- Finally, we introduce the novel scheduling policy called PASU, which determines the order in which the Linked Data sources should be fetched. The overall flow of the algorithm is shown in Algorithm 2.

3.2 Change Metric

Due to the continuous evolution of Linked Data, local views have become outdated. We proposed a change metric that quantifies the evolution of Linked Data. The main benefit of using the change metric is to alleviate the expensive job of copying the whole data instead of only updating local views with the changed items. We utilized the Dynamic characteristic Set (DCS), which identifies the addition and deletion of the items in the Linked Data cloud. Typically, queries issued by the end-user are repetitive and follow similar patterns that only differ in a specific element. Therefore, we use a bottom-up matching approach to find similar queries. For the structurally similar queries, we first compute the distance between the patterns, then the results of these queries are placed in

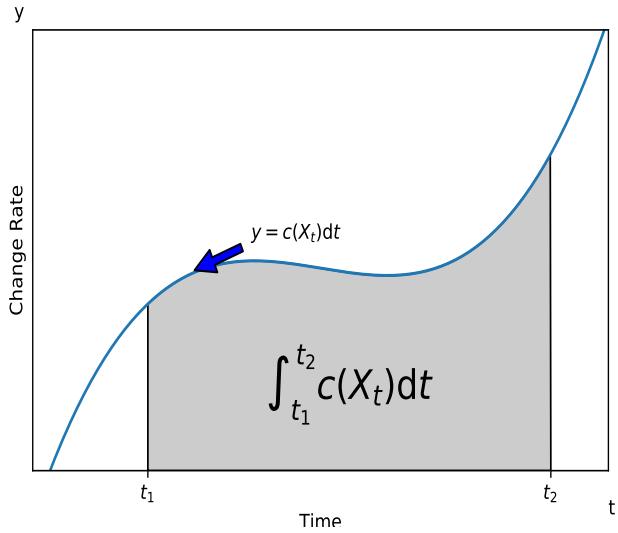


Figure 3.2: Integrate the Change Rate Function to Quantify the Evolution of Data

a cache for future access. To improve the efficiency of our system, we schedule the view maintenance as a low priority job that combines the multiple tasks into one larger job and executes it when the system is in an idle state.

We propose a change metric to quantify the evolution of the Linked Data Cloud. Change metric utilize the Dynamic Characteristics Sets (DCS) [39], a scheme abstraction that classifies the Linked Data on the basis of the properties of *subjects* and *objects*. Moreover, DCS captures the inherent structure and relationships of the Linked Data cloud. The DCS is a combination of properties and types that are used to describe the content in Linked Data. A change at any level of Linked Data implies a change in the mapping of the *properties* and *type*.

The change metric quantifies the evolution of Linked Data. Jaccard distance is widely used to measure the difference between the dataset, such as the addition and deletion of the items. The difference is represented as $\Delta(X_{t_2}, X_{t_1}) \geq 0$. Our assumption is that the change rate of all the snapshot X captured at time t is given by the function $c(X_t)$. As shown in Eq. 3.1, we can integrate the change rate function to quantify the evolution of the LOD sources as shown in Fig. 3.2.

$$\Delta(X_{t_2}, X_{t_1}) = \int_{t_1}^{t_2} c(X_t)dt \tag{3.1}$$

The rate of change $c(X_t)$ is not explicitly known and our formulation for the change rate is

Table 3.1: Rate of change in snapshots using DCS

Snapshot X_{t_1}	Snapshot X_{t_2}	Status
DCS_1 = {foaf:Person, foaf: knows}	DCS_1 = {foaf:Person, foaf: knows}	Unchanged
DCS_2 = {foaf:name, foaf: knows, dbpedia:project/media}		Deleted
	DCS_{2a} = {foaf:name, foaf: knows, dbpedia:project/social}	New
	DCS_{2b} = {foaf:name, foaf: knows, dbpedia:internProject}	New

based on the idea of a Characteristics Set (CS) [158]. To estimate the change, we have proposed the Dynamic Characteristics Set (DCS). The DCS is a combination of properties and types that are used to describe the content in LOD. A change in any level of Linked Data implies a change in the mapping of the properties and type.

3.2.1 Dynamic Characteristic Set (DCS)

Let X_t represent the LOD snapshot captured at time t, which consist of subject s, predicate p, and object o, where P are the set of the properties in X_t, and T is the set of the type. The DCS is an element of the powerset over P and T, and it can be represented as $DCS \in \mathscr{P}(P \cup T)$. Any change in the DCS shows that the new content in the LOD has been added or removed. The addition of a new item in the DCS means that the properties that are used in the LOD have not been observed before. Similarly, the deletion of the properties from the DCS shows that the term is no longer used. An example of the DCS is shown in Table 3.1.

Example 3.1. To explain the changes in the LOD cloud, consider the snapshots of LOD captures at time t as shown in Table 3.1. The changes are illustrated by DCS_1, DCS_2 , DCS_{2a} and DCS_{2b}. The changes in DCS_1 = {foaf:Person, foaf: knows} captured at X_{t_1} and DCS_1 =

{foaf:Person, foaf: knows} captured at X_{t_2} remain unchanged across snapshots. From the later version of the snapshot DCS_2 was deleted, and new combinations of DCS_{2a} and DCS_{2b} were observed. Thus, DCS_2 is no longer used in the LOD.

3.2.2 Prioritize Recent Changes

So far, we are able to quantify the changes in the LOD, and it is important to prioritize the changes that tend to be less important as time passes; therefore, a change metric should strengthen the recent changes and weaken the older changes. Intuitively, we consider the index update scenario in which the recent changes are more important than older changes. To achieve this goal, we have extended our proposed change metric and incorporate a weight function to assign importance to recent changes. We have modified the change metric and proposed the AACP algorithm to assign importance to recent changes. Suppose X_t is a set of triples, $c(X_t)$ is a change rate of the LOD dataset and $w(t)$ is a function that assign weight as mentioned in Eq. 3.2

$$\Delta(X_{t_2}, X_{t_1}) = \int_{t_1}^{t_2} w(t).c(X_t)dt \tag{3.2}$$

In the proposed Algorithm 1, we take the *last modified date* as input and assign the weight based on the recent changes. The weight function of the item can be written as a function of its age [159], where the particular age of the LOD is the time when the last time the LOD was updated. To identify the correct present time of an item in the LOD the proposed weighted function satisfies the following properties:

- $w(X_t, t_i) = 1$ when $t_i = t$ and $0 \leqslant w(X_t, t_i)$ for all $t \geqslant t_i$.
- The weight of the item is monotone non-increasing as time increases.

As shown in Algorithm 1, our first step is to compute the age, determine which items were updated, and assign weight based on the recently changed item. So far, we have constructed the change metric and also able to identify the recent changes using the AACP algorithm. In the next section, we propose an updated policy that utilizes a change metric when conducting updates, keeping the Linked Data cache up-to-date.

Algorithm 1: Application-Aware Change Prioritization (AACP)

Weight Function $w(X_t, t)$ ▷ Assign weight to recent changes
Input : A timestamp t of Linked Data item X_t
Output: Assign weight to the recent change items
for $t_i \leftarrow X_i$ **do**
    // Compute age of the LOD items
    if *t is the timestamp of the record* **then**
        $t = t_p$; ▷ If t_p is a present time of the item update
        $Age = t_p - t_{last}$; ▷ Compute the age of the item
    else
    end
end
if $t_i \neq X_i$ **then**
    // Set condition on weight
    $f(X_t, t_i) = 1$; when t_p ▷ present time satisfied
    $0 \leqslant f(X_i, t_i)$; $\forall\, t \leqslant t_p$
else
end
// return the most recent changes
return $w(X_i, t_i)$;

3.3 Scheduling Update of Linked Data

An update strategy aims for driving the data sources based on the preference. In an An update strategy aims for driving the data sources based on the preference. In an ideal case, [23], an update strategy only visits the source that has been changed.

3.3.1 Preference-Aware Source Update (PASU) Algorithm

We have proposed our update policy called Preference-Aware Source Update (PASU), which visits the LOD sources based on the preference score. Most of the LOD applications pre-fetch data and store it in its cache. However, a preference score is required to determine important sources. In our case, the preference scores could be sorted in ascending and descending order, and the update function uses this information for cache replacement. The size of the cache is limited; therefore, cache replacement is a problem of identifying a recently changed item and replacing the local data cache. Based on the high dynamics, sources will be updated at the highest priority. The preference

assigns the score based on the history of the sources. We have utilized the change metric as an aggregation of absolute changes. The preference score is computed as shown below:

$$f_{PASU}(X) = \sum_{i=0} \frac{\Delta(\mathrm{X_{t_{lastupdate-1}}}, \mathrm{X_{t_{lastupdate}}})}{\mathrm{t_{lastupdate(X)}} - t_{lastupdate(X)-1}} \tag{3.3}$$

In Eq 3.7, $t_{lastupdate}$ is a function that returns the time when LOD was last updated. Similarly, the function is recursive, as $t_{lastupdate-1}$ returns the time prior to the last update and Δ quantifies amount of the changes between the two snapshots. The scheduling strategy indicates a point at which the LOD source should be fetched. However, it is possible to retrieve the sources in their order of their assigned preference. Our proposed update strategy construct history of data source for their respective update plan. PASU assigns the preference score to the LOD sources. Based on the score, the update function assign visits the resources that need to be fetched.

The overall flow of the proposed algorithm is shown in Algorithm 2. The input of the algorithm is data source X_t, and then time when the data source was last updated. The proposed scheduling approach assigns preference to each data source based on the previous estimation. Information about the recently changed items are provide by the AACP algorithm. Based on this information, the PASU updates the local data cache. PASU utilized a two phase update policy. In the first phase, PASU quantifies the changes using the Jaccard distance and update from most to least dynamic sources e.g. addition and deletion of triples as discussed in Example 3. We have applied the Jaccard distance; the results out-performs others metrics [72].

Example 3.2. *Quantify the change using the Jaccard distance.* Consider the datasets $X_{Dataset} = \{X_{t1}, X_{t2}, X_{t3}\}$, where $X_{t1}, X_{t2}, and X_{t3}$ represent the snapshots at three different point in time as shown in the Table 2.1. To quantify the changes, $\triangle$-metrics determine the difference in the snapshots such as addition and deletion of the triples [160].

PASU utilized the Jaccard distance, which maps the changes to a real number. The Jaccard distance between the set of triples can be calculated as follows:

$$\triangle(X_{t1}, X_{t2}) = 1 - \frac{|X_{t1} \cap X_{t2}|}{|X_{t1} \cup X_{t2}|} = 1 - \frac{14}{18} = 0.22 \tag{3.4}$$

Example 3.3. *Calculating the last update time of RDF Triples*. In this example, we consider

Algorithm 2: Preference Aware Source Update (PASU)

Input : Data source X and time t
Output: Preference (X,t)

```
for c ← t_i do
    // Update from the most dynamic sources
    if f_dynamics ≥ X_t then
        updatecache ← t_p;
    else
    end
end
// Check that the recent changes occur in the LOD sources
if f(X_i,t_i) recent changes then
    updatecache ← t_{i+1} ;
else
    return f(X_i,K_max) = update(X_i);
end
```

the fact that the data is coming from three different cloud sources: *identi.ca*, *data.gov.uk*, and *ontologycentral.com*. We have retrieved the data at two different time, on January 5^{th}, 2018 and March 10^{th}, 2018 as represented in Eq. 3.5 and 3.6.

$$X_{2018-01-05} = \begin{bmatrix} X_{identi.ca,2018-01-05} \\ X_{data.gov.uk,2018-01-05} \\ X_{ontologycentral.com,2018-01-05} \end{bmatrix} \tag{3.5}$$

$$X_{2018-03-10} = \begin{cases} X_{identi.ca,2018-03-10} \\ X_{data.gov.uk,2018-03-10} \\ X_{ontologycentral.com,2018-03-10} \end{cases} \tag{3.6}$$

Considering that the application maintaining the local data caches was updated on January 5^{th}, 2018, the source of the data has been updated and we want to gain the update on the local copy. Our proposed scheduling will assign preference and fetch the sources to keep the local data cache up-to-date.

Algorithm 3: Change-Aware Maintenance Policy (CAMP)

Input : Query Q, Job Scheduler H
Output: The entire maintenance process P

```
// Creation of the maintenance process
```
$V^q = createView(Q)$
for $(i = 0; i \leq Maintenance; i++)$ **do**
  $P = MaintenancePolicy(Q, Views(V^q))$
  `// Update from the most dynamic sources`
  $updatecache \leftarrow J^r$;
  $J^r = P.release(JobScheduler)$
end
`// Check that the recent changes occur in the LOD sources`
if $f(X_i, t_i)$ *recent changes* **then**
  $updatecache \leftarrow t_{i+1}$;
else
  return $f(X_i, K_{max}) = update(X_i)$;
end

3.3.2 Change-Aware Maintenance Policy (CAMP) Algorithm

To update the view, for each transaction, a maintenance task is created and accumulated in the task queue. The traditional policy is to update each job one by one regardless of the time consumption. However, in our approach, we combine each small task into a larger maintenance job. The maintenance job will be executed only once. Our maintenance manager eliminates redundant update tasks. Therefore, reducing the size of the update stream. The maintenance job begins when the system has a freecycle. We hide the maintenance process from incoming queries to improve the performance of response time. We use CAMP [23, 39] to determine when a local view needs to be updated. CAMP identifies the highly dynamic sources in the Linked Data that are frequently updated and captures these changes to updates the local views. CAMP visits the data sources based on a preference score. Highly dynamic sources should be fetched at a high priority. The preference score is based on the history of the sources. The preference score is computed as shown in Eq 3.7:

$$f_{PASU}(X) = \sum_{i=0} \frac{\Delta(\mathrm{X}_{\mathrm{t}_{\mathrm{lastupdate}-1}}, \mathrm{X}_{\mathrm{t}_{\mathrm{lastupdate}}})}{\mathrm{t}_{\mathrm{lastupdate(X)}} - t_{lastupdate(X)-1}} \tag{3.7}$$

In Eq 3.7, $t_{lastupdate}$ is a function that returns the time when Linked Data was last updated. This function is recursive, as $t_{lastupdate-1}$ returns the time before the last update, and Δ quantifies the

amount of the changes such as addition and deletion of items. The maintenance policy indicates when the Linked Data source should be fetched. However, it is possible to retrieve the sources in the order of their assigned preference. The overall flow of the proposed algorithm is shown in Algorithm 3. The CAMP algorithm first generates the view for the incoming queries as shown in Line 1. As an input parameter, CAMP takes a query model Q and job scheduler, which indicates the time to schedule the resources. CAMP also chooses the appropriate policy to facilitate the performance of the maintenance policy P. The maintenance manager releases a policy J^r to update the local views before the query evaluation in Line 6 and 7 in a maximum execution time K_{max}.

When a query is registered with the system the maintenance manager creates a view and selects the appropriate policy to keep the view up-to-date. These policies invoke the job scheduler to select the relevant stream data and store the appropriate results in the view. A maintenance policy P consists of the sequence of the job needed to update the local views. To be able to keep track of all the views this module maintains hash tables of each view along with the maintenance tasks. To keep track of which version of the data is needed by the views, this module keeps track of the status of the maintenance task. The maintenance of each job is scheduled as a low priority background job and the maintenance manager combines multiple tasks to schedule as a larger job to be executed when the system is in an idle state. When a maintenance job is completed, the manager removes the job from the list and releases the remaining pending tasks. Similarly, in the query execution phase, we first check whether the views used by the query contain any pending maintenance jobs. In case of a pending task, the query requests the manager to perform the task to update the local views. This process is similar to on-demand maintenance, where the query waits until the maintenance process is completed.

3.4 Summary

The Web of Data continuously growing at an alarming rate and there is a need for a flexible way to query. Public endpoints are available to query, but the major problem is poor availability under high querying loads. Existing change detection approaches only focus on the structure similarity, whereas our approach introduces a distance-based query similarity metric, which considers both content-wise and structural similarities for more accurate comparison between queries. It is ad-

vantageous to maintain a local cache for efficient querying and processing. Due to the continuous evolution of the LOD cloud, local copies have become outdated. In order to utilize the best resources, improvised scheduling is required to maintain the freshness of the local data cache. We have proposed an approach to efficiently capture the changes and update the cache. Our proposed approach, called application-aware change prioritization (AACP), consists of a change metric that quantifies the changes in LOD, and a weight function that assigns importance to recent changes. We have also proposed a mechanism to update policies, called preference-aware source update (PASU), which incorporates the previous estimation of changes and establishes when the local data cache needs to be updated. To reduce the overhead cost, modern database systems maintain a cache of previously searched content. The challenge with Linked Data is that databases are constantly evolving and cached content quickly becomes outdated. To overcome this challenge, we propose a Change-Aware Maintenance Policy (CAMP) for updating cached content.

Chapter 4

Prefetching and Cache Replacement

4.1 Overview

We proposed an offline analysis to identify similar structure queries. The result of the previously issued queries is extracted from the log. By using query matching, we prefetch the results of similar queries and place them in a cache for future access. This approach lowers the burden on SPARQL endpoints. To find a similar query, we compute the *Levenshtein Distance* between two queries, normalized by the size of the query strings. We define the Distance Score as:

$$DistanceScore = \frac{Levenshtein(Q_1, Q_2)}{max(|Q_1, Q_2|)} \tag{4.1}$$

In equation 4.6, we define the distance score to match the triple patterns. The overall distance of the triple pattern is calculated by aggregating the individual score of the *subject*, *predicate*, and an *object*. Only the structure base similarity is not sufficient to return a search result, as it only relies on the ordering of symbols. It is possible that two queries represent the same structure of order, but share a different content. We utilize the distance function score to determine the query matching. The query distance between the two graphs is the minimum number of edit operations, such as addition, deletion, and insertion, to transform one graph into another. However, the cost is determined by the distance between triple patterns $\Delta(T_1, T_2)$. Complete matching is only possible in the case of bipartite graphs, where *Basic Graph Patterns (BGP)* occur with the same number as for the triple patterns. A maximum matching is determined in polynomial time. We utilize the Hungarian method [10] to obtain the individual triple patterns with the assignment of minimum cost. Therefore, the higher cost of triple matching is set to $\Delta(T_1, T_2) > 1$ which is considered the infinite cost, where there is no matching between the BGP elements. The score derived from the

complete matching is defined as:

$$\Delta(BGP_1, BGP_2) = \frac{\sum(Triple_i, Triple_j \in M_T)\Delta(Triple_i, Triple_j)}{M_T} \tag{4.2}$$

As shown in equation 4.2, M_T assigns the cost to triples. In the case that M_T is finite, there is a valid triple pattern matching scenario exists and if M_T is infinite, no such scenario exists. However, real-world queries are more complex than the basic graph patterns, therefore, we use a bottom-up approach to derive the minimum cost between BGP. We continue to check the alignment of query patterns, in the case of no alignment, then the matching has an infinite cost. In the case of maximal matching, two BGPs can be matched if they are aligned, and matching with finite cost can be established between all of the patterns.

4.2 Query similarity

The existing approach [11] relies on the structural similarity of the query for improving the performance of the triple stores. We argue that the structural similarity is based on the ordering of the symbols and it is not sufficient as two queries may represent the same structure of ordering but share a different content.

