Towards Empowering Data Lakes with Knowledge Graphs

Ahmed Helal

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Computer Science (Computer Science) at

Concordia University

Montréal, Québec, Canada

August 2021

© Ahmed Helal, 2021

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Ahmed Helal**

Entitled: **Towards Empowering Data Lakes with Knowledge Graphs**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

_____ Chair
Dr. René Witte

_____ Examiner
Dr. René Witte

_____ Examiner
Dr. Tristan Glatard

_____ Supervisor
Dr. Essam Mansour

Approved by _____
Dr. Lata Narayanan, Chair
Department of Computer Science and Software Engineering

_____ 2021

_____
Dr. Mourad Debbabi, Dean
Faculty of Engineering and Computer Science

Abstract

Towards Empowering Data Lakes with Knowledge Graphs

Ahmed Helal

The emergence of data lakes has permitted storing a large amount of data coming in different formats and at high speed. Data lakes are simultaneously a boon and a bane: while they are great data stores, it is tedious to explore their content. In fact, data lakes are schema-agnostic. In other words, they come with limited or no metadata, making consequently data discovery time-consuming and cumbersome. In addition, some of the already existing data lakes, like the open data portals, have few functionalities that a user can instrumentalize to look for datasets. In addition, these functionalities merely consist of basic search coupled with some filters. These limitations are costly because users would spend considerable time looking for data rather than working on their main tasks. To mitigate this shortcoming, this thesis presents an approach to create metadata on top of the content of data lakes to facilitate data discovery and data enrichment. This approach consists of two steps: First, constructing an RDF knowledge graph (KG) as a navigational structure to model the schema. Second, providing the user with a set of APIs to discover and enrich data. To demonstrate this approach, this work will present a proof of concept (POC) system that captures the schema of tabular-like data and represent it as a KG (GLac), with the means of LAC, an ontology for data lakes. Then it will equip the practitioners with user-friendly interface services to interact with GLac and compile a dataset for a given task. With these main contributions, the system offers promising results in terms of the quality of the generated schema.

The main findings of this thesis have been published in two venues: as an extended abstract named 'Data Lakes Empowered by Knowledge Graphs' [24] and 'A Demonstration of KGLac: A data Discovery an Enrichment Platform for Data Science' [25]. The former, accepted to the poster session of SIGMOD/PODS'21, presents an approach describing how to utilize KGs to facilitate

leveraging the content of data lakes. The latter, accepted to the demo session of VLDB'21, provides an overview of KGLac and illustrates the various functionalities the platform supports on top of data lakes after processing their content.

Acknowledgments

First, I would like to thank Dr. Mansour, who has helped during the 18 months I spent at Concordia University and Concordia Data Systems (CoDS) lab specifically, on several levels. As Dr. Mansour always says, CoDS is our startup, and I am happy that he entrusted me to contribute to building it and being the first to graduate from it. During my masters. I had the privilege to have a unique experience during which I learned how to conduct research, write proposals to submit to top-tier conferences, and get motivated when they are accepted. Dr. Mansour helped me work and put the building stones for KGLac, a dear project to me that has great potential. In addition, thanks to CoDS, my path crossed many people who have taught me a lot about life.

Second, I would like to thank all my friends and flatmates who showed immense support regardless of where they are staying and made my integration in Montreal smooth and with whom I share incomparable memories, especially during the covid time.

Last and most importantly, I cannot find the words to thank my parents: my father Moncef Helal, and my mother Moufida Helal. They supported me unconditionally. They always believed in me and shared with me every single moment I lived in Montreal even though they live in Tunisia. In addition, I would like to also thank my siblings and their partners: Mehdi, Imen, Khalil, Walid, Kayla, and Yosra. Furthermore, my extended family also played a crucial role and they continuously showed support. They are my uncles and Aunt: Hassan, Abdelrahman, Mohamed, Khemais, Samia, Hafedh, and Mehrez. Lastly, I cannot forget the memory of my grandparents and how it carried me during these years: Mhamed, Ahmed, Khdija, and Khadouja.

In the end, I would like to emphasize the fact that I could not reach this stage without the education I have obtained in my home country, Tunisia, that I will always love.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Since 2010, the amount of produced data has been exponentially increasing as it goes from 2 zettabytes in 2010 and reaches 79 in 2021. The number is estimated to reach 181 zettabytes by 2025 [27]. The colossal amount of data comes at a high rate and in different formats, structured, semi-structured, and unstructured. Hence, several data stores have emerged to accommodate this data characterized by its volume, velocity, and variety. For this reason, James Dixon introduced data lakes that are adequate for this kind of data [14]. A data lake is a scalable repository that allows the user to store data in different types. It also permits running analytics on top of it [1]. By adopting this data store, businesses were capable of generating 9% more in organic revenue growth than those which did not [30]. However, with all the benefits they offer, data lakes suffer from some challenges[38]. It is difficult for data scientists and analysts to reason about data due to its lack of reliability [2]. In addition, they do not provide insights about its content. They usually come with limited or no metadata leading to having data swamps [38] instead, and as a consequence, data discovery is a more cumbersome task [49]. For this reason, this thesis proposes empowering data lakes with knowledge graph technologies that capture its metadata as we show in our publication [24] in Sigmod Student Research Competition.

[1] https://aws.amazon.com/big-data/datalakes-and-analytics/what-is-a-data-lake/
[2] https://databricks.com/discover/data-lakes/introduction

Our work presents a system that generates a knowledge graph on top of data lakes representing its schema. In fact, knowledge graphs are scalable models that capture data of different formats and are suitable to unify data and capture the relationships between them [19]. In addition, the KG will enable to user to perform data discovery and data enrichment operations. These operations will help users avoid spending considerable time compiling a dataset instead of analyzing it [24]. Hence, in this thesis, we will present a proof of concept system of KGLac, a data discovery platform that processes the content of data lakes to create a KG. This knowledge graph, called GLac, captures data lakes metadata representing, as a consequence, a semantic layer on top of the data lake. In addition, the system equips AI practitioners with a set of services to seamlessly interact with GLac.

1.2 Contributions

The thesis presents three contributions:

A **Proof of Concept System** of KGLac, a data discovery platform. It is a scalable system capable of processing the voluminous tabular structured and semi-structured content of data lakes thanks to cutting-edge technologies like Apache Spark and Elasticsearch. In other words, the POC only treated tabular-like data, such as CSV and JSON files. Hence, it creates data items following a hierarchical structure where a data lake has several datasets, where each consists of several tables, and each table has several columns. Furthermore, the system captures data lake metadata to generate GLac, a highly interconnected graph. The nodes represent the various data items in the data lake and the edges reflect similarities between them. The similarities that the system establishes are content similarity, semantic similarity, primary-key/foreign-key (PKFK), and inclusion dependency. Even though it is not part of the contributions, the system has a comparable performance with the state-of-art system [12]. Our system slightly outperforms it in establishing the relationships except for the semantic ones as it gives more reliable results.

In addition, the system equips the users with a set of **interface services** to explore the content of the data lakes. These services are data discovery and data enrichment operations which we show in our demo proposal [25] accepted into VLDB'21. The operations range from basic functionalities like searching for data entities to finding the shortest path along with a given relationship. The

operations allow the user to harness the latest methods to find joinable and unionable columns using embeddings. Besides, the services promote data reusability, as the user can provide comments on each table or dataset in addition to a project using the data. Such information is an insightful reference for other AI practitioners in the future.

Moreover, the thesis presents ***LAC***, an ontology conceptualizing entities and their relationships in data lakes. Since we are in Quebec, LAC is adopted because it means lake in French. This ontology creates a unified representation of the different concepts inherited from data lakes such as columns, tables, and datasets. These concepts are the various classes forming the hierarchical structure in GLac. In addition, *LAC* provides terms for the previously mentioned relationships, in addition to those used to create the associated metadata like the number of distinct values and the path to the table. Hence, GLac leverage *LAC* and core ontologies to create the data entities with their relationships by means of several languages. The subsequent chapters will provide more details about each contribution.

1.3 Outline

The thesis consists of 6 chapters where Chapter 2 presents the related works, including examples of data lakes and how they support searching their content. Besides, it covers functionalities and systems operating on these data stores. In the end, some approaches how to map relational data to KGs. Chapter 3 highlights how KGs are adequate to represent metadata for data lakes and how they empower them to support data discovery and enrichment operations. It also covers *LAC* how GLac leverages to model the schema for data lakes. Chapter 4 presents a proof of concept system of KGLac, the data discovery platform, and lists which technologies it uses to build the KG and enables the users to interact with it. Before showing the future work and concluding in chapter 6, chapter 5 demonstrates a use case where the proof of concept system is seamlessly integrated into a data science pipeline. The data scientist will compile a dataset to investigate the compensation gap based on gender. Then, the chapter will report the evaluation of the system in establishing relationships between the nodes.

Chapter 2

Related Work

The emergence of data lakes has shown the limits of the current tools to process data. The content of data lakes is heterogeneous as they contain all sorts of data, voluminous as the data is usually in terabytes if not petabytes, and schema-less, which makes data discovery a challenging and time-consuming task. For this reason, many works have proposed scalable technologies that facilitate data discovery and help the users leverage the content of these data sources. In this chapter, we will start by reporting the current efforts to organize data in data lakes. Afterward, we will cover the different scalable functionalities that address the barriers when dealing with big data. Then we will present various systems that encompass various of these functionalities to handle data lakes. Finally, we will report some efforts to formalize ways to map structured datasets into KG.

2.1 Current Portals

In the last few years, we have noticed the continuous proliferation of data published on the web. Several reasons can explain this trend: by adopting policies opting