To overcome this drawback of existing methods, we propose a query similarity metric that considers both content-wise and structure similarity. Consider the two queries illustrated in Fig. 4.1. Two queries Q_1 and Q_2 have a similar-structured if the ordering of their symbols is the same. To determine the similarity between two query patterns, we first compute the Levenshtein distance between their query patterns. Where the Line number 6,7 and 8 shows the triple patterns exits in the query. Here, the most similar triple patterns can be determined by computing the minimum distance between the $\Delta(s_1, s_2)$, $\Delta(p_1, p_2)$, and $\Delta(o_1, o_2)$. The composition of the SPARQL query contains a number of different patterns. To find the similarity between queries, we need to decompose the SPARQL queries into subgraph patterns. The triple pattern distance is the minimum number of edit operations, such as addition, deletion, and insertion, to transform one graph to another. We introduce three functions *AND*, *UNION*, and *OPTIONAL*. These patterns take the input graph and decomposed it into three sets of non-empty patterns. As an example, consider

```
1  # PREFIX DECLARATIONS
2  PREFIX :<http://dbpedia.org/resource/>
3   PREFIX dbpedia2:<http://dbpedia.org/property/>
4   PREFIX dbpedia:<http://dbpedia.org/>
5   SELECT ?city ?influence
6   WHERE {
7   ?city1 rdfs:label "Paris" .
8   ?person ?relationshipWith :city1 .
9   :Auguste_Comte foaf:givenName "Auguste" .
10  }
```

Figure 4.1: Example of a SPARQL Query with BGP_1

```
1  # PREFIX DECLARATIONS
2  PREFIX :<http://dbpedia.org/resource/>
3  PREFIX dbpedia2:<http://dbpedia.org/property/>
4   PREFIX dbpedia:<http://dbpedia.org/>
5   SELECT ?city ?influence
6   WHERE {
7   :Auguste_Comte foaf:surname "Comte" .
8   ?city2 rdfs:label "Montpellier" .
9   ?Auguste_Comte ?association :city2 .
10  }
```

Figure 4.2: Example of a SPARQL Query with BGP_2

the SPARQL query in Fig. 4.2 that contains the following graph patterns represented as Q_{AND}, $Q_{OPTIONAL}$, and Q_{UNION} and if no such triple patterns exist the result is $\emptyset$.

$$Q_{AND} = \begin{pmatrix} ?philisopher1 & foaf:name & "AugusteComte" & . \\ ?philisopher1 & ?relationshipWith & :Paris & . \end{pmatrix} \tag{4.3}$$

$$Q_{UNION} = \{Q_{AND}, Q_{OPTIONAL}\} \tag{4.4}$$

More formally, we defined a *QueryDecomposition* to deduce whether the decomposition of query patterns exits, as shown in equation (4.5).

$$QueryDecomposition := \begin{cases} \Theta_{UNION}(Q), & iff\ \Theta_{UNION}(Q) \neq \emptyset \\ \Theta_{OPTIONAL}(Q), & iff\ \Theta_{OPTIONAL}(Q) \neq \emptyset \\ \Theta_{AND}(Q), & else. \end{cases} \tag{4.5}$$

To calculate the similarity between two query patterns, we use the Levenshtein distance that is a string metric for assessing the difference between the two sequences. For example, the Levenshtein distance [72] of the two similar-structured queries is in the range of $[0,1]$. The overall distance of the triple pattern is calculated by aggregating the individual score of the subjects, predicates, and objects. The general formula for the distance score is defined as:

$$Distancescore := \begin{cases} 0, \\ \frac{Levenshtein(Q_1,Q_2)}{max(stringlength(Q_1),stringlength(Q_2))} \\ 1, \end{cases} \tag{4.6}$$

By using this distance score in equation (4.6), we can determine the matching between the triples. Consider the triple matching between the two Basic Graph Patterns (BGP) as shown in Figure 4.1 and Figure 4.2. Here the most similar patterns for L_6, L_7, L_8 in BGP_1 are L_7, L_8, L_6 in BGP_2, respectively e.g., L represents the Line number correspond to Figure 4.1 and Figure 4.2. For example, the minimum value for the edit distance is calculated as aggregating the individual distance score of subject, predicate and an object as follows: $\Delta(L6_{BGP1}, L7_{BGP2}) = (0+0+\frac{4}{16}) = 0.75$. Complete matching is only possible in the case of bipartite graphs, where triples occur with the same number as for the triples patterns. Maximum matching can be determined in polynomial time. The matching of the triple is a computationally expensive process and existing approaches [53, 161] do not consider the cost while performing the matching between two queries. In contrast, we use a cost threshold to cut off too expensive matching and utilize the classical algorithm called Hungarian Method [160] to solve the maximum matching of triples with minimum cost. This algorithm computes an optimal solution in a finite time. More specifically, we consider only the contents of which the minimum matching cost does not exceed one. Thus, the maximum matching of the triples $\{(L6_{BGP1}, L7_{BGP2}), (L7_{BGP1}, L8_{BGP2})\}$ has a cost $\frac{0.75+\infty}{2}$, which shows that these BGP_1 and BGP_2 are unfit to match with each other.

4.3 Idea of Query Prefetching

Whenever there is a public data available, data consumers wants to query it, and nothing is more compelling than querying the vast amount of Linked Data. Recently, Linked Data has emerged as

one of the best practices to represent and connect these repositories, also allowing the exchange of information in an interoperable manner. Linked Data not only supports the integration of multiple data from diverse sources but also provide a way to query these datasets. With over 800 million triples are currently stored in DBpedia[1], the search of the specific resources has never been this high before [11].

Despite the increased performance of SPARQL endpoints, the main problem remains due to the low availability of these endpoints as on average the downtime of SPARQL endpoints is more than 2 days each month [55]. A survey conducted on 427 public SPARQL endpoints registered on the DataHub shows the low efficiency of these endpoints with availability rate above 90% [55]. Oftentimes, SPARQL endpoints are not processed the specific workload as efficiently. The major challenge faced by querying the SPARQL endpoint on account of the inherently distributed nature of Linked Data is its high search latency and lack of tools to connect SPARQL endpoints. Accessing Linked Data cloud at query time is prohibited due to high latency in searching the content and limited capability of tools to connect to these databases. Therefore, the performance of retrieving data from Linked Data repositories is one of the major challenges. Although the Linked Data cloud supports SPARQL queries to access data from its publicly available interfaces, a central problem is the lack of trust regarding these endpoints due to network instability and latency. Therefore, the typical solution is to dump the data locally and maintain endpoints to process these data. The data stored at the local endpoints are not up-to-date and require constant updates, therefore, accurately hosting the endpoints requires expensive infrastructure support [56].

Existing research shows that the SPARQL server with high demands is often hard to host and which is further complicated as the endpoints are publicly hosted due to the unpredictable workloads [57]. The current de-facto way of querying the SPARQL endpoint is to utilize the HTTP that is implemented on top. The client sent the request through these endpoints and the server returns the request. Due to the massive data involved servers need to execute a significant amount of work [58]. The SPARQL query processing is different than the regular HTTP processing as the query-based partitioning of resources occurs. Therefore, regular HTTP caching strategies can not be fully applied to the Linked Data scenario. In the recent past, many efforts have been made to

[1]https://www.dbpedia.org/

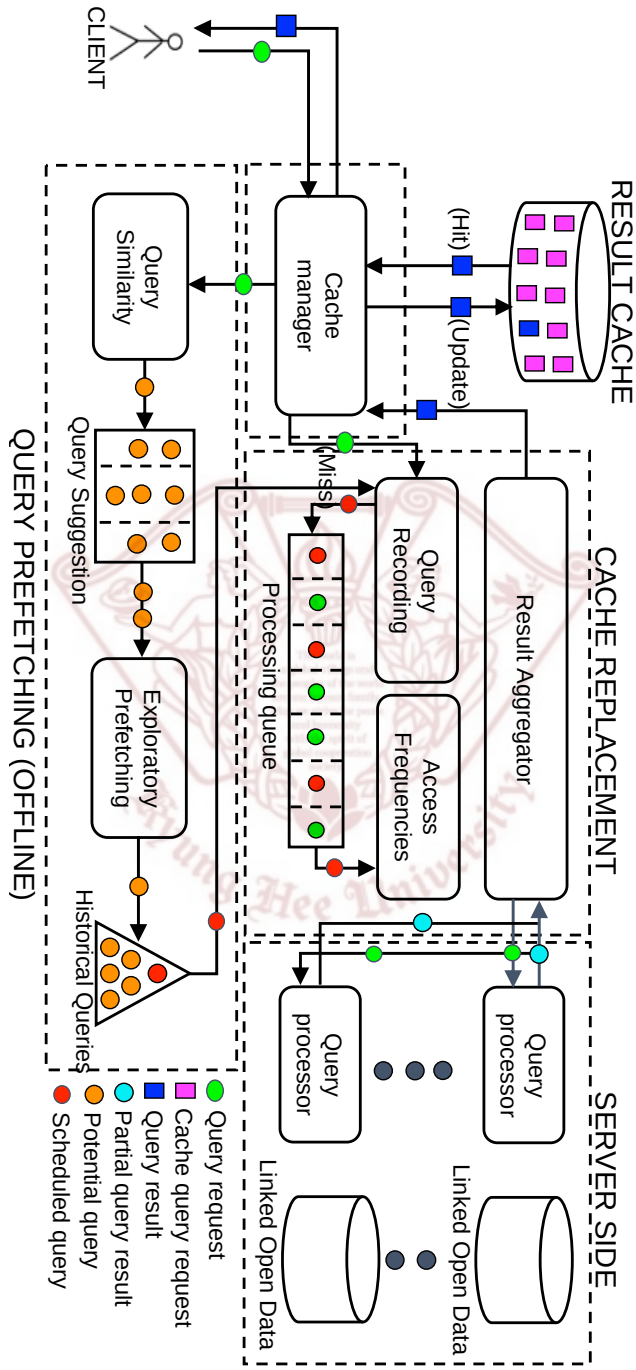


Figure 4.3: Work Flow of Query Prefetching and Cache Replacement

improve the performance of effectively querying Linked Data [59, 60], as compared to querying a triple stored in a relational database, querying a triplestore is still slower by 20% [11]. To circumvent the problem of effectively querying Linked Data [61, 62], caching is the most popular technique to reduce query time by serving the requests from a cache. The idea of query caching is to reuse the previously issued queries. In the distributed setting the benefits of caching are more evident, as the previously issued queries are stored on the remote source. The caching will immediately improve the robustness of the system where the remote resources are not available due to the network insatiability. Existing caching approaches consider caching of the entire query result which means that similar queries can not be served from cache.

Our idea is based on the idea that the clients who processed similar queries are likely to process similar queries in the future. The aim is to search and gather content possibly requested at the future queries by identifying the concepts of the previously issued queries. The main benefit is to reduce the transmission overhead and improve the hit rate and query time by retrieving contents for future queries at once. We prefetch all contents that were used to answer the queries since those queries are more likely to be requested by the same user in the future. For cache replacement, we serve each query according to the estimated frequency, the query with the highest frequencies are kept in the cache. Figure 4.3 illustrates the overall flow of the proposed approach. When a client sends a new query the cache manager first check if an identical query has been cached. In this case, the results are immediately sent to the client. In case of the cache miss, new queries are sent to the query prefetching where similar queries are suggested and the result of these queries are retrieved from the SPARQL endpoints. As an offline process, the result of a similar query is placed in the cache for future access queries. The cache replacement process is triggered when the cache is full and it runs on a separate thread that does not affect the query answering process. The cache replacement is based on the frequency, the higher accessed queries are placed in the cache.

Similar queries occur frequently in real-world SPARQL query logs. This has also been reported previously [69]. The query prefetching approach is suitable for alleviating the burden on SPARQL endpoints by extracting the results of subsequent queries. In the common keyword-based search engines, the user is often not aware of the most suitable keyword to optimally extract information from the resource. In several iterations, the user is more likely to formulate their keyword

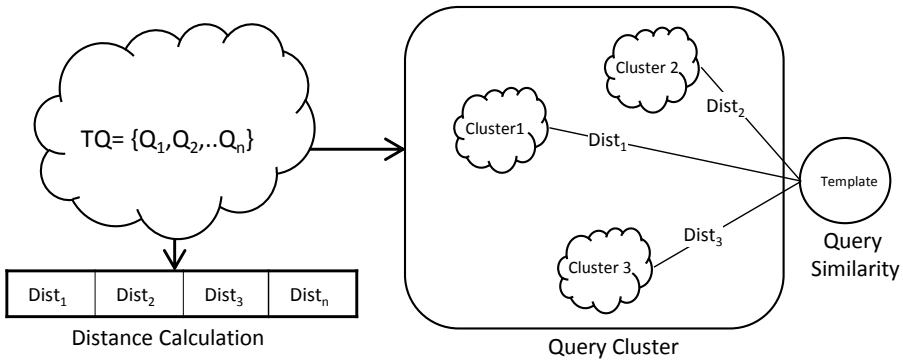


Figure 4.4: Showing the Example of Query Cluster of Similar-Structured Queries

```
1  PREFIX : <http://dbpedia.org/resource/>
2  PREFIX dbo: <http://dbpedia.org/ontology/>
3  PREFIX foaf: <http://xmlns.com/foaf/0.1/>
4  SELECT ?p ?o ?birthPlace ?influence
5  WHERE {
6  :Auguste_Comte dbo:birthPlace ?birthplace .
7  ?birthplace dbo:country :France .
8  :Auguste_Comte dbo:influenceBy ?influence .
9  ?Auguste_Comte ?p ?o .
10 }
```

Figure 4.5: Example of a SPARQL Query Prefteching

to find the correct answer. Similarly, in a LOD user might query for additional details based on the initial results, after making incremental changes to the initial queries.

4.3.1 Finding Structure Similar Queries

To identify the similar-structured queries, we propose a query cluster. Consider $T_Q = \{Q_1, Q_2, .., Q_n\}$ be the set of the SPARQL queries with corresponding query patterns $\{PQ_1, PQ_2, .., PQ_n\}$. A query cluster is defined based on the pairwise matching with three constraints between the triple patterns such as $\Delta(s_i, s_j) \leq 1$, $\Delta(p_i, p_j) \leq 1$, and $\Delta(o_i, o_j) \leq 1$.

Given this definition, the query cluster only derived the query using parameter $\Delta_{max} = 0$ that can only be derived if the two queries are identical. Therefore, a query cluster consists of those query patterns that are structurally the same, based on the corresponding mapping $m \sqsubseteq \Theta(PQ_1) \times (PQ_2)$. To represent the query patterns as a feature, we first cluster queries based on the content-

p	o
rdf:type	dbo:person
dbo:birthDate	1798-01-19
dbo:idea	:Positivism
dbo:influenced	:KarlMarx

Figure 4.6: Showing the Result of the Query

wise and structural similarities as shown in Figure 4.4 and distances between each pair of queries are computed by adopting the *k-medoids* algorithm [162]. We use this algorithm to cluster the training data of the query. This algorithm chooses the data points and allows us to utilize the distance function. To calculate the query distance, we utilize the distance score in equation (4.6) and define the similarity score of the query cluster as shown in the equation (4.7).

$$SimilarityScore(T_Q, Q_c) = \frac{1}{1 + Distancescore(T_Q, Q_c)} \tag{4.7}$$

Furthermore, we introduce an exploratory prefetching. More specifically, we identify a query cluster that previously issued queries belong to and construct a query template using all queries in the cluster. Then, we prefetch all contents that were used to answer the queries in the template since those queries are more likely to be requested by the same user in the future. This prefetching is completed by issuing one single query which includes all queries in the template. For example, we modified the content of the queries previously issued in Figure 4.1 and retrieve all relevant contents that are useful for future queries as shown in the Figure 4.1. This query retrieves the additional information based on the central concept, instead of issuing the many similar-structured queries, prefetching retrieve all the relevant information by issuing a single query.

For extracting the additional information for the specific resource, we propose an Algorithm 4 called central Concept Fetching (CCF) to generate the central concept. In the CCF algorithm, we first discovered the frequency of the subjects in all query patterns in Line 7 and aggregate whether the subject is already included in the *S.Count*. We further increase the count of the subject and add to *S.Count* in Line 11 and analyzed all the triple patterns. This algorithm will analyze all the triple and give a good indication of a common theme in all the SPARQL queries.

Algorithm 4: Central Concept Fetching (CCF)

Input : $T_Q = \{Q_1, Q_2, .., Q_n\}$
Output: Occurrence of most frequent subject

```
1  S.Count← 0
2  foreach Q_p ∈ T condition do
3      S ← Θ(P_Qi)
4      while S ≠ 0 do
5          foreach Q_p ∈ S do
6              if Θ(P_Qi) > 1 then
7                  S ← S ∪ Θ(P_Qi)
8              else
9                  (S, P, O) ← Θ(P_Qi)
10             if S ∈ S.Count then
11                 S.Count.increasecount(S)
12             else
13                 S.Count.put(S,1)
14             end
15         end
16     end
17 end
18 return getHighestCount();
```

4.4 Adaptive Cache Replacement (ACR) Algorithm

We propose an offline process for cache replacement to calculate the access frequency. Logging every record produces the most accurate result, however, it is computationally expensive. Existing approach [33] utilize forward scanning to identify record access with a time slice $[t_n, t_{n+1}]$.

performance, as it requires scanning and storage of the entire record. Moreover, the forward scanning approach requires a significant amount of time to classify the record. To optimize this process, we maintain partial records within a specific time period. We parallelize this task by splitting the logs into n consecutive periods and use a hash function to store the frequency estimation for each query as shown in Figure 4.7. Where Q_1 represents the query and R_1 denotes the results of the query. The estimation of the record is calculated by equation (4.8) and this algorithm ranks each query by its access frequencies. The storage of the access log is placed in a separate hash table. When each of the parallel executions finishes, the results of the most highly accessed fre-

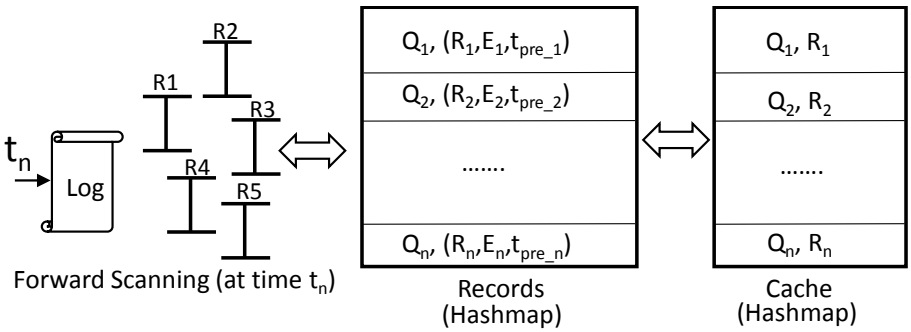


Figure 4.7: Showing Working Example of the ACR Algorithm Maintaining Cache and Query Access Frequency

quencies are returned immediately, and infrequently accessed queries are removed from the cache.

The overall flow of ACR is described in Algorithm 5, which explains the details of updating the cache by analyzing the access logs. The ACR algorithm takes previously access logs to calculate the access frequencies and provide the list of the updated cache triple, where LA_t represents the last access time of the triple and CA_t represents the current access time. ACR algorithm scans the records and updates the frequency. In the case of a cache miss, the algorithm first checks for the case of a record in the cache and updates the LA_t. Based on the access frequency ACR decides whether the new triples need to be added to the cache.

We have calculated the frequency of the data access using the exponential smoothing technique [163]. This method is widely utilized to predict economic data in financial applications. The traditional approach [10] contains all the accessed queries in the cache. In our work, ACR serves each query according to the estimated frequency, the query with the highest frequencies are kept in the cache for future access. The general formula of exponential smoothing is as follows:

$$E_t = \alpha * x_t + (1-\alpha) * E_{t-1} \tag{4.8}$$

As shown in equation (4.8), where E_t represents on exponentially moving average of access frequencies up to time t and x_t represents an access frequency observed at time t in discrete time with the smoothing constant $\alpha \in (0,1)$. The high value of α gives significance to the new observations. By using the equation (4.8), we satisfies our requirement of selecting the highly accessed queries. We further modified equation (4.8) to represent the time of the last hit. In equation (4.9),

Algorithm 5: Adaptive Cache Replacement (ACR)

Input : Query Log Q, Job Scheduler H
Output: List of added cache triples

$t_{latest} \leftarrow max(LA_t, CA_t)$;
$t_{earliest} \leftarrow min(LA_t, CA_t)$;
$Records \leftarrow getRecords(t_{latest}, t_{earliest})$;
Function: $ModifiedForwardAlgo(Q,H)$;
if *newTriples in Records* **then**
    $max(estimation, cachedTripples)$;
    $Calculate(Frequency, LA_t)$;
    $update(Frequency, LA_t)$;
    $Remove = Leastaccessedtriples$;
else
    $Triples$ not in $Records$;
    $Calculate(Frequency, LA_t)$;
    $Add(newAccessTriples$;
end
return $addnewAccessTriples$;

t_{prev} represents the time of the last query hit and $X_{t_{prev}}$ represents the frequency estimate of the previous query at t_{prev}. For example, assume that $\alpha = 0.05$, $t = 12$, $t_{prev} = 3$, $x_t = 0.6$, and $x_{t_{prev}} = 0.5$. The value of E_t is calculated by $E_t = 0.05 * 0.6 + 0.05(1-0.05)^{12-3} * 0.5 = 0.046$.

$$E_t = \alpha x_t + \alpha(1-\alpha)^{t-t_{prev}} x_{t_{prev}} \tag{4.9}$$

In case of the queries that are not similar to the previous ones stored in the cache, the result of these queries is served from the LOD and ACR algorithm store the access frequencies by using equation (4.9). When the cache becomes full, replacement is based on the access score; the top queries are kept in cache and less-frequent queries are removed from the cache.

4.5 Summary

In this chapter, we proposed a client-side caching that works as a proxy between the querying agent and the SPARQL endpoint. To accelerate the querying answering process, our approach can either be deployed within the SPARQL endpoint or querying agent to eliminate the burden on these endpoints. We propose an exploratory prefetching to retrieve contents possibly requested at

the future queries by identifying the concepts of the previously issued queries and issuing a single query for all required contents. Its benefit is to reduce the transmission overhead and improve the hit rate and query time by retrieving contents for future queries at once. For cache replacement, we proposed an exponential smoothing forecasting method to replace the less valuable cache content. We propose a frequency-based cache replacement method to rank each query according to its estimated access frequency. The most frequently accessed queries are kept in the cache. Thus, our work benefits the triple stores in replacing the cache. We also propose an Adaptive Cache Replacement strategy (ACR) that aims to accelerate the overall query processing of the LOD cloud. ACR alleviates the burden on SPARQL endpoints by identifying subsequent queries learned from the client's historical query patterns and caching the result of these queries.

Chapter 5

Evaluation and Results

To validate the proposed model, we performed extensive experiments on real-world datasets. The major goal of the experiment is to examine the hit rates and overhead comparison of the proposed approach with the current state-of-the-art cache replacement approaches.

5.1 Experimental Results related to Change Detection for Evolution Analysis

This section is devoted to analysis and comparison with other approaches. We first describe the setup of our experimentation, dataset details, analysis of dynamic sources, and comparison with the existing approaches. Our experimental procedure follows the work of Dividino et al. [24], however, we have evaluated the effectivity to get further insights related to runtime overhead of our proposed approach, and the results outperform as compared to other approaches. We have used precision and recall to check the effectiveness of our proposed approach.

$$Precision = \frac{\sum |X_{c,t} \bigcap X'_{c,t}|}{\sum X'_t |X'_{c,t}|} \tag{5.1}$$

$$Recall = \frac{\sum |X_{c,t} \bigcap X'_{c,t}|}{\sum X'_t |X_{c,t}|} \tag{5.2}$$

5.1.1 Experimental Setup

The comprehensive quality of our approach is conducted using real-world datasets. We applied our proposed approach to real Linked Datasets. All the experiments were performed on a 4x AMD A8-7650K Radeon R7, 64bit Ubuntu 16.04.2 LTS, and OpenLink Virtuoso Server 07.10 with 16

GB RAM.

In our experiment, we used the DYLDO[1] and BTC[2] datasets. Both of these datasets are serialized in the N-Quad format. To parse both of the datasets, we used NxParser[3]. We have calculated the dynamic score of both datasets as shown in Table 5.3 and Table 5.4. The following are the details of both datasets.

DYLDO. As the first dataset, we use DYLDO [1,44], which is composed of 149 weekly crawls. DYLDO contains various well known sources such as *dbpedia.org*, *identi.ca*, and *dbtropes.org*. On average, the size of DYLDO of every snapshot is approximately 1.35 GB (for three years, it is approximately 36 GB). From the original dataset, we parse the dataset and extract the triples from each snapshot. The total number of data sources varies in each snapshot.

BTC. This data was collected from the Multi-crawler framework [164] collected for the Billion Triple Challenge[4]. The dataset was crawled from January 2014 to July 2014. The size of the dataset varies in each crawl. In our experiment, we evaluated our approach on three different snapshots. The size of the first snapshot is 3.7GB (46 GB unzipped), the second is 3.7GB (59GB unzipped) and the third snapshot size is 2.1GB (44GB unzipped). We observed that most of the changes (73%) occur in a data source with more than 10.8 million triples.

Table 5.3: Dynamic score of the DYLDO sources

PLD	Dynamic Score	AACP Score
Identi.ca	58.99	19.65
loc.gov	46.63	14.33
Linkedct.org	51.69	21.36
Dbpedia.org	53.63	19.86
Dbtropes.org	66.90	28.80
Neuinfo.org	41.30	10.96

[1]http://swse.deri.org/dyldo/data/
[2]http://km.aifb.kit.edu/projects/btc-2014/
[3]https://www.w3.org/2001/sw/wiki/NxParser
[4]http://km.aifb.kit.edu/projects/btc-2014/

Table 5.1: Dynamics of the DYLDO dataset

PLD	**Description**	**Linked Data Category**	**Avg. Triples per Snapshot**	**Avg. stmt added**	**Avg. stmt remove**
Identi.ca	Open source Social engine	Cross domain	1,341,045	2930	2563
loc.gov	Library of Congress	Cross domain	369,884	222	242
Linkedct.org	Live data browser	life sciences	1,782,886	4263	3529
Dbpedia.org	DBpedia Project	Cross domain	4,080,910	5414	45518
Dbtropes.org	Online wrapper	Media	1,729,455	381	408
Neuinfo.org	Neuroscience information	Life sciences	2,065,028	2630	2108

Table 5.2: Dynamics of the BTC dataset

PLD	Description	Linked Data Category	Avg. Triples per Snapshot	Avg. stmt added	Avg. stmt remove
ontologycentral	Open Source Social engine	cross domain	55,124,003	2930	2563
bio2rdf	Live data browser	life sciences	20,168,230	4263	3529
data.gov.uk	Dbpedia Project	cross domain	13,302,277	5414	45518
dataincubator	Online wrapper	media	1,729,455	381	408
freebase.com	Neuroscience Information	life sciences	25,488,720	2630	2108
Berkeleybop	Berkeley Bioinformatics project	life sciences	903,318	1726	908

Table 5.4: Dynamic score of the BTC sources

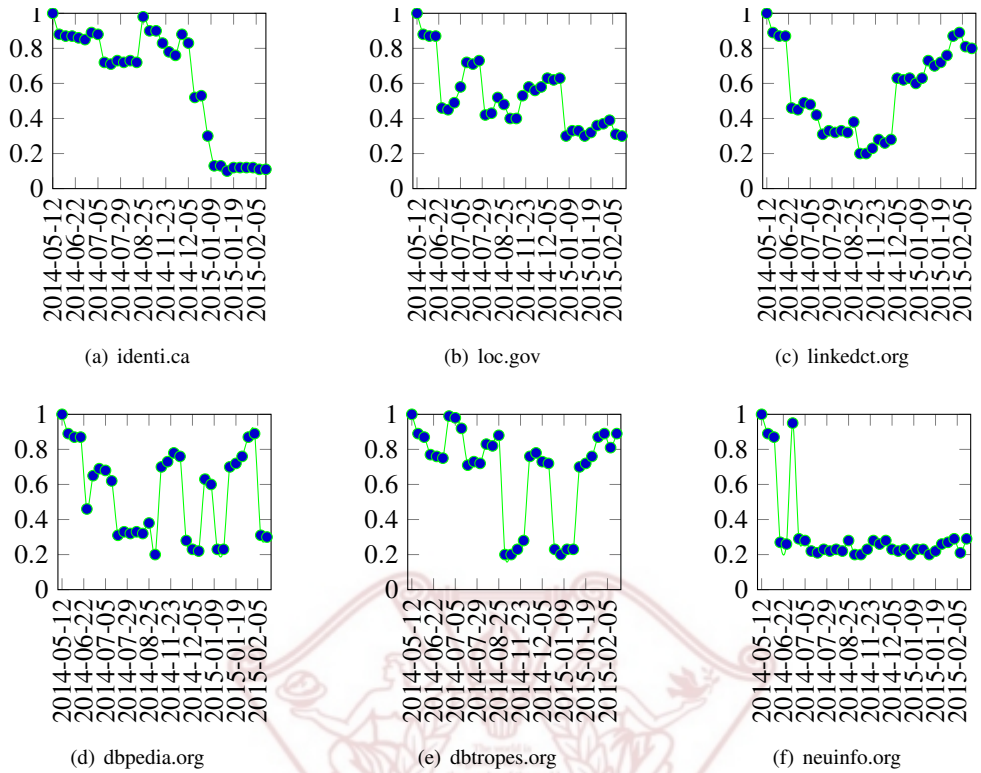
PLD	Dynamic Score	AACP Score
ontologycentral.com	54.30	21.30
bio2rdf.org	49.63	19.30
data.gov.uk	68.30	30.15
dataincubator.org	57.56	25.63
freebase.com	34.39	18.53
berkeleybop.org	31.10	17.63

The content in both of the datasets is subjected to frequent changes. From the available snapshots, we have selected the top five PLD from both of the datasets as shown in Table 5.1 and Table 5.2. Both of the sources evolved. We have categorized both the datasets. In each of the snapshots, we have calculated the average addition and deletion of the statements. We have applied the dynamic function on the most frequently changed sources. To identify the most recent changes, we have applied the AACP algorithm as shown in the Table 5.3 and Table 5.4. In the next section, we perform analysis to identify the most dynamic sources.

5.1.2 Analysis of the Dynamic Sources

Due to the heterogeneous nature of the Linked Data sources, we expect dynamic behavior in time. Some sources evolved while other sources did not change. Similarly, we expected a wide range of changed behavior in the period from 2014-05-12 to 2015-02-05. Some of the sources have evolved more than the other sources, as shown in Fig. 5.1, such as identi.ca (in Fig. 5.1(a), which consist of more than 4 million triples per snapshot. Other sources, such as neuifo.org (in Fig. 5.1(f)) are very small and consist of 10,000 triples per snapshot.

From all of the snapshots of DYLDO, we have computed the dynamic function over a period of time. The dynamic function will measure the change in the source that has gone through i.e., addition, and deletion of triples. Linked Data sources, in particular loc.gov (in Fig. 5.1(b)), linkedct.org (in Fig. 5.1(c)) and dbtropes.org (in Fig. 5.1(d)), show a higher change rate over the considered period of time, whereas neuinfo.org (in fig Fig. 5.1(f)) shows a lower change rate. We observed



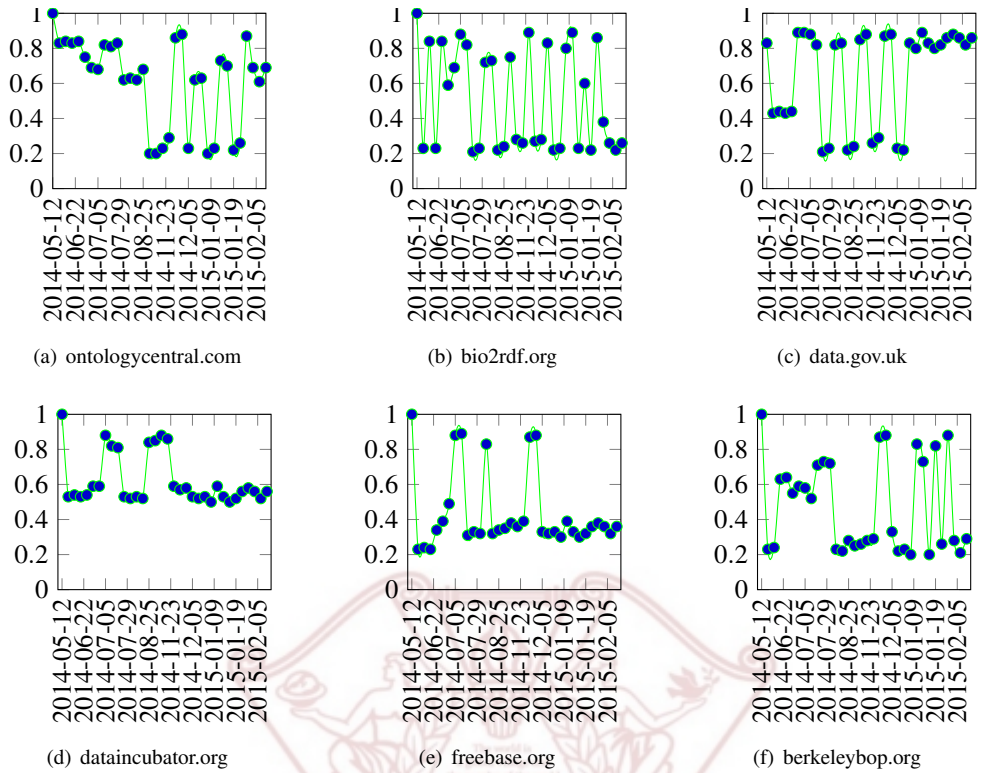
(a) identi.ca (b) loc.gov (c) linkedct.org

(d) dbpedia.org (e) dbtropes.org (f) neuinfo.org

Figure 5.1: Jaccard Distance Plots for the DYLDO Dataset.

small peaks in identi.ca (in Fig. 5.1(a)), where the changes occurred only in the early period of time and then the sources remained almost constant. Our dynamic function also suggests that the sources linkedct.org (in fig Fig. 5.1(c)) and dbtropes.org (in Fig. 5.1(e)) shows higher change rates in the latest period of time.

A similar trend has been noticed in the BTC datasets, as shown in the Fig. 5.2. Sources such as ontologycentral.com (in Fig. 5.2(a)) show a decreasing change rate, which later start increasing. The sources data.gov.uk (in Fig. 5.2(c)) and dataincubator.org (in Fig. 5.2(d)) are highly dynamic and their change rates are higher than other sources. Some of the sources changed recently, i.e., data.gov.uk (in Fig. 5.2(c)) and dataincubator.org (in Fig. 5.2(d)), showing higher change rates. We have summarized the dynamic score and the AACP score, as shown in Table 5.3 and Table 5.4. According to our results, the source Dbtrobes.org (in Fig. 5.1(e)) showed the highest dynamic score of 66.90 and AACP score of 28.80. Some of the sources were dynamic, e.g., data.gov.uk (in



(a) ontologycentral.com (b) bio2rdf.org (c) data.gov.uk

(d) dataincubator.org (e) freebase.org (f) berkeleybop.org

Figure 5.2: Jaccard Distance Plots for the BTC Dataset.

Fig. 5.2(c)), with 68.30 and an AACP sore of 30.15.

5.1.3 Comparison with existing approaches

In this section, we briefly discuss the existing update approaches. We have evaluated our approach in two different scenarios. In the single setup scenario, we utilized the quality of the updates performed for a single iterative update. We also noticed the runtime overhead while executing the updates. In the iterative setup, the quality of the update was measured over a long period of time. We have utilized precision and recall as our metrics.

5.1.3.1 Age

Most of the update strategies use *Age*, as it is featured to capture the age of the data source. Based on the age the preference is assigned to fetch the sources. Age is an important feature that

illustrates when the data source was last visited and updated [36].

$$f_{age}(X_{t,c}) = t_i - t_{last}(X_{t,c}) \tag{5.3}$$

As shown in Eq. 5.9, age is used by the scheduling strategies to fetch the data source c in time t_i, whereas, t_{last} is the last update time.

5.1.3.2 PageRank

This updates the sources from highest to lowest priority of the Linked Data sources [165]. *PageRank* is represented as:

$$f_{PageRank}(X_{t,c}) = PR(X_{c,t_{lastupdate}}) \tag{5.4}$$

5.1.3.3 Size

The *size* shows the number of the triples provided by the data source. Most of the update function prioritizes the sources based on the size of the data item, e.g., it updates from largest to smallest datasource [37]. The formula for *size* is as follows:

$$f_{size}(c, X_{t_i}) = |X_{c,t_{lastupdate}}| \tag{5.5}$$

5.1.3.4 ChangeRatio

The *ChangeRatio* provides how many changed items in the Linked Data sources have changed from the last known time period [166].

$$f_{ChangeRatio} = |X_{c,t_{lastupdate}}| + |X_{c,t_{lastupdate}}| \tag{5.6}$$

Table 5.5: Evaluating the effectivity of the update strategies

Update Strategies	Total Query Execution	Irrelevant	Relevant	Effectivity (%)	Runtime (sec)
PageRank	32,690	30,650	2,040	6.24	800
Size	28,521	16,448	12,073	42.3	650
Age	29,128	10,523	18,605	63.9	500
ChangeRatio	29,550	9,800	19,750	66.8	320
ChangeRate	29,560	4,500	25,060	84.7	120
PASU	29,565	1,900	27,756	93.5	32

5.1.3.5 ChangeRate

This update function quantifies the change by comparing the two snapshots. The *ChangeRate* of the Linked Data can be represented as Δ, which quantifies the changes between two datasets or the distance between two data sources. The scheduling strategies based on the *ChangeRate* quantifies the evolution of the Linked Data over the period of time [26]. It is represented as follows:

$$f_{ChangeRate} = \sum_{i=0}^{j} \frac{\Delta\left(X_i, X_{t_{i-1}}\right)}{t_i - t_{i-1}} \tag{5.7}$$

We have evaluated the effectiveness of our approach in two different setups, e.g., single and iterative setup. In the single setup, we have considered the effectiveness of our approach for a single update. We have compared our approach with the state-of-the-art approach in the iterative scenario considering the update of the local data cache over a longer time. Moreover, we have discussed the runtime overhead comparison in Table 5.5.

5.1.4 Performance Evaluation

In this performance evaluation, we performed two different setups. In the single-step scenario, we utilized the quality of the updates performed by the update strategies for a single update, i.e., starting from the accurate copy of the sources. The goal of the iterative setup was to estimate the accuracy, i.e., how good is the updated policy for maintaining an accurate local data cache.

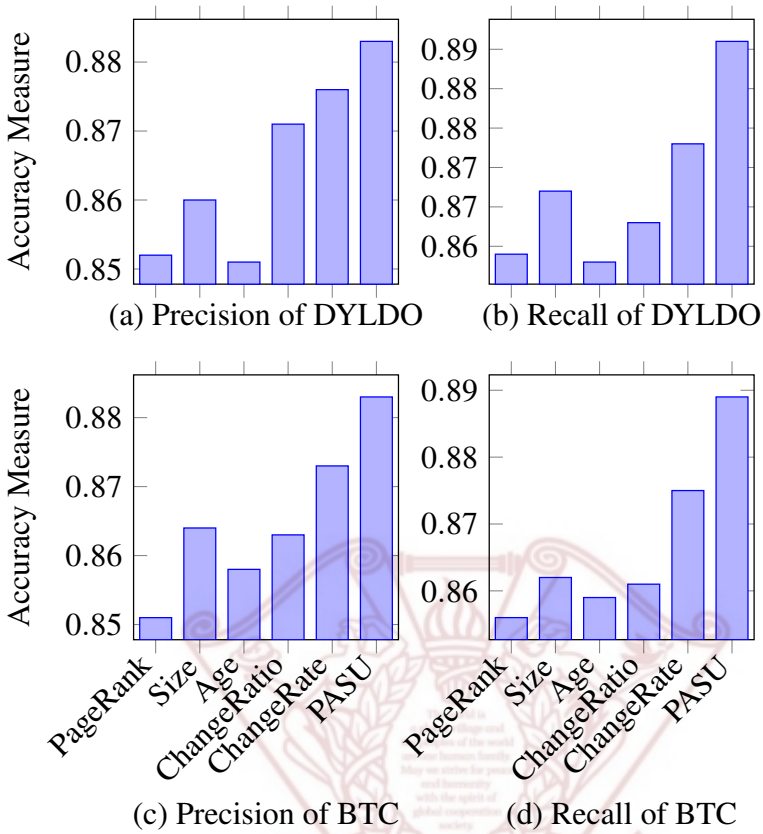


Figure 5.3: Single Setup: Comparison with Other state-of-the-art Strategies.

5.1.4.1 Single Setup

First, we started with the perfectly up-to-date cache and then assumed the change occurred at time t_i. We have evaluated how the existing and proposed strategies update the local copy of the sources, as shown in Fig. 5.3. Although in both of the datasets our proposed dynamic function based strategy out-performed *ChangeRate*, *ChangeRatio*, *Age*, *Size*, and *PageRank*, all these strategies showed a uniform loss of quality. After a single update, we have achieved 88% (F-measure) accuracy.

We observed that our approach only executes the relevant data updates with less drop and delay. All the other strategies execute a massive amount of overhead, resulting in a low effectivity. To obtain further insights regarding the runtime overhead of the proposed approach, we have

extended our previous evaluation results [19]. We utilized the Linked SPARQL Queries (LSQ)[5] [167] that was extracted from the access log of a public SPARQL endpoint. We selected LSQ because it matches our current structure of the DYLDO and BTC datasets. In Table 5.5, we have summarized the result of the update strategies. The effectivity and the runtime overhead were set as the key metrics. We have calculated the effectivity as follows:

$$Effectivity(\%) = \frac{R_{relevantquery}}{T_{totalexecution}} \tag{5.8}$$

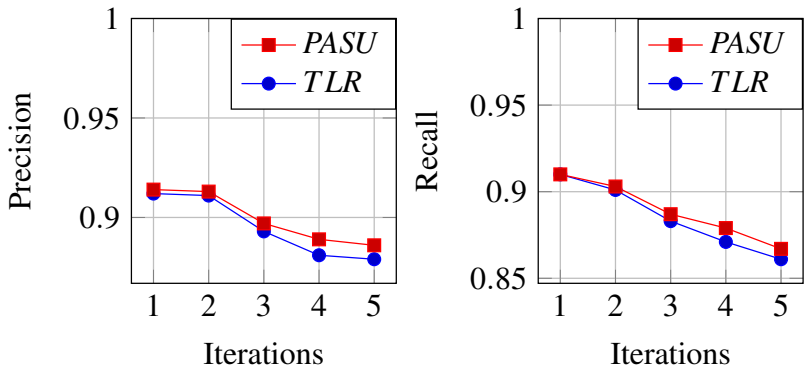
Since irrelevant executions create unnecessary load, among all the strategies, *PageRank* is the least effective with only 6.24% (32,690 vs. 2,040) and a longer runtime overhead as compared to all other strategies. The strategies *PageRank*, *Size*, and *Age* showed the worst results. These strategies executed irrelevant queries and could not detect all the changes. In contrast, *ChangeRatio* and *ChangeRate* executed less irrelevant queries but the runtime overhead was high. *PASU* outperforms other approaches with an effectivity of 93.6% .

5.1.4.2 Iterative Setup

In this setup, we compared our approach with the state-of-the-art approach called TLR. We assumed that the data is fetched at a fixed point in time t. We denote the size of the dataset that is fetched from the source by $|X_{c,t}|$, which contains a number of triples in a dataset at context c. We aim to measure how well the update policies perform in terms of maintaining an accurate local copy at times $t_{i+1}, t_{i+2} \ldots t_{i+n}$.

A comparison of the proposed and the existing approaches in the iterative setup is shown in Fig 5.4. We have analyzed that our approach outperforms the baseline approach called Triple Linear Regression (TLR) [23]. In the iterative setup, we look at the precision and recall of both approaches. We noticed a drop in the quality of the TLR approach along with the iteration. In the first iteration, the TLR approach achieves 0.903 precision and drops along with three iterations by 0.871. We noticed a better precision score in our proposed approach. Our precision score in the first iteration was 0.903 and dropped along with the iteration by 0.891. Similar trends exist in the recall score. The recall drops from 0.901 to 0.871 along with iterations. In our proposed approach,

[5] http://aksw.github.io/LSQ/



(a) Precision of DYLDO (b) Recall of DYLDO

(c) Precision of BTC (d) Recall of BTC

Figure 5.4: Iterative Setup: Comparison with Other state-of-the-art Strategies.

the recall score was 0.916 and the drop was 0.881, but it is less affected by the dynamic nature of the Linked Data cloud.

The proposed approach is reported to outperform in the single and iterative setups. We also compared the effectiveness of our approach with other state-of-the-art approaches. In the single setup, we have evaluated the quality of the updates of the local data cache based on precision and recall, and our approach achieved 88% (F-measure) accuracy and precision and recall from 0.883 to 0.890 and from 0.884 to 0.894, respectively. To check the effectiveness of our proposed approach, we have observed less runtime overhead, and the proposed scheduling strategy outperforms with effectivity of 93.6%. In the iterative setup, we have evaluated the quality of the updates performing over a longer period of time. We noticed a drop in the quality of existing as compared to our approach. The effect of the dynamic nature of LOD on our proposed approach is minimum

and it can be utilized in practical application scenarios of LOD.

5.2 Experimental Results related to Prefetching and Cache Replacement

This section is devoted to showing the effectiveness of the proposed approach. We performed an evaluation on real-world datasets. The major goal of the experiment is to examine the hit rates and overhead comparison of the proposed approach with the current state-of-the-art cache replacement approaches.

5.2.1 Experimental setup

We conducted the experiments on an OpenLink Virtuoso Server 07.10 with a 4x AMD A8-7650K Radeon R7 graphics card, 64bit Ubuntu 16.04.2 LTS, and 32 GB of RAM. We utilized the *DBpedia3.6* and *Linked Geo Data (LGD)* query logs provided by the USEWOD 2014 challenge[6]. The query log contains a number of requests received by the SPARQL endpoint. The log is formatted in the form of the Apache common log format and contains the information about the query session that is used to retrieve the data from the endpoint. It is possible that in a single query session, two queries are issued by the same user over time. The requests included in the *DBpedia3.6* query logs include the timestamp. The query log contains IP address, timestamp, query and userID. The valid queries were extracted from the query logs and the syntax of the query was checked according to the SPARQL1.1 specification.

The *DBpedia3.6* dataset contain the structure information extracted from the WIKIpedia and published over the LOD cloud. This dataset is obtained directly from the USEWOD query logs for DBpedia 2013, 2014 and 2015 as shown in the Table 5.6. The *DBpedia3.6* KBs contain 3.0M entities about the general knowledge. We first extracted the textual information from the log to get the previously issued queries then parse each query using *Apache Jena*[7].

The *Linked Geo Data (LGD)* dataset holds the geographic sensor information mainly related to the OpenStreetMap and it is currently available as RDF format. We utilized the LGD 2013

[6]http://usewod.org/usewod2014.html
[7]https://jena.apache.org/

Table 5.6: Showing the size of the query logs used in our evaluation

Source	Total queries	Valid queries	Unique queries
DBpedia 2013	28,423,201	27,563,105	12,326,855
DBpedia 2014	4,132,742	3,708,727	1,517,002
DBpedia 2015	31,345,875	30,245,552	3,258,671
LGD 2013	1,721,770	1,512,785	247,731
LGD 2014	1,730,770	1,513,895	517,530
Total	67,354,358	64,544,064	17,867,789

and 2014 that consist of more than 10 billion triples. From the available LGD query logs, our evaluation contain repetitive and unique queries.

In both datasets, the majority of the queries are the SELECT queries in the DBpedia, and LinkedGeodata logs and within these SELECT queries, we identified the occurrences of BGPS, as in Figure 5.5, which shows SELECT, CONSTRUCT, DESCRIBE and ASK. Most of the queries in both datasets are SELECT queries (95% in DBpedia and 89.3% in LinkedGeoData) and the most widely used features are ASK (4.4% in DBpedia and 8.4% in LinkedGeoData) followed by CONSTRUCT (1.2% in DBpedia and 2.3 % in LinkedGeoData).

Figure 5.6 shows the impact of the unique query account for the query execution. Our aim is to ascertain the impact of the unique and frequently executed queries on the overall execution. We analyzed the execution using the DBpedia and LGD query logs. In DBpedia, 70% of the unique queries account for 30% of the overall executions, which shows that most of the execution instances involved the frequently accessed queries. Similarly, the impact of the unique queries on the overall execution is low, as almost 90% unique queries account for the 20% of the total query executions.

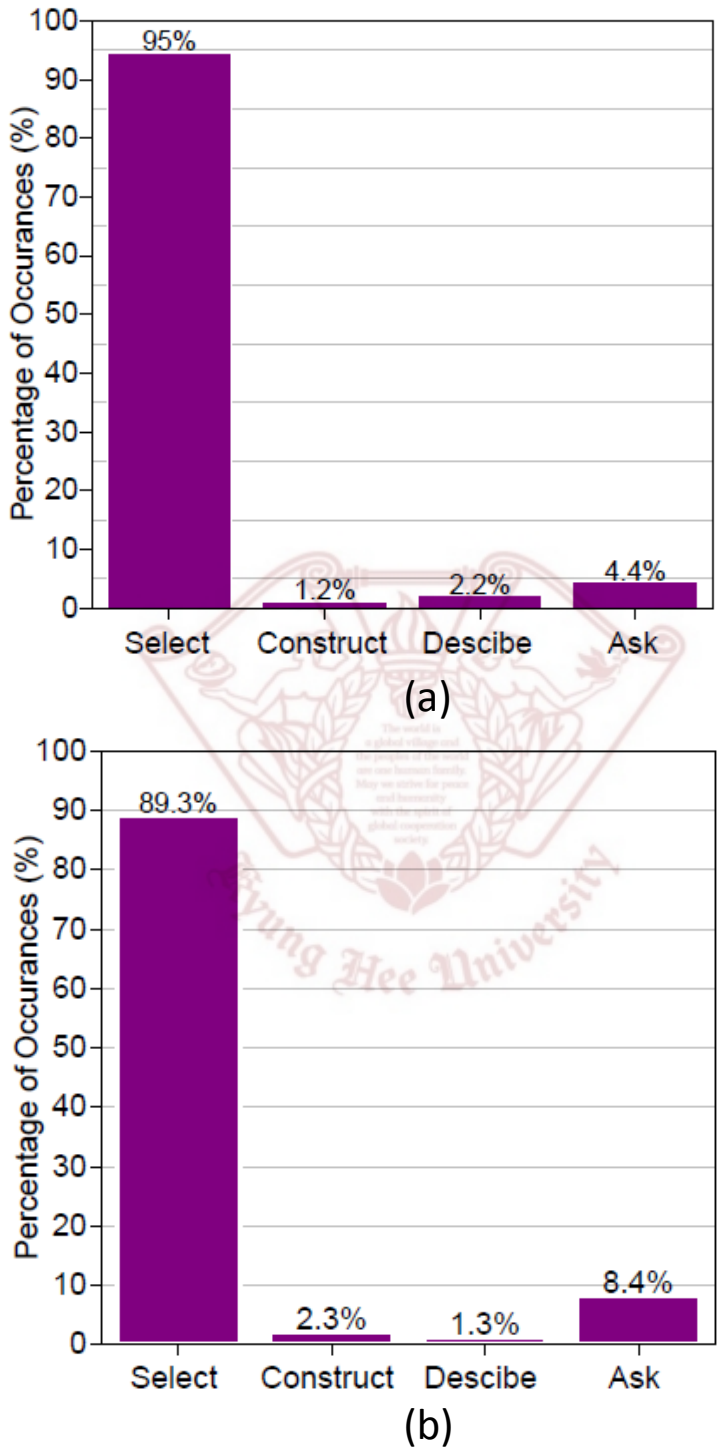


Figure 5.5: Showing the Patterns of the Queries in (a) DBpedia and (b) LinkedGeoData

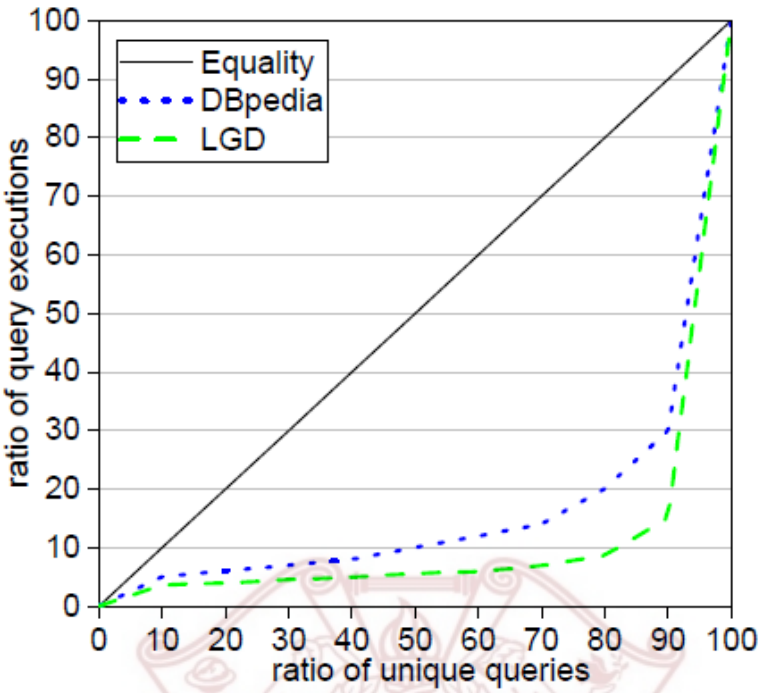


Figure 5.6: The Lorenz Curve for the Impact of a Unique Query on Query Execution

5.2.2 Comparison with Existing Approaches

In this evaluation, we compare ACR with existing approaches, such as LRU (Least Recently Used) [168], LFU (Least Frequently Used) [32], and SQC (SPARQL Query Caching) [11] and measure the efficiency in terms of average hit rate and space overhead.

5.2.2.1 Least Recently Used (LRU)

We evaluate the impact of existing cache replacement algorithms to improve the performance in terms of hit rates and overhead. Therefore, we compare ACR with three well-known cache replacement approach, Least Recently Used (LRU) is applied to remove the items from the cache in order to provide space for the new item. This approach is simple to implement, especially when the objects are uniform.

5.2.2.2 Least Frequently Used (LFU)

The Least Frequently Used (LFU) resources are removed from the cache and the cache item is replaced with a new resource. However, LFU does not consider the size of the objects and CPU memory utilization.

5.2.2.3 SPARQL Query Caching (SQC)

PARQL Query Caching (SQC) [11] improves the performance of triple stores by the selective invalidation of cache objects. This approach eliminates the cache objects that do not contain the predefined timestamp.

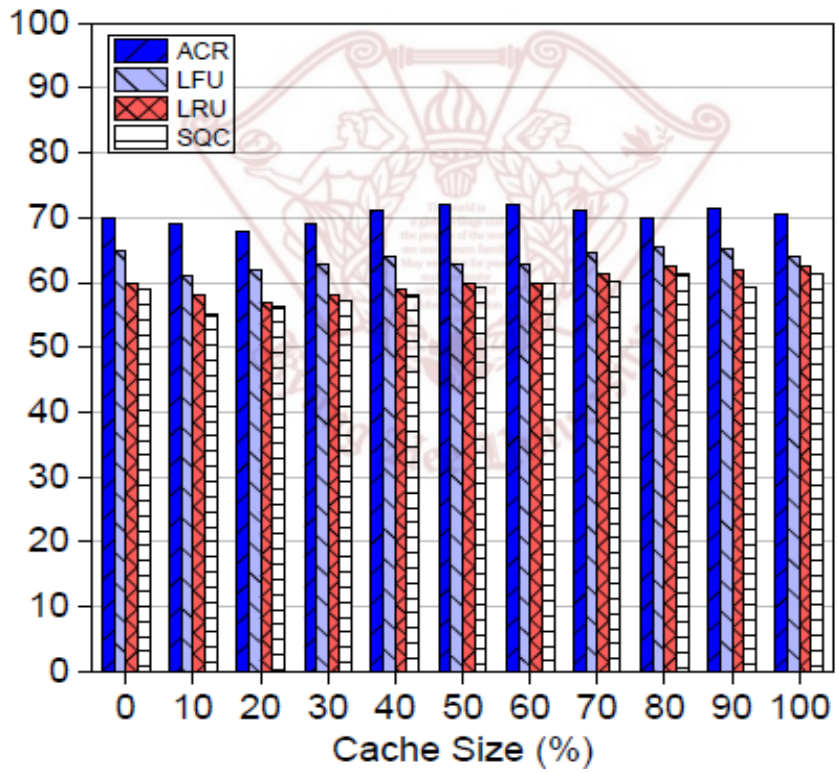


Figure 5.7: Hit Rate Achieved by Varying the Size of the Cache as Compared to Existing Approaches

5.2.3 Performance Evaluation

Figure 5.7 shows the hit rates achieved by the existing approaches. This experiment feeds the access logs of 3M to the ACR algorithm, whose job is to rank the access frequencies of the queries based on the exponential smoothing technique. It is noted that ACR outperforms existing approaches. However, the LFU technique remains accurate for a cache with a small size. The choice of the α effects the performance of the hit rate. We have set the value to 0.05 due to the higher accuracy of the results obtained, as the optimal value of α is almost certainly inversely proportional to the size of the cache, and perhaps related to the size of the database. If the cache is smaller, α should probably be larger as shown in Figure 5.9. Upon varying the size of the cache, our proposed approach outperform the other approaches, as shown in Figure 5.8. On average, our approach outperforms existing approaches in terms of higher hit rates, up to (80.65%).

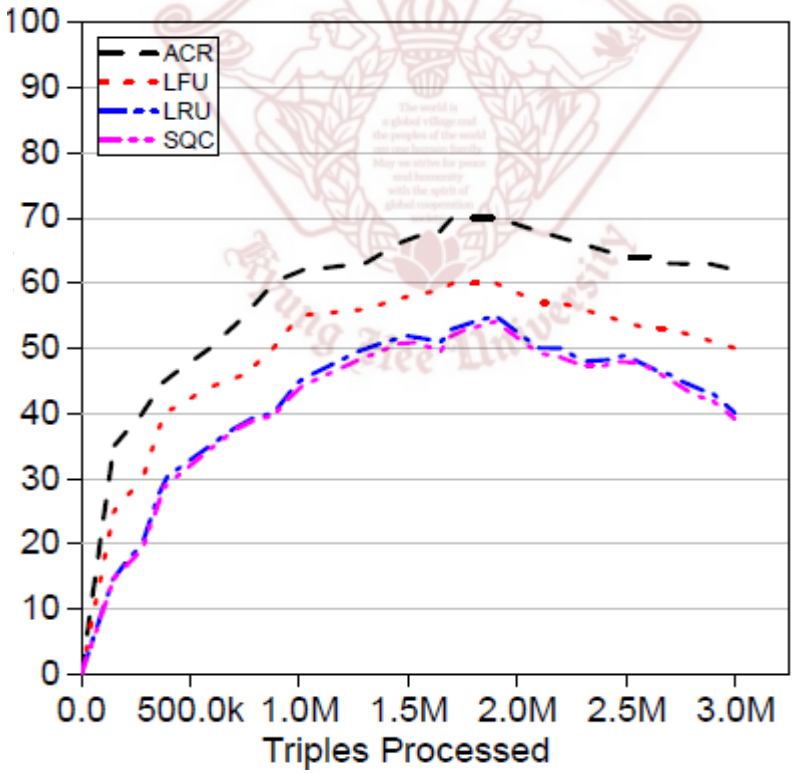


Figure 5.8: Hit Rate Achieved by Varying the Size of the Triples as Compared to Existing Approaches.

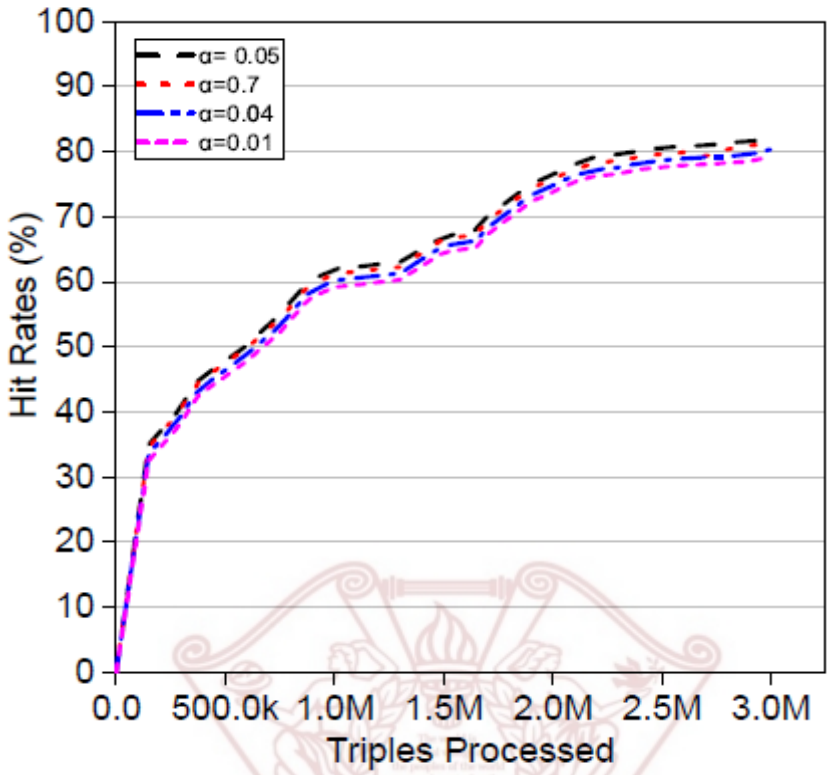


Figure 5.9: Hit Rate Achieved by Varying the Parameters of Exponential Smoothing

Figure 5.10 depicts the space overhead used by the cache replacement algorithms for varying data set sizes. We measure the maximum space consumption of each approach based on the maximum number of records that each algorithm stores. It is observed that existing approaches consume more space to maintain the records. Figure 5.10 shows the time overhead average of the proposed ACR technique compared with state-of-the-art solutions. The existing solutions take a long time; on average the hit checking time of our approach takes (280 ms), which is almost 10 times better than other approaches.

Through experimental evaluation, we found that our approach outperforms the state-of-the-art approaches in terms of better query response time and less space overhead without losing the cache hit rate. This shows that on average, we achieve hit rates of 80.66%, which accelerates the querying speed by 6.34%. Specifically, our ACR technique is capable of classifying the access log with better space efficiency as compared to LFU, LRU, and SQC.

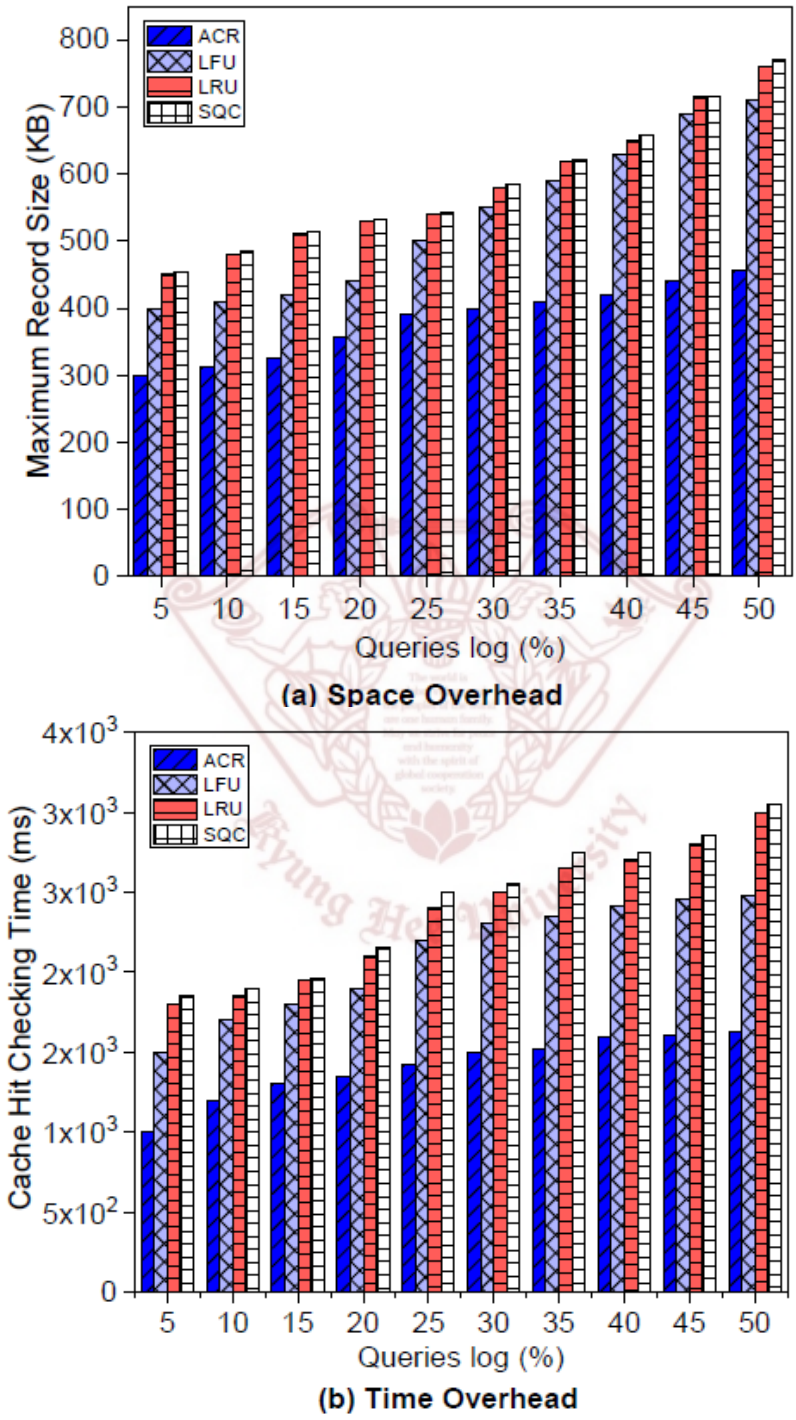


Figure 5.10: Space and Time Overhead of Existing as Compared to ACR

5.3 Detailed Evaluation: Accuracy vs Performance

5.3.1 Evaluation Setup

This section describes the evaluation of our approach and comparison with other approaches. We first describe the setup of our experimentation and dataset used. For performance evaluation we measure the response time, maintenance cost and cache hit rates. We conducted the experiments on OpenLink Virtuoso Server 07.10 with 4xAMD A8-7650K Radeon R7 and 64bit Ubuntu 16.04.2 LTS, and 32 GB of RAM with 7200 RPM. There are numerous datasets available in *Biomedicine* that originate from different sources.

In our experimentation, we utilized two datasets, LinkedCT [8] and DYLDO [9]. Both of these datasets are serialized in triple format, we use NxParser[10] to parse these datasets. For query evaluation, we have utilized the queries extracted from the logs of LinkedCT and DYLDO [167]. In our evaluation, we have utilized the queries extracted from the public available SPARQL endpoints provided by the Linked SPARQL Queries (LSQ) [11] datasets.

The *Linked Clinical Trials (LinkedCT)* dataset [7] is a Linked Data representation of the open dataset *ClinicalTrials.gov*. The original dataset is publisehd in XML format and the main benefit of its linked representation is that it facilitates SPARQL queries. This dataset contains information about governmental and privately funded clinical trials, in approximately 9.8 million triples. This dataset also contains links to external datasets such as *DBpedia* and *Bio2RDF.org* via SPARQL endpoints..

The Dynamic Linked Data Observatory ***DYLDO*** [1,44] monitors the evolution of Linked Data over time. It collects snapshots of the Web of Data using 149 weekly crawls of the Linked Data Cloud. The average size of a snapshot is about 1.35 GB. The data collected during three years adds up to approximately 36 GB. DYLDO includes various well known sources such as *DBpedia.org*, *identi.ca*, and *DBtropes.org*.

[8] https://www.w3.org/wiki/HCLSIG/LODD/Data
[9] http://swse.deri.org/dyldo/data/
[10] https://www.w3.org/2001/sw/wiki/NxParser
[11] https://aksw.github.io/LSQ/

5.3.2 Comparison with existing approaches

We have evaluated our approach against *Eager Maintenance*, *Time To Live (TTL)*, *PageRank*, *Size*, *ChangeRatio*, and *ChangeRate*.

5.3.2.1 Eager maintenance

In eager maintenance, all the updating jobs are performed immediately after the new data arrives. Eager maintenance updates all the materialized views immediately after the query evaluation and each update query has to wait until view maintenance is done. The cost of the view maintenance is normally high [169].

5.3.2.2 Time To Live (TTL)

TTL, it is a fixed threshold; the age at which the view must be refreshed. Most of the maintenance manager use *TTL* to captures how old the views from the last query evaluation are. TTL is an important feature that illustrates when the data source was last visited and updated [36].

$$P_{TTL}(X_{t,c}) = t_i - t_{last}(X_{t,c}) \tag{5.9}$$

As shown in Eq. 5.9, TTL is used by the scheduling strategies to fetch the data source c in time t_i, whereas, t_{last} is the last update time.

5.3.2.3 PageRank (PR)

In PageRank, the updates of local views is performed on the basis of ranking of the Linked Data sources. The ranking is calculated based on the number of incoming links and the sources are prefetched from highest PageRank [165]. It is represented as:

$$P_{PR}(X_{t,c}) = PR(X_{c_{highest},t_{update}}) \tag{5.10}$$

5.3.2.4 Size

In this policy, *Size* is determined by checking the Linked Data sources. The priority is given to the largest datasource [37]. The formula for *size* is as follows:

$$P_{size}(X_{t,c}) = |X_{c_{max},t_{update}}| \tag{5.11}$$

5.3.2.5 ChangeRatio

In this policy, *ChangeRatio* counts how many items were updated based on the last known time period [166]. This metric is useful for storing the change history and number of detected changes of the Linked Data as shown in the Eq. 5.12.

$$P_{ChangeRatio} = \sum_{i=no.of changes} |X_{c,t_{lastupdate}}| \tag{5.12}$$

5.3.2.6 ChangeRate

In this maintenance policy, the local views are updated from the sources with the most to the least changes [26] observed at last known points. It is represented as follows:

$$P_{ChangeRatio} = \Delta\left(X_{c,t_{lastupdate}}, X_{c,t_{lastupdate-1}} \right) \tag{5.13}$$

5.3.3 Performance Evaluation

To evaluate the effectiveness of CAMP, we compared our proposed approach with the state-of-the-art approaches to find out the quality of the maintenance performed by existing strategies. We evaluated on the basis of the maintenance cost, quality and cache hit rates.

5.3.3.1 Accuracy of Maintenance Cost

We define the maintenance cost as the time taken to perform maintenance operations. Figure 5.11 shows the maintenance cost for each policy on each dataset. Our proposed approach CAMP performed the maintenance job in the background when the system was idle and enough resources

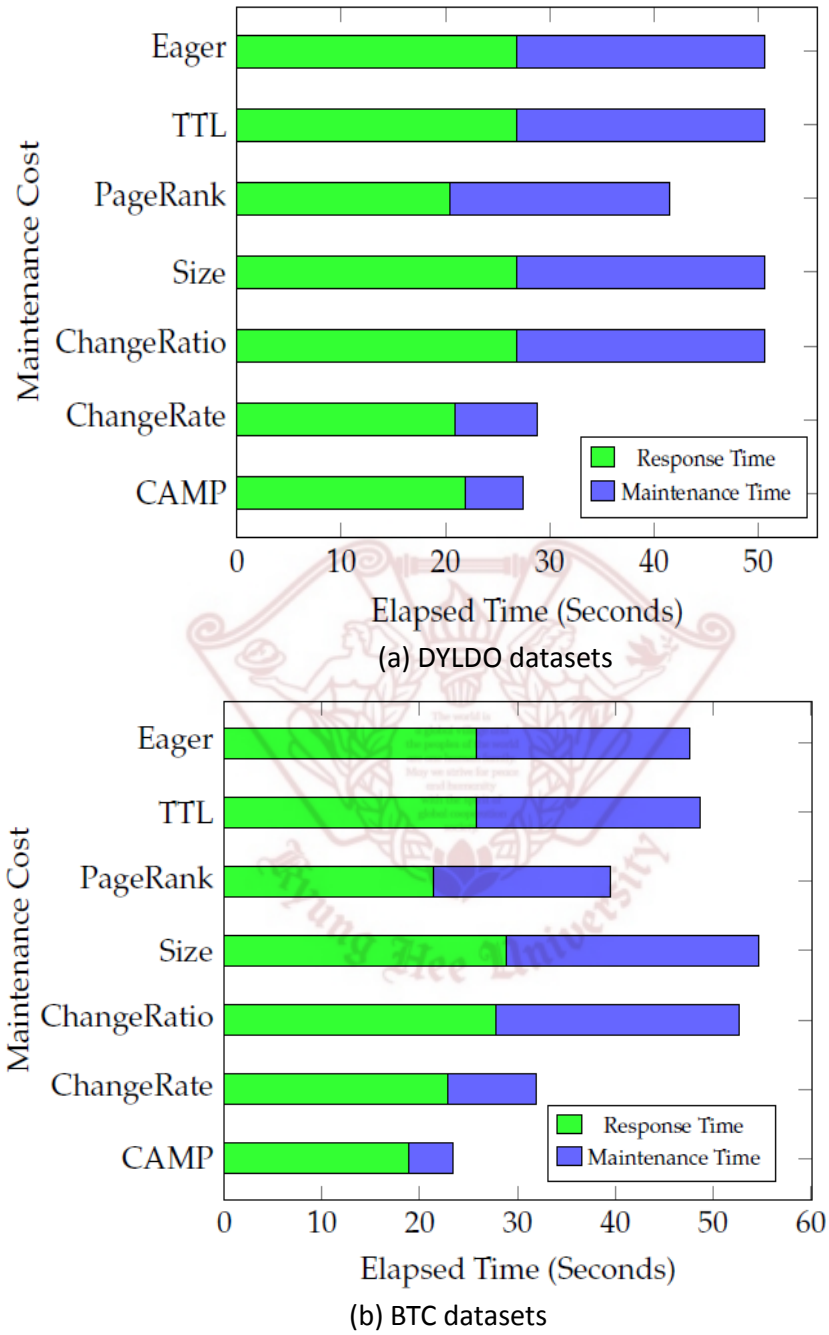


Figure 5.11: Maintenance Cost: Showing the Comparison with Other state-of-the-art Approaches on (a) LinkedCT and (b) DYLDO Datasets.

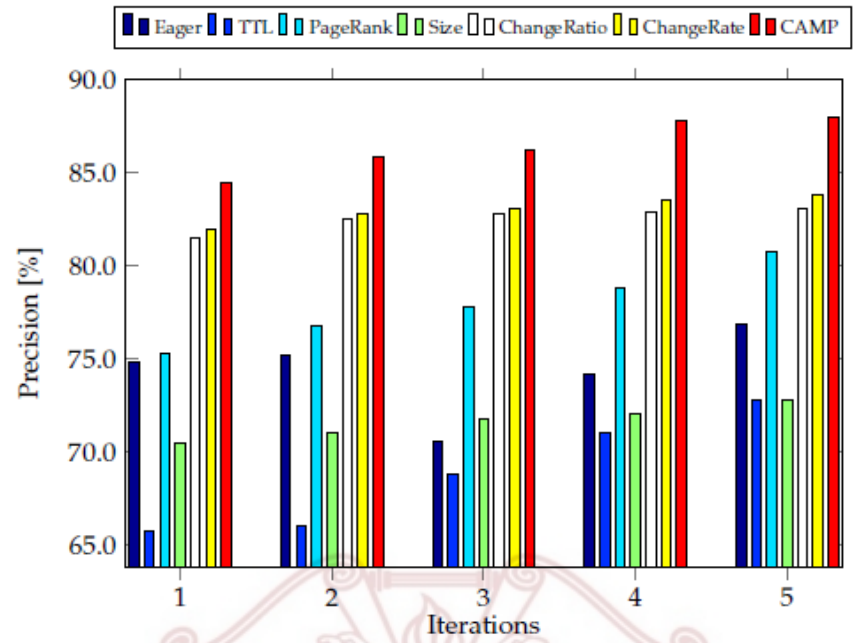
were available. Therefore, the overhead of our proposed approach was completely hidden from the user and measured the required system time in order to perform the offline maintenance task. The maintenance cost was measured as a sum of the response time and total time. Our approach produced a lower elapsed time of 5 seconds as the query does not pay for the cost of maintaining the view. Compared with *Eager Maintenance* where the query has to wait until the maintenance job is completed, the average response time was 15 seconds. However, PageRank, Size, and TTL did not consider maintenance cost while updating the local views and these strategies performed worst as query had to wait until the maintenance jobs were completed. Similarly, in the case of the ChangeRatio and ChangeRate, these policies use the periodic update function that keeps on tracking the changed occurred in the Linked Data cloud. As these policies often run in the background they produced high latency.

5.3.3.2 Accuracy of Maintenance Quality

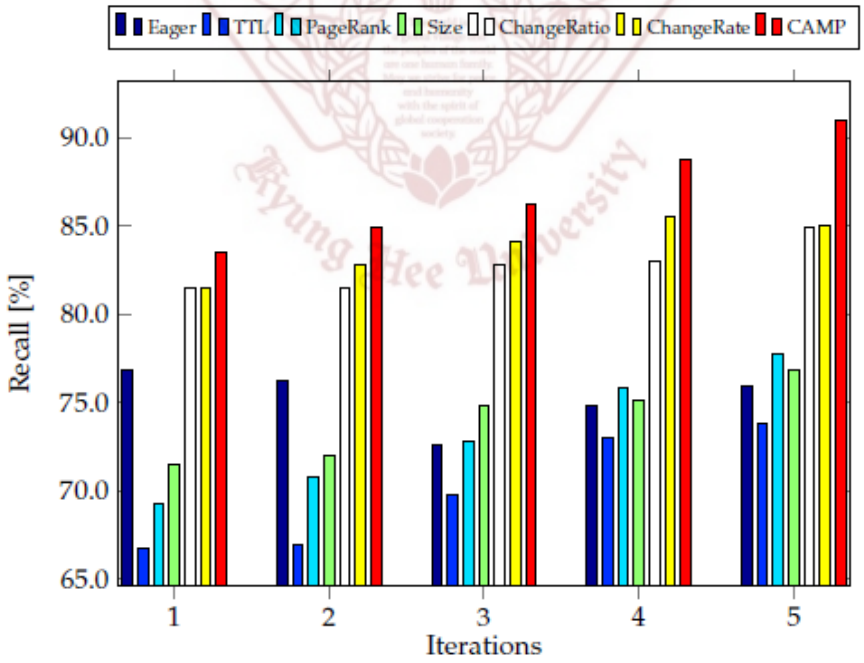
In this evaluation, we utilized the quality of the updates performed by the maintenance policies under consideration. We started with the perfect cache and assumed that a change occurs in the cloud. Due to this change the local views become outdated. Therefore, our main goal was to check the quality of the updates performed by the state-of-the-art strategies as shown in the Figure 5.12. And we evaluated how the existing strategies perform to update the local views. We used the precision and recall as an evaluation metric to measure the quality of the updates results are shown in Figure 5.12. In this setup, all the existing strategies showed a uniform loss in the quality of update. CAMP outperformed all other strategies, achieving 91% (precision) and 89% (recall) accuracy in the datasets. We observed that our proposed strategy only updates the relevant data sources with less overhead and delay. Strategies like *Pagerank*, *Size* and *TTL* performed worst because these strategies were executing irrelevant queries. ChangeRatio and ChangeRate only capture changed items and their efficiency degraded with each iteration over time.

5.3.3.3 Performance of Cache Hit Rates

As cache had a limited space; it was advantageous to replace the cache with more valuable content to improve query performance (i.e., cache hit rate). In this evaluation, we measured the perfor-

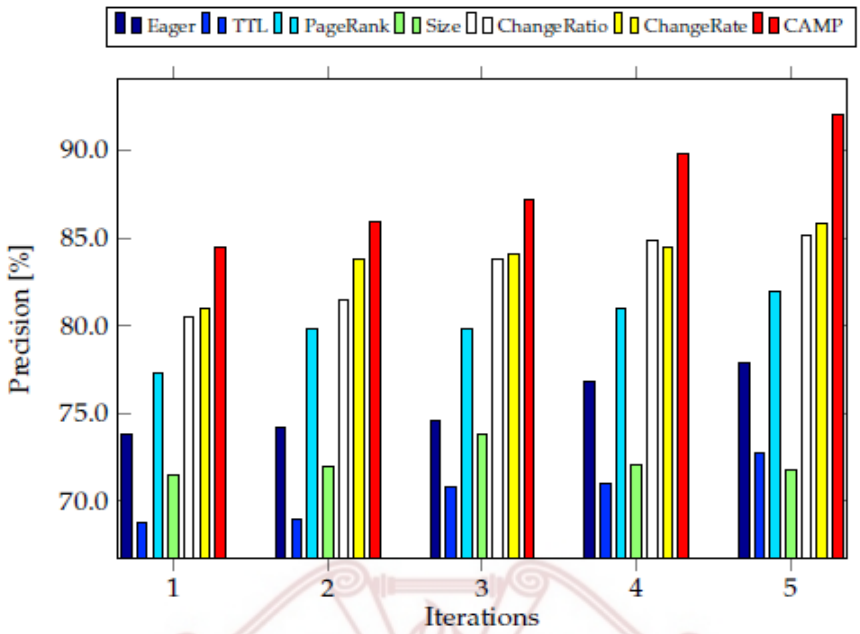


(a) Precision on LinkedCT Dataset

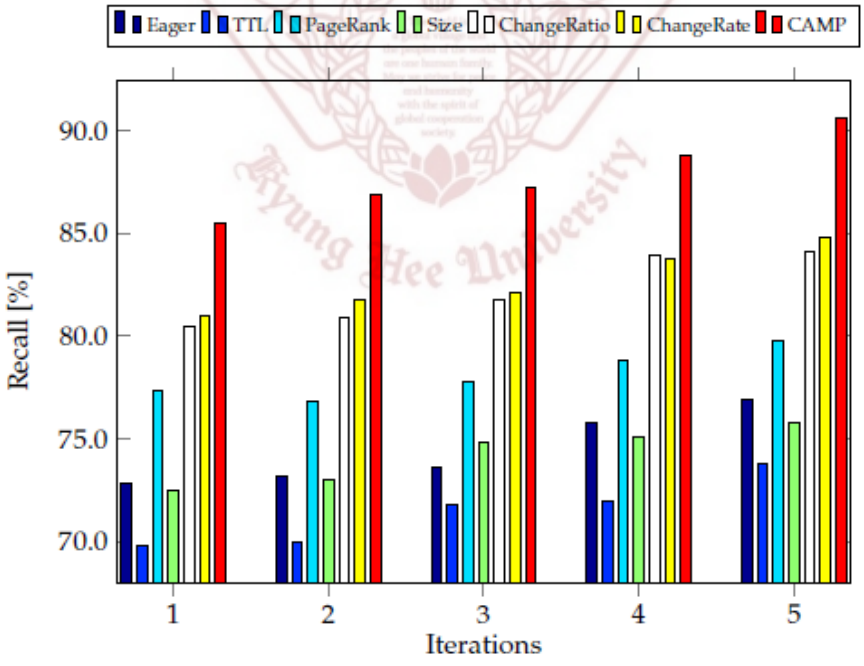


(b) Recall on LinkedCT Dataset

Figure 5.12: Quality of Updates performed by the Proposed Approach on LinkedCT Dataset.



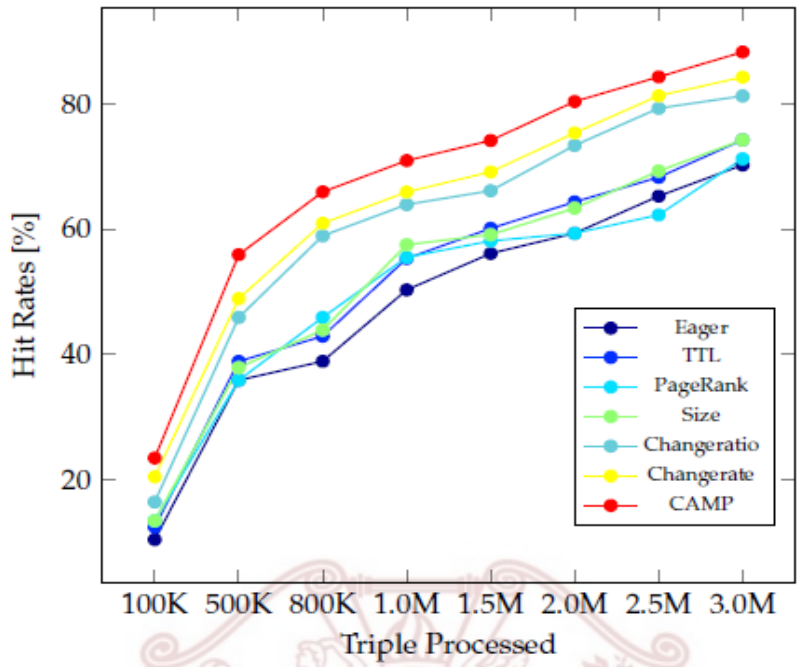
(c) Precision on DYLDO Dataset



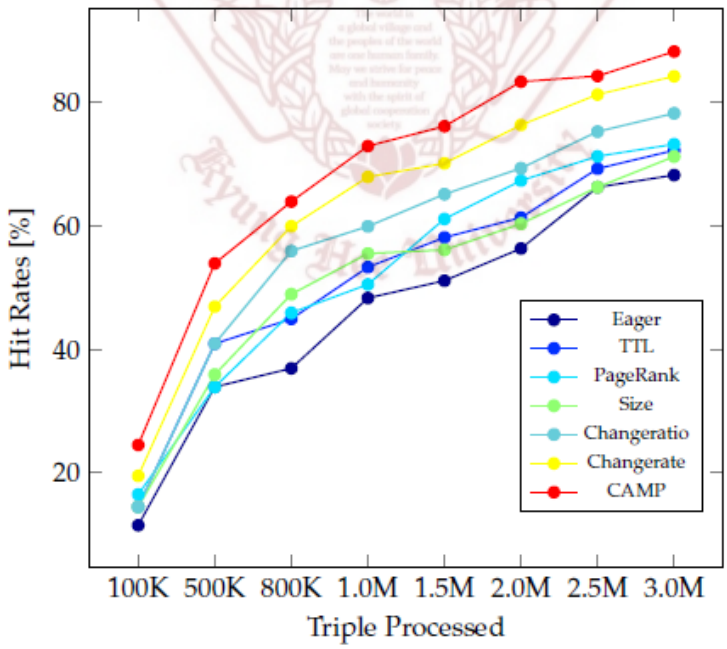
(d) Recall on DYLDO Dataset

Figure 5.13: Quality of Updates Performed by the Proposed Approach on DYLDO dataset.

mance of query times in terms of better hit rates. We evaluated on both of the datasets LinkedCT and DYLDO. Figure 5.14 shows the hit rates achieved by the existing approaches. We utilized the access logs and based on the maintenance policy, each approach replaced the cache to keep it up-to-date. CAMP replaced the cache based on the access frequency and more frequent access queries were placed in a cache for future access. On average, CAMP achieved 82% hit rates as compared to the eager (77%) and ChangeRate (70%). *PageRank*, *Eager*, *TTL* and *Size* performed worst in term of cache hit rates.



(a) Hit Rate on LinkedCT Dataset



(b) Hit Rate on DYLDO Dataset

Figure 5.14: Hit Rate Achieved by Proposed Approach as Compared to Existing Approaches.

The proposed maintenance policy utilizes a change metric together with a query similarity measure to identify and update changed items. Most of the queries issued by the Linked Data client are similar in structure. Therefore, instead of running the queries repeatedly, we prefetched the results of these queries to improve cache hit rates. For the maintenance jobs, we combined the smaller task into one job to reduce resource utilization. We compared the effectiveness of our proposed approach to state-of-the-art approaches namely eager, *TTL*, *PageRank*, *Size*, *ChangeRatio*, and *ChangeRate*. The proposed approach outperformed the existing policies in terms of less maintenance cost, higher maintenance quality, and better cache hit rates.

5.4 Summary

Quite often, Linked Data applications prefetch data and place it in a cache for future access. Due to the continuous evolution of the Linked Data Cloud, the local cache becomes outdated. In this paper, we proposed a maintenance policy that performs the local cache update jobs before query evaluation. The proposed maintenance policy utilizes a change metric together with a query similarity measure to identify and update changed items. Most of the queries issued by the Linked Data client are similar in structure. Therefore, instead of running the queries repeatedly, we prefetched the results of these queries to improve cache hit rates. For the maintenance jobs, we combined the smaller task into one job to reduce resource utilization. We compared the effectiveness of our proposed approach to state-of-the-art approaches namely eager, *TTL*, *PageRank*, *Size*, *ChangeRatio*, and *ChangeRate*. The proposed approach outperformed the existing policies in terms of less maintenance cost, higher maintenance quality, and better cache hit rates. In the evaluation section, we have performed comprehensive experiments to show the effectiveness of the proposed approach. In the first experiments, we have evaluated the effectiveness of the proposed approach in DYLDO and BTC datasets, we noticed that the proposed change metric able to identified the dynamic sources as compared to the existing approaches our approach produce less overhead and run time. Therefore, we have evaluated our approach in a single and iterative setup. Overall we achieved an accuracy of 88%(F-measure). The goal of the second experiment is to examine the hit rates and overhead comparison of the proposed approach. On average, our proposed approach achieved a higher hit rate 80.65% on varying the size of the cache our approach performs better

as compared with the other approaches. In the third experiment, we have evaluated the maintenance quality and cost of the proposed approach as compared with existing approaches, overall we achieved 91% accuracy, and strategies such as PageRank, Size, TTL performed worst while executing the irrelevant queries and uniform loss in the quality.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This work demonstrates the importance and benefits of using caching in querying the LOD cloud. Quite often, the LOD application pre-fetches data and maintains local copies in its cache. Due to the highly dynamic nature of LOD, these local data caches have become outdated. There is a need for efficient scheduling to replace the local cache. In this thesis, we have presented our methodology to capture the changes in LOD and maintain the local data cache up-to-date. Linked Data applications pre-fetch data and place it for future access. Due to the continuous evolution of the Linked Data cloud, the local cache becomes outdated. The flexibility of the proposed approach makes my work a possible building block for retrieving the data from LOD effectively.

In this thesis, we proposed a maintenance policy that performs the local cache update jobs before query evaluation. The proposed maintenance policy utilizes a change metric together with a query similarity measure to identify and update changed items. Most of the queries issued by the Linked Data client are similar in structure. Therefore, instead of running the queries repeatedly, we prefetched the results of these queries to improve cache hit rates. For the maintenance jobs, we combined the smaller task into one job to reduce resource utilization. We compared the effectiveness of our proposed approach to state-of-the-art approaches namely eager, *TTL*, *PageRank*, *Size*, *ChangeRatio*, and *ChangeRate*. The proposed approach outperformed the existing policies in terms of less maintenance cost, higher maintenance quality, and better cache hit rates. In the future, we will investigate the benefit of our proposed approach to accelerate the query time in real-world data analytic applications.

Our approach is based on the idea that the clients who processed similar queries are likely to process similar queries in the future. The aim is to search and gather content possibly requested at

the future queries by identifying the concepts of the previously issued queries. The main benefit is to reduce the transmission overhead and improve the hit rate and query time by retrieving contents for future queries at once. We prefetch all contents that were used to answer the queries since those queries are more likely to be requested by the same user in the future. For cache replacement, we serve each query according to the estimated frequency, the query with the highest frequencies are kept in the cache. When a client sends a new query the cache manager first check if an identical query has been cached. In this case, the results are immediately sent to the client. In case of the cache miss, new queries are sent to the query prefetching where similar queries are suggested and the result of these queries are retrieved from the SPARQL endpoints. As an offline process, the result of a similar query is placed in the cache for future access queries. The cache replacement process is triggered when the cache is full and it runs on a separate thread that does not affect the query answering process. The cache replacement is based on the frequency, the higher accessed queries are placed in the cache. The key contributions of this research are as follows:

1. Proposed a change metric to quantify the evolution of changes in Linked Data cloud. This approach alleviates the expensive job of copying the whole data instead of only updating local data caches with the changed items. This change metric was also utilized to find the addition and deletion of the items in the Linked Data cloud. As we also find out that queries issued by the agent are repetitive and follow similar patterns that only differ in a specific item. Therefore, it is useful to store the results of similar structure queries and place them in the cache for future access. Therefore, the structure-based similarity is are important for the application where the assessment of the query content is important. However, the structure-based content is not enough as it is less flexible due to the ordering of the symbols. On the other hand, the content-based similarity is typically useful for applications such as query suggestion and query recommendation.

2. Proposed a prefetching approach that is suitable for alleviating the burden on the SPARQL endpoint. Instead of retrieving the results of similar queries, prefetching allows gathering the relevant data potentially useful for future queries. Therefore, the query prefetching approach is suitable for a search engine to optimally extract the information from the resources. The most benefit of the query prefetching approach is the increase in the cache hit rate as most

of the queries can be served from the cache. The results of the queries need to prefetch as an offline process during system idle time when system traffic is low.

3. Proposed a frequency-based cache replacement method using data from the access log. Exponential smoothing is applied to rank each query according to its estimated frequency, the most frequently accessed queries are kept in the cache. The cache replacement is the process to keep the local data caches up-to-date. Cache replacement incorporates the previous estimation and establishes when a local data cache needs to be upgraded. However, instead of replacing, all the cache preference should be given by replacing it with the changed items.

6.2 Future Work

This research investigated a cache-based method to improve the query performance of Linked Open Data (LOD). Future application scenarios and research endeavors include:

1. **Additional Augmentation and Caching Methods:** In this thesis, we proposed query augmentation that is based on the template to retrieve data that is requested for future queries. Therefore, we will investigate the machine learning approaches to examine the query logs, and prefetching is based on the prediction where the multiple queries are issued by the same user. For a more fine-grained analysis, shorter sequences, e.g., request pairs, could be considered as well.

2. **Linked Data Evolution:** The models presented in this dissertation make a simplifying assumption that a queried web of Linked data is dynamic in nature and contents are added, updated, and removed. Therefore, the extension of our work includes the framework to deal with the evolution of the Linked Data to query over a changing Web of Linked Data.

3. **Linked Data Archiving:** The Linked data is constantly evolving there is a need for systems that support efficiently storing and querying over evolving data. We will applied our approach on Archiving systems such as x-RDF-3X [148], SemVersion [2], R43ples [141], TailR [170] and Memento [171] to improve the versioning over the archiving data.

In summary, in this thesis, we addressed multiple challenges of querying Linked Data access through Sparql queries. Whereas existing state-of-the-art approaches focus on publishing, processing, and managing LOD data, investigating and assisting user interaction with this information is crucial for establishing its acceptance among data consumers.

Bibliography

[1] T. Käfer, A. Abdelrahman, J. Umbrich, P. O'Byrne, and A. Hogan, "Observing linked data dynamics," in *Extended Semantic Web Conference*. Springer, 2013, pp. 213–227.

[2] M. Völkel and T. Groza, "Semversion: An rdf-based ontology versioning system," in *Proceedings of the IADIS international conference WWW/Internet*, vol. 2006, 2006, p. 44.

[3] D. Zeginis, Y. Tzitzikas, and V. Christophides, "On computing deltas of rdf/s knowledge bases," *ACM Transactions on the Web (TWEB)*, vol. 5, no. 3, p. 14, 2011.

[4] T. Berners-Lee, J. Hendler, and O. Lassila, "The semantic web," *Scientific american*, vol. 284, no. 5, pp. 34–43, 2001.

[5] J. Lehmann and L. Bühmann, "Autosparql: Let users query your knowledge base," in *Extended Semantic Web Conference*. Springer, 2011, pp. 63–79.

[6] S. Auer, J. Lehmann, and S. Hellmann, "Linkedgeodata: Adding a spatial dimension to the web of data," in *International Semantic Web Conference*. Springer, 2009, pp. 731–746.

[7] O. Hassanzadeh and R. J. Miller, "Automatic curation of clinical trials data in linkedct," in *International Semantic Web Conference*. Springer, 2015, pp. 270–278.

[8] I. Keivanloo, C. Forbes, A. Hmood, M. Erfani, C. Neal, G. Peristerakis, and J. Rilling, "A linked data platform for mining software repositories," in *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*. IEEE, 2012, pp. 32–35.

[9] J. Umbrich, M. Karnstedt, A. Hogan, and J. X. Parreira, "Hybrid sparql queries: fresh vs. fast results," in *International Semantic Web Conference*. Springer, 2012, pp. 608–624.

[10] J. Lorey and F. Naumann, "Caching and prefetching strategies for sparql queries," in *Extended Semantic Web Conference*. Springer, 2013, pp. 46–65.

[11] M. Martin, J. Unbehauen, and S. Auer, "Improving the performance of semantic web applications with sparql query caching," in *Extended Semantic Web Conference*. Springer, 2010, pp. 304–318.

[12] U. Akhtar, A. Sant'Anna, C.-H. Jihn, M. A. Razzaq, J. Bang, and S. Lee, "A cache-based method to improve query performance of linked open data cloud," *Computing*, pp. 1–21, 2020.

[13] U. Akhtar, A. Sant'Anna, and S. Lee, "A dynamic, cost-aware, optimized maintenance policy for interactive exploration of linked data," *Applied Sciences*, vol. 9, no. 22, p. 4818, 2019.

[14] M. K. Saggi and S. Jain, "A survey towards an integration of big data analytics to big insights for value-creation," *Information Processing & Management*, vol. 54, no. 5, pp. 758–790, 2018.

[15] S. Khan, X. Liu, K. A. Shakil, and M. Alam, "A survey on scholarly data: From big data perspective," *Information Processing & Management*, vol. 53, no. 4, pp. 923–944, 2017.

[16] J. D. Fernández, J. Umbrich, A. Polleres, and M. Knuth, "Evaluating query and storage strategies for rdf archives," in *Proceedings of the 12th International Conference on Semantic Systems*. ACM, 2016, pp. 41–48.

[17] T. Gottron, M. Knauf, and A. Scherp, "Analysis of schema structures in the linked open data graph based on unique subject uris, pay-level domains, and vocabulary usage," *Distributed and Parallel Databases*, vol. 33, no. 4, pp. 515–553, 2015.

[18] J. Cho and H. Garcia-Molina, "Estimating frequency of change," *ACM Transactions on Internet Technology (TOIT)*, vol. 3, no. 3, pp. 256–290, 2003.

[19] U. Akhtar, M. B. Amin, and S. Lee, "Evaluating scheduling strategies in lod based application," in *Network Operations and Management Symposium (APNOMS), 2017 19th Asia-Pacific*. IEEE, 2017, pp. 255–258.

[20] C. Bizer, T. Heath, and T. Berners-Lee, "Linked data-the story so far," *International journal on semantic web and information systems*, vol. 5, no. 3, pp. 1–22, 2009.

[21] M. Konrath, T. Gottron, S. Staab, and A. Scherp, "Schemex—efficient construction of a data catalogue by stream-based indexing of linked data," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 16, pp. 52–58, 2012.

[22] T. Gottron, "Measuring the accuracy of linked data indices," *arXiv preprint arXiv:1603.06068*, 2016.

[23] C. Nishioka and A. Scherp, "Keeping linked open data caches up-to-date by predicting the life-time of rdf triples," in *Proceedings of the International Conference on Web Intelligence*. ACM, 2017, pp. 73–80.

[24] R. Dividino, T. Gottron, and A. Scherp, "Strategies for efficiently keeping local linked open data caches up-to-date," in *International Semantic Web Conference*. Springer, 2015, pp. 356–373.

[25] R. Dividino, A. Scherp, G. Gröner, and T. Grotton, "Change-a-lod: does the schema on the linked data cloud change or not?" in *Proceedings of the Fourth International Conference on Consuming Linked Data-Volume 1034*. CEUR-WS. org, 2013, pp. 87–98.

[26] M. Knuth, O. Hartig, and H. Sack, "Scheduling refresh queries for keeping results from a sparql endpoint up-to-date (extended version)," *arXiv preprint arXiv:1608.08130*, 2016.

[27] C. Nishioka and A. Scherp, "Temporal patterns and periodicity of entity dynamics in the linked open data cloud," in *Proceedings of the 8th International Conference on Knowledge Capture*. ACM, 2015, p. 22.

[28] E. Fotopoulou, A. Zafeiropoulos, D. Papaspyros, P. Hasapis, G. Tsiolis, T. Bouras, S. Mouzakitis, and N. Zanetti, "Linked data analytics in interdisciplinary studies: The health impact of air pollution in urban areas," *IEEE Access*, vol. 4, pp. 149–164, 2015.

[29] T. Gottron and C. Gottron, "Perplexity of index models over evolving linked data," in *European Semantic Web Conference*. Springer, 2014, pp. 161–175.

[30] S. Podlipnig and L. Böszörmenyi, "A survey of web cache replacement strategies," *ACM Computing Surveys (CSUR)*, vol. 35, no. 4, pp. 374–398, 2003.

[31] P. J. Denning, "The working set model for program behavior," *Communications of the ACM*, vol. 11, no. 5, pp. 323–333, 1968.

[32] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies," *IEEE transactions on Computers*, vol. 50, no. 12, pp. 1352–1361, 2001.

[33] J. J. Levandoski, P.-Å. Larson, and R. Stoica, "Identifying hot and cold data in main-memory databases," in *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*. IEEE, 2013, pp. 26–37.

[34] A. Rula, M. Palmonari, A. Harth, S. Stadtmüller, and A. Maurino, "On the diversity and availability of temporal information in linked open data," *The Semantic Web–ISWC 2012*, pp. 492–507, 2012.

[35] R. Dividino, A. Kramer, and T. Gottron, "An investigation of http header information for detecting changes of linked open data sources," in *European Semantic Web Conference*. Springer, 2014, pp. 199–203.

[36] J. Cho and H. Garcia-Molina, "Synchronizing a database to improve freshness," in *ACM Sigmod Record*, vol. 29, no. 2. ACM, 2000, pp. 117–128.

[37] M. Harchol-Balter, B. Schroeder, N. Bansal, and M. Agrawal, "Size-based scheduling to improve web performance," *ACM Transactions on Computer Systems (TOCS)*, vol. 21, no. 2, pp. 207–233, 2003.

[38] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang, "A distributed graph engine for web scale rdf data," in *Proceedings of the VLDB Endowment*, vol. 6, no. 4. VLDB Endowment, 2013, pp. 265–276.

[39] U. Akhtar, M. A. Razzaq, U. U. Rehman, M. B. Amin, W. A. Khan, E.-N. Huh, and S. Lee, "Change-aware scheduling for effectively updating linked open data caches," *IEEE Access*, vol. 6, pp. 65 862–65 873, 2018.

[40] N. Papailiou, D. Tsoumakos, P. Karras, and N. Koziris, "Graph-aware, workload-adaptive sparql query caching," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 1777–1792.

[41] N. Megiddo and D. S. Modha, "Arc: A self-tuning, low overhead replacement cache." in *FAST*, vol. 3, no. 2003, 2003, pp. 115–130.

[42] E. J. O'neil, P. E. O'neil, and G. Weikum, "The lru-k page replacement algorithm for database disk buffering," *ACM SIGMOD Record*, vol. 22, no. 2, pp. 297–306, 1993.

[43] A. Passant and P. N. Mendes, "sparqlpush: Proactive notification of data updates in rdf stores using pubsubhubbub." in *SFSW*, 2010.

[44] T. Käfer, J. Umbrich, A. Hogan, and A. Polleres, "Towards a dynamic linked data observatory," *LDOW at WWW*, 2012.

[45] S. Auer, J. Demter, M. Martin, and J. Lehmann, "Lodstats–an extensible framework for high-performance dataset analytics," in *International Conference on Knowledge Engineering and Knowledge Management*. Springer, 2012, pp. 353–362.

[46] L. Po, N. Bikakis, F. Desimoni, and G. Papastefanatos, "Linked data visualization: Techniques, tools, and big data," *Synthesis Lectures on Semantic Web: Theory and Technology*, vol. 10, no. 1, pp. 1–157, 2020.

[47] V. Mallawaarachchi, L. Meegahapola, R. Madhushanka, E. Heshan, D. Meedeniya, and S. Jayarathna, "Change detection and notification of web pages: A survey," *ACM Computing Surveys (CSUR)*, vol. 53, no. 1, pp. 1–35, 2020.

[48] K. Kjernsmo, "A survey of http caching implementations on the open semantic web," in *European Semantic Web Conference*. Springer, 2015, pp. 286–301.

[49] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey, "Algorithms for deferred view maintenance," in *ACM SIGMOD Record*, vol. 25. ACM, 1996, pp. 469–480.

[50] J. Zhou, P.-A. Larson, and H. G. Elmongui, "Lazy maintenance of materialized views," in *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 2007, pp. 231–242.

[51] S. Dar, M. J. Franklin, B. T. Jonsson, D. Srivastava, M. Tan *et al.*, "Semantic data caching and replacement," in *VLDB*, vol. 96, 1996, pp. 330–341.

[52] P. Godfrey and J. Gryz, "Answering queries by semantic caches," in *International Conference on Database and Expert Systems Applications*. Springer, 1999, pp. 485–498.

[53] Y. Shu, M. Compton, H. Müller, and K. Taylor, "Towards content-aware sparql query caching for semantic web applications," in *International Conference on Web Information Systems Engineering*. Springer, 2013, pp. 320–329.

[54] M. Yang and G. Wu, "Caching intermediate result of sparql queries," in *Proceedings of the 20th international conference companion on World wide web*. ACM, 2011, pp. 159–160.

[55] C. Buil-Aranda, A. Hogan, J. Umbrich, and P.-Y. Vandenbussche, "Sparql web-querying infrastructure: Ready for action?" in *International Semantic Web Conference*. Springer, 2013, pp. 277–293.

[56] J. Lorey, "Identifying and determining sparql endpoint characteristics," *International Journal of Web Information Systems*, 2014.

[57] L. Feigenbaum, G. T. Williams, K. G. Clark, and E. Torres, "sparql 1.1 protocol. recommendation," *World Wide Web Consortium, March*, 2013.

[58] C. Bizer and A. Schultz, "The berlin sparql benchmark," *International Journal on Semantic Web and Information Systems (IJSWIS)*, vol. 5, no. 2, pp. 1–24, 2009.

[59] P. Kulkarni, "Distributed sparql query engine using mapreduce," *Master of Science, Computer Science, School of Informatics, University of Edinburgh*, 2010.

[60] A. Schätzle, M. Przyjaciel-Zablocki, and G. Lausen, "Pigsparql: Mapping sparql to pig latin," in *Proceedings of the International Workshop on Semantic Web Information Management*, 2011, pp. 1–8.

[61] J. Leeka and S. Bedathur, "Rq-rdf-3x: going beyond triplestores," in *2014 IEEE 30th International Conference on Data Engineering Workshops*. IEEE, 2014, pp. 263–268.

[62] B. Quilitz and U. Leser, "Querying distributed rdf data sources with sparql," in *European semantic web conference*. Springer, 2008, pp. 524–538.

[63] A. Hasnain, Q. Mehmood, S. S. e Zainab, M. Saleem, C. Warren, D. Zehra, S. Decker, and D. Rebholz-Schuhmann, "Biofed: federated query processing over life sciences linked open data," *Journal of biomedical semantics*, vol. 8, no. 1, p. 13, 2017.

[64] B. Spahiu, A. Maurino, and R. Meusel, "Topic profiling benchmarks in the linked open data cloud: Issues and lessons learned," *Semantic Web*, vol. 10, no. 2, pp. 329–348, 2019.

[65] M. R. Kamdar, J. D. Fernández, A. Polleres, T. Tudorache, and M. A. Musen, "Enabling web-scale data integration in biomedicine through linked open data," *NPJ digital medicine*, vol. 2, no. 1, pp. 1–14, 2019.

[66] G. Nassopoulos, P. Serrano-Alvarado, P. Molli, and E. Desmontils, "Extracting basic graph patterns from triple pattern fragment logs," 2017.

[67] E. T. Zacharatou, "Efficient query processing for spatial and temporal data exploration," Ph.D. dissertation, Ecole Polytechnique Fédérale de Lausanne, 2019.

[68] R. Al-Harbi, "Accelerating sparql queries and analytics on rdf data," Ph.D. dissertation, 2016.

[69] L. Yan, R. Ma, D. Li, and J. Cheng, "Rdf approximate queries based on semantic similarity," *Computing*, vol. 99, no. 5, pp. 481–491, 2017.

[70] B. B. Cambazoglu, F. P. Junqueira, V. Plachouras, S. Banachowski, B. Cui, S. Lim, and B. Bridge, "A refreshing perspective of search engine caching," in *Proceedings of the 19th International Conference on World Wide Web*, ser. WWW '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 181–190. [Online]. Available: https://doi.org/10.1145/1772690.1772710

[71] O. Hartig, "How caching improves efficiency and result completeness for querying linked data," 2011.

[72] R. Q. Dividino and G. Gröner, "Which of the following sparql queries are similar? why?" in *LD4IE@ ISWC*, 2013.

[73] T. J. Berners-Lee, "Information management: A proposal," Tech. Rep., 1989.

[74] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor, "Freebase: a collaboratively created graph database for structuring human knowledge," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. AcM, 2008, pp. 1247–1250.

[75] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. Van Kleef, S. Auer *et al.*, "Dbpedia–a large-scale, multilingual knowledge base extracted from wikipedia," *Semantic Web*, vol. 6, no. 2, pp. 167–195, 2015.

[76] F. M. Suchanek, G. Kasneci, and G. Weikum, "Yago: a core of semantic knowledge," in *Proceedings of the 16th international conference on World Wide Web*. ACM, 2007, pp. 697–706.

[77] A. O. Bahanshal and H. S. Al-Khalifa, "Evolution of linked data application domains from 2009 to 2015: A systematic literature review," *International Journal of Technology Diffusion (IJTD)*, vol. 9, no. 2, pp. 1–20, 2018.

[78] J. Marden, C. Li-Madeo, N. Whysel, and J. Edelstein, "Linked open data for cultural heritage: evolution of an information technology," in *Proceedings of the 31st ACM international conference on Design of communication*, 2013, pp. 107–112.

[79] G. Tsatsaronis, M. Schroeder, G. Paliouras, Y. Almirantis, I. Androutsopoulos, E. Gaussier, P. Gallinari, T. Artieres, M. R. Alvers, M. Zschunke *et al.*, "Bioasq: A challenge on large-scale biomedical semantic indexing and question answering." in *AAAI fall symposium: Information retrieval and knowledge discovery in biomedical text*, 2012.

[80] T. Hamon, N. Grabar, and F. Mougin, "Querying biomedical linked data with natural language questions," *Semantic Web*, vol. 8, no. 4, pp. 581–599, 2017.

[81] K. Höffner and J. Lehmann, "Towards question answering on statistical linked data," in *Proceedings of the 10th International Conference on Semantic Systems*. ACM, 2014, pp. 61–64.

[82] K. Hoffner, J. Lehmann, and R. Usbeck, "CubeQA-question answering on RDF data cubes," in *International Semantic Web Conference*. Springer, 2016, pp. 325–340.

[83] M. Röder, D. Kuchelev, and A.-C. Ngonga Ngomo, "Hobbit: A platform for benchmarking big linked data," *Data Science*, no. Preprint, pp. 1–21, 2019.

[84] S. Auer, L. Bühmann, C. Dirschl, O. Erling, M. Hausenblas, R. Isele, J. Lehmann, M. Martin, P. N. Mendes, B. Van Nuffelen *et al.*, "Managing the life-cycle of linked data with the lod2 stack," in *International semantic Web conference*. Springer, 2012, pp. 1–16.

[85] M. Atzeni and M. Atzori, "Codeontology: Rdf-ization of source code," in *International Semantic Web Conference*. Springer, 2017, pp. 20–28.

[86] I. Dongo and R. Chbeir, "S-rdf: A new rdf serialization format for better storage without losing human readability," in *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer, 2019, pp. 246–264.

[87] A. Sanfeliu and K.-S. Fu, "A distance measure between attributed relational graphs for pattern recognition," *IEEE transactions on systems, man, and cybernetics*, vol. 3, pp. 353–362, 1983.

[88] A. Jounaidi and M. Bahaj, "Designing and implementing xml schema inside owl ontology," in *2017 International Conference on Wireless Networks and Mobile Communications (WINCOM)*. IEEE, 2017, pp. 1–7.

[89] M. M. Mkhinini, O. Labbani-Narsis, and C. Nicolle, "Combining uml and ontology: An exploratory survey," *Computer Science Review*, vol. 35, p. 100223, 2020.

[90] A. Basu, "Semantic web, ontology, and linked data," in *Web services: concepts, methodologies, tools, and applications*. IGI Global, 2019, pp. 127–148.

[91] C. Bizer, T. Heath, and T. Berners-Lee, "Linked data: The story so far," in *Semantic services, interoperability and web applications: emerging concepts*. IGI Global, 2011, pp. 205–227.

[92] F. Belleau, M.-A. Nolin, N. Tourigny, P. Rigault, and J. Morissette, "Bio2rdf: towards a mashup to build bioinformatics knowledge systems," *Journal of biomedical informatics*, vol. 41, no. 5, pp. 706–716, 2008.

[93] N. P. Popitsch and B. Haslhofer, "Dsnotify: handling broken links in the web of data," in *Proceedings of the 19th international conference on World wide web*. ACM, 2010, pp. 761–770.

[94] E. Rajabi, S. Sanchez-Alonso, and M.-A. Sicilia, "Analyzing broken links on the web of data: An experiment with dbpedia," *Journal of the Association for Information Science and Technology*, vol. 65, no. 8, pp. 1721–1727, 2014.

[95] J. Umbrich, S. Decker, M. Hausenblas, A. Polleres, and A. Hogan, "Towards dataset dynamics: Change frequency of linked open data sources," 2010.

[96] R. Q. Dividino, T. Gottron, A. Scherp, and G. Gröner, "From changes to dynamics: Dynamics analysis of linked open data sources." in *PROFILES@ ESWC*, 2014.

[97] A. Third and J. Domingue, "Linked data indexing of distributed ledgers," in *Proceedings of the 26th International Conference on World Wide Web Companion*, 2017, pp. 1431–1436.

[98] A. Freitas, E. Curry, J. G. Oliveira, and S. O'Riain, "Querying heterogeneous datasets on the linked data web: challenges, approaches, and trends," *IEEE Internet Computing*, vol. 16, no. 1, pp. 24–33, 2011.

[99] M. Svoboda and I. Mlỳnková, "Linked data indexing methods: A survey," in *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer, 2011, pp. 474–483.

[100] T. White, *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[101] A. Chugh, V. K. Sharma, and C. Jain, "Big data and query optimization techniques," in *Advances in Computing and Intelligent Systems*. Springer, 2020, pp. 337–345.

[102] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad, "Hadoop++: making a yellow elephant run like a cheetah (without it even noticing)," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 515–529, 2010.

[103] S. Idreos, M. L. Kersten, S. Manegold *et al.*, "Database cracking." in *CIDR*, vol. 7, 2007, pp. 68–78.

[104] J. Dittrich, J.-A. Quiané-Ruiz, S. Richter, S. Schuh, A. Jindal, and J. Schad, "Only aggressive elephants are fast elephants," *Proceedings of the VLDB Endowment*, vol. 5, no. 11, pp. 1591–1602, 2012.

[105] Y. Zhuang, N. Jiang, Z. Wu, Q. Li, D. K. Chiu, and H. Hu, "Efficient and robust large medical image retrieval in mobile cloud computing environment," *Information Sciences*, vol. 263, pp. 60–86, 2014.

[106] A. Gani, A. Siddiqa, S. Shamshirband, and F. Hanum, "A survey on indexing techniques for big data: taxonomy and performance evaluation," *Knowledge and Information Systems*, vol. 46, no. 2, pp. 241–284, 2016.

[107] M. McCandless, E. Hatcher, and O. Gospodnetic, *Lucene in Action: Covers Apache Lucene 3.0*. Manning Publications Co., 2010.

[108] S. Bechhofer, I. Buchan, D. De Roure, P. Missier, J. Ainsworth, J. Bhagat, P. Couch, D. Cruickshank, M. Delderfield, I. Dunlop *et al.*, "Why linked data is not enough for scientists," *Future Generation Computer Systems*, vol. 29, no. 2, pp. 599–611, 2013.

[109] O. Hartig, K. Hose, and J. F. Sequeda, "Linked data management." 2019.

[110] L. Roffia, P. Azzoni, C. Aguzzi, F. Viola, F. Antoniazzi, and T. Salmon Cinotti, "Dynamic linked data: A sparql event processing architecture," *Future Internet*, vol. 10, no. 4, p. 36, 2018.

[111] N. Pernelle, F. Saïs, D. Mercier, and S. Thuraisamy, "Rdf data evolution: efficient detection and semantic representation of changes," *Semantic Systems-SEMANTiCS2016*, 2016.

[112] A. Singh, "Regions in a linked dataset for change detection," *arXiv preprint arXiv:1905.07663*, 2019.

[113] A. Singh, R. Brennan, and D. O'Sullivan, "Delta-ld: A change detection approach for linked datasets," in *4th Workshop on Managing the Evolution and Preservation of the Data Web (MEPDaW)*, 2018.

[114] K. M. Endris, S. Faisal, F. Orlandi, S. Auer, and S. Scerri, "Interest-based rdf update propagation," in *International Semantic Web Conference*. Springer, 2015, pp. 513–529.

[115] M. Bienvenu, D. Deutch, and F. M. Suchanek, "Provenance for web 2.0 data," in *Workshop on Secure Data Management*. Springer, 2012, pp. 148–155.

[116] A. M. Khattak, K. Latif, and S. Lee, "Change management in evolving web ontologies," *Knowledge-Based Systems*, vol. 37, pp. 1–18, 2013.

[117] A. Assaf, R. Troncy, and A. Senart, "What's up lod cloud?" in *European Semantic Web Conference*. Springer, 2015, pp. 247–254.

[118] L. Rietveld, W. Beek, R. Hoekstra, and S. Schlobach, "Meta-data for a lot of lod," *Semantic Web*, vol. 8, no. 6, pp. 1067–1080, 2017.

[119] R. Cyganiak and M. Hausenblas, "Describing linked datasets-on the design and usage of," *Technology*, vol. 24, no. 1, pp. 9–11, 1997.

[120] L. Ding and T. Finin, "Characterizing the semantic web on the web," in *International Semantic Web Conference*, vol. 4273. Springer, 2006, pp. 242–257.

[121] A. Hogan, J. Umbrich, A. Harth, R. Cyganiak, A. Polleres, and S. Decker, "An empirical survey of linked data conformance," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 14, pp. 14–44, 2012.

[122] S. Tramp, P. Frischmuth, T. Ermilov, and S. Auer, "Weaving a social data web with semantic pingback," in *International Conference on Knowledge Engineering and Knowledge Management*. Springer, 2010, pp. 135–149.

[123] V. Papavasileiou, G. Flouris, I. Fundulaki, D. Kotzinos, and V. Christophides, "High-level change detection in rdf (s) kbs," *ACM Transactions on Database Systems (TODS)*, vol. 38, no. 1, pp. 1–42, 2013.

[124] W. Xing and A. Ghorbani, "Weighted pagerank algorithm," in *Proceedings. Second Annual Conference on Communication Networks and Services Research, 2004.* IEEE, 2004, pp. 305–314.

[125] T. Bray, "Measuring the web," *Computer networks and ISDN systems*, vol. 28, no. 7-11, pp. 993–1005, 1996.

[126] B. E. Brewington and G. Cybenko, "How dynamic is the web?" *Computer Networks*, vol. 33, no. 1-6, pp. 257–276, 2000.

[127] B. Brewington and G. Cybenko, "Keeping up with the changing web," *Computer*, vol. 33, no. 5, pp. 52–58, 2000.

[128] J. Cho and H. Garcia-Molina, "Estimating frequency of change," *ACM Transactions on Internet Technology (TOIT)*, vol. 3, no. 3, pp. 256–290, 2003.

[129] M. Knuth, O. Hartig, and H. Sack, *Scheduling Refresh Queries for Keeping Results from a SPARQL Endpoint Up-to-Date (Short Paper)*. Cham: Springer International Publishing, 2016, pp. 780–791. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-48472-3_49

[130] R. Aldeco-Pérez and L. Moreau, "A provenance-based compliance framework," in *Future Internet Symposium*. Springer, 2010, pp. 128–137.

[131] J. D. Fernández, A. Polleres, and J. Umbrich, "Towards efficient archiving of dynamic linked open data." *DIACRON@ ESWC*, vol. 1377, pp. 34–49, 2015.

[132] W. W. W. Consortium *et al.*, "Rdf 1.1 concepts and abstract syntax," 2014.

[133] T. Neumann and G. Weikum, "Rdf-3x: a risc-style engine for rdf," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 647–659, 2008.

[134] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: sextuple indexing for semantic web data management," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 1008–1019, 2008.

[135] P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu, "Triplebit: a fast and compact system for large scale rdf data," *Proceedings of the VLDB Endowment*, vol. 6, no. 7, pp. 517–528, 2013.

[136] S. Álvarez-García, N. R. Brisaboa, J. D. Fernández, and M. A. Martínez-Prieto, "Compressed k2-triples for full-in-memory rdf engines," *arXiv preprint arXiv:1105.4004*, 2011.

[137] N. R. Brisaboa, A. Cerdeira-Pena, A. Farina, and G. Navarro, "A compact rdf store using suffix arrays," in *International Symposium on String Processing and Information Retrieval*. Springer, 2015, pp. 103–115.

[138] J. D. Fernández, M. A. Martínez-Prieto, C. Gutiérrez, A. Polleres, and M. Arias, "Binary rdf representation for publication and exchange (hdt)," *Journal of Web Semantics*, vol. 19, pp. 22–41, 2013.

[139] S. Cassidy, J. Ballantine *et al.*, "Version control for rdf triple stores." *ICSOFT (ISDM/EHST/DC)*, vol. 7, pp. 5–12, 2007.

[140] M. Vander Sande, P. Colpaert, R. Verborgh, S. Coppens, E. Mannens, and R. Van de Walle, "R&wbase: git for triples." *LDOW*, vol. 996, 2013.

[141] M. Graube, S. Hensel, and L. Urbas, "R43ples: Revisions for triples," in *Proceedings of the 1st Workshop on Linked Data Quality co-located with 10th International Conference on Semantic Systems (SEMANTiCS 2014)*. Citeseer, 2014.

[142] S. Gao, J. Gu, and C. Zaniolo, "Rdf-tx: A fast, user-friendly system for querying the history of rdf knowledge bases." in *EDBT*, 2016, pp. 269–280.

[143] A. Cerdeira-Pena, A. Farina, J. D. Fernández, and M. A. Martínez-Prieto, "Self-indexing rdf archives," in *2016 Data Compression Conference (DCC)*. IEEE, 2016, pp. 526–535.

[144] D.-H. Im, S.-W. Lee, and H.-J. Kim, "A version management framework for rdf triple stores," *International Journal of Software Engineering and Knowledge Engineering*, vol. 22, no. 01, pp. 85–106, 2012.

[145] C. Hauptmann, M. Brocco, and W. Wörndl, "Scalable semantic version control for linked data management." in *LDQ@ ESWC*, 2015.

[146] J. Broekstra, A. Kampman, and F. Van Harmelen, "Sesame: A generic architecture for storing and querying rdf and rdf schema," in *International semantic web conference*. Springer, 2002, pp. 54–68.

[147] B. Thompson, M. Personick, and M. Cutcher, "The bigdata® rdf graph database," in *Linked Data Management*. Chapman and Hall/CRC, 2016, pp. 221–266.

[148] T. Neumann and G. Weikum, "x-rdf-3x: fast querying, high update rates, and consistency for rdf databases," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 256–263, 2010.

[149] J. Anderson and A. Bendiken, "Transaction-time queries in dydra." *MEPDaW/LDQ@ ESWC*, vol. 1585, pp. 11–19, 2016.

[150] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel, "Spˆ 2bench: a sparql performance benchmark," in *2009 IEEE 25th International Conference on Data Engineering*. IEEE, 2009, pp. 222–233.

[151] H. Stuckenschmidt, "Similarity-based query caching," in *International Conference on Flexible Query Answering Systems*. Springer, 2004, pp. 295–306.

[152] V. Christophides, D. Plexousakis, M. Scholl, and S. Tourtounis, "On labeling schemes for the semantic web," in *Proceedings of the 12th international conference on World Wide Web*, 2003, pp. 544–555.

[153] Q. Ren, M. H. Dunham, and V. Kumar, "Semantic caching and query processing," *IEEE transactions on knowledge and data engineering*, vol. 15, no. 1, pp. 192–210, 2003.

[154] S. Chun, J. Jung, and K.-H. Lee, "Proactive policy for efficiently updating join views on continuous queries over data streams and linked data," *IEEE Access*, vol. 7, pp. 86 226–86 241, 2019.

[155] J. D. Fernández, J. Umbrich, A. Polleres, and M. Knuth, "Evaluating query and storage strategies for rdf archives," *Semantic Web*, vol. 10, no. 2, pp. 247–291, 2019.

[156] J. Lorey and F. Naumann, "Detecting sparql query templates for data prefetching," in *Extended Semantic Web Conference*. Springer, 2013, pp. 124–139.

[157] R. Hasan, "Predicting sparql query performance and explaining linked data," in *European Semantic Web Conference*. Springer, 2014, pp. 795–805.

[158] T. Neumann and G. Moerkotte, "Characteristic sets: Accurate cardinality estimation for rdf queries with multiple joins," in *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*. IEEE, 2011, pp. 984–994.

[159] G. Cormode, V. Shkapenyuk, D. Srivastava, and B. Xu, "Forward decay: A practical time decay model for streaming systems," in *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*. IEEE, 2009, pp. 138–149.

[160] W. Zheng, L. Zou, W. Peng, X. Yan, S. Song, and D. Zhao, "Semantic sparql similarity search over rdf knowledge graphs," *Proceedings of the VLDB Endowment*, vol. 9, no. 11, pp. 840–851, 2016.

[161] W. E. Zhang, Q. Z. Sheng, K. Taylor, and Y. Qin, "Identifying and caching hot triples for efficient rdf query processing," in *International Conference on Database Systems for Advanced Applications*. Springer, 2015, pp. 259–274.

[162] H.-S. Park and C.-H. Jun, "A simple and fast algorithm for k-medoids clustering," *Expert systems with applications*, vol. 36, no. 2, pp. 3336–3341, 2009.

[163] E. S. Gardner Jr, "Exponential smoothing: The state of the art—part ii," *International journal of forecasting*, vol. 22, no. 4, pp. 637–666, 2006.

[164] A. Harth, J. Umbrich, and S. Decker, "Multicrawler: A pipelined architecture for crawling and indexing semantic web data," in *International Semantic Web Conference*. Springer, 2006, pp. 258–271.

[165] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Tech. Rep., 1999.

[166] J. Cho and A. Ntoulas, "Effective change detection using sampling," in *Proceedings of the 28th international conference on Very Large Data Bases*. VLDB Endowment, 2002, pp. 514–525.

[167] M. Saleem, M. I. Ali, A. Hogan, Q. Mehmood, and A.-C. N. Ngomo, "Lsq: the linked sparql queries dataset," in *International Semantic Web Conference*. Springer, 2015, pp. 261–269.

[168] P. Jelenković and A. Radovanović, "Optimizing the lru algorithm for web caching," in *Teletraffic Science and Engineering*. Elsevier, 2003, vol. 5, pp. 191–200.

[169] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom, "View maintenance in a warehousing environment," in *ACM SIGMOD Record*, vol. 24. ACM, 1995, pp. 316–327.

[170] P. Meinhardt, M. Knuth, and H. Sack, "Tailr: a platform for preserving history on the web of data," in *Proceedings of the 11th International Conference on Semantic Systems*, 2015, pp. 57–64.

[171] H. Van de Sompel, M. L. Nelson, R. Sanderson, L. L. Balakireva, S. Ainsworth, and H. Shankar, "Memento: Time travel for the web," *arXiv preprint arXiv:0911.1112*, 2009.

Appendix A

List of Acronyms

Acronyms

In alphabetical order:

ACR Adaptive Cache Replacement

AACP Application-Aware Change Prioritization

BGP Basic Graph Patterns

BTC Billion Triple Challenge

CB Change Based

CR Change Rate

CAMP Change-Aware Maintenance Policy

CCF Central Concept Fetching

DCS Dynamic Characterizing Sets

DBMS Database Management System

DYLDO Dynamic Linked Data Observatory

ES Exponential Smoothing

ETL Extract, Transform and Load

GED Graph Edit Distance

HDT Header Dictionary Triples

HTML Hypertext Transfer Protocol

IC Independent Copies

KBs Knowledge Bases

LOD Linked Open Data

LRU Least Recently Used

LFU Least Frequently Used

LSQ Linked SPARQL Queries

LinkedCT Linked Clinical Trials

PASU Preference-Aware Source Update

QC Query Cluster

RDF Resource Description Framework

SQC SPARQL Query Caching

SPARQL SPARQL Protocol and RDF Query Language

TB Time-Stamp Based

TTL Time To Live

TLR Triple Linear Regression

URI Uniform Resource Identifier

WSDL Web Services Description Language

WWW World Wide Web

WOL Web Ontology Language

Appendix B

List of Publications

B.1 International Journal Papers

[1] **Usman Akhtar**, Anita Sant'Anna, Chang-Ho Jihn, Muhammad Asif Razzaq, Jaehun Bang, Sungyoung Lee, "A Cache-based Method to Improve Query Performance of Linked Open Data Cloud", Journal of Computing, 2020

[2] **Usman Akhtar**, Muhammad Asif Razzaq, Ubaid Ur Rehman, Muhammad Bilal Amin, Wajahat Ali Khan, Eui-Nam Huh, and Sungyoung Lee, "Change-Aware Scheduling for Effectively Updating Linked Open Data Caches", IEEE Access (SCIE, IF: 3.557), DOI:10.1109/ACCESS.2018.2871511, 2018

[3] **Usman Akhtar**, Anita Sant'Anna and Sungyoung Lee, "A Dynamic, Cost-Aware, Optimized Maintenance Policy for Interactive Exploration of Linked Data", Applied Sciences (SCIE, IF:2.217) , Volume 9, No 22, 2019

[4] Muhammad Asif Razzaq, Javier Medina Quero, Ian Cleland, Chris Nugent, **Usman Akhtar**, Hafiz Syed Bilal Ali, Ubaid Ur Rehman and Sungyoung Lee, "uMoDT: An unobtrusive Multi-occupant Detection and Tracking using robust Kalman filter for real-time activity recognition", Multimedia Systems (SCI, IF:1.956), Vol. 26, No.5, pp.553-569, 2020

[5] Ubaid Ur Rehman, Dong Jin Chang, Younhea Jung, **Usman Akhtar**, Muhammad Asif Razzaq and Sungyoung Lee, "Medical Instructed Real-time Assistant for Patient with Glaucoma and Diabetic Conditions", Applied Sciences (SCIE, IF: 2.217), (Accepted), 2020

[6] Taqdir Ali, Jamil Hussain, Muhammad Bilal Amin, Musarrat Hussain, **Usman Akhtar**, Wajahat Ali Khan, Ubaid Ur Rehman, Sungyoung Lee, Byeong Ho Kang, Maqbool Hussain,

Muhammad Afzal, Ho-Seong Han, June Young Choi, Hyeong Won Yu and Arif Jamshed, "Intelligent Medical Platform (IMP): A novel dialogue-based platform for healthcare services", IEEE Magazine: Computer (SCI, IF:1.94), 2019

[7] Muhammad Asif Razzaq, Claudia Villalonga, Sungyoung Lee, **Usman Akhtar**, Maqbool Ali, Eun-Soo Kim, Asad Masood Khattak, Hyonwoo Seung, Taeho Hur, Jaehun Bang, Dohyeong Kim and Wajahat Ali Khan, "mlCAF: Multi-Level Cross-Domain Semantic Context Fusioning for Behavior Identification", Sensors (SCIE, IF:2.677), Doi: 10.3390/s17102433., 2017

[8] Muhammad Bilal Amin, Oresti Banos, Wajahat Ali Khan, Hafiz Syed Muhammad Bilal, Jinhyuk Gong, Dinh-Mao Bui, Soung Ho Cho, Shujaat Hussain, Taqdir Ali, **Usman Akhtar**, Tae Choong Chung and Sungyoung Lee, "On Curating Multimodal Sensory Data for Health and Wellness Platforms", Sensors (SCIE, IF: 2.033), vol. 16,no. 7, doi:10.3390/s16070980 , 2016

[9] Musarrat Hussain, Taqdir Ali, Jamil Hussain, Fahad Ahmed Satti, **Usman Akhtar**, 방재훈, 허태호, 강선무, 강병호, 이승룡, "지능형 의료플랫폼 (Intelligent Medical Platform: IMP)", 한국통신학회(정보와통신), 제37권 제9호, pp.3-17, 2020

B.2 International Conference Papers

[10] **Usman Akhtar**, Jong Won Lee, Hafiz Syed Muhammad Bilal, Taqdir Ali, Wajahat Ali Khan and Sungyoung Lee, "The Impact of Big Data In Healthcare Analytics", The 34th International Conference on Information Networking (ICOIN 2020), Barcelona, Spain , 7-10 Jan, 2020

[11] **Usman Akhtar**, and Sungyoung Lee, "Adaptive Cache Replacement in Efficiently Querying Semantic Big Data", 2018 IEEE International Conference on Web Services (ICWS), San Francisco, USA, pp.367-370, July 2-7, 2018

[12] **Usman Akhtar**, Muhammad Bilal Amin and Sungyoung Lee, "Evaluating Scheduling

Strategies in LOD Based Application", The 19th Asia-Pacific Network Operations and Management Symposium (APNOMS 2017), Seoul, Korea, Sep 27-29, 2017

[13] **Usman Akhtar**, Jamil Hussain and Sungyoung Lee, "Medical Semantic Question Answering Framework on RDF Data Cubes", 15th International Conference On Smart homes and health Telematics (ICOST 2017), Paris, France, Aug 29-31, 2017

[14] **Usman Akhtar**, and Sungyoung Lee, "Medical Question Answering Framework To Query Statistical Data", 39th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC 2017), Jeju island, Korea, July 11-15, 2017

[15] **Usman Akhtar**, Asad Masood Khattak and Sungyoung Lee, "Challenges in Managing Real-Time Data in Health Information System (HIS)", 14th International Conference on Smart Homes and Health Telematics(ICOST), Inclusive Smart Cities and Digital Health, pp 305-313, Wuhan, China, May 25-27, 2016.

[16] Hafiz Syed Muhammad Bilal, **Usman Akhtar**, Shin Won Chul and Sungyoung Lee , "Just in time Intervention for Personalized Healthcare: Behavior- Context based Intervention Adaptation", The 34th International Conference on Information Networking (ICOIN 2020), Barcelona, Spain , Jan 7-10, 2020

[17] Muhammad Bilal Amin, Sungyoung Lee, Muhammad Sadiq, Wajahat Ali Khan, Asad Masood Khattak and **Usman Akhtar**, "Configurable Data Acquisition for Cloud-centric IoT", 12th International Conference on Ubiquitous Information Management and Communication (IMCOM 2018), Langkawi, Malaysia, Jan 5-7, 2018

B.3 Local Conference Papers

[18] **Usman Akhtar**, Sungyoung Lee "Early Prediction of Heart Disease based on Scalable Linear Regression model", Heart Failure, Seoul, 2020

[19] **Usman Akhtar**, Sungyoung Lee "ACR: Adaptive Cache Replacement for Effectively Querying Semantic Knowledge Bases" KISE, November, Seoul, 2019

[20] **Usman Akhtar**, Sungyoung Lee ”A Prediction of Acute Myocardial Infarction at Early Stage Using Big Data Analytics” KOSMI, 2019

[21] **Usman Akhtar**, Sungyoung Lee. ”Performance-oriented Security Solution for Medical Information System” KISE, November, Seoul, 2018

[22] **Usman Akhtar**, Sungyoung Lee. “Measuring the Impact of Performance on Encrypted Big Data”, KISE, November, Seoul, 2016

[23] **Usman Akhtar**, Sungyoung Lee. “Measuring Data Retrieval Performance on Apache Hive”, Korea Computer Congress, jeju, 2016.

B.4 Domestic Patents

[24] 이승룡, **Usman Akhtar**, ”캐시 교체 방법 및 이를 실행하는 캐시 교체 장치”, 출원인: 경희대학교 산학협력단, 출원번호: 10-2019-0062237, 2019년 5월 28일

[25] 이승룡, **Usman Akhtar**, ”푸아송 과정 및 Self-Organizing 지도를 사용하여 연결된 데이터 클라우드에 엔티티의 역학을 예측하는 방법”, 출원인: 경희대학교 산학협력단, 출원번호: 10-2019-0002617, 2019년 1월 9일

[26] 이승룡, **우스만 아크타르**, ”효과적인 빅 데이터 질의를 위한 적응형 캐시 교체 기법”, 출원인: 경희대학교 산학협력단, 출원번호: 10-2018-0061507, 2018년 5월 30일

[27] 이승룡, **우스만 아크타르**, ”분산 캐시를 통한 데이터 웹 쿼리의 미흡한 역설 해결”, 출원인: 경희대학교 산학협력단, TBA, 2020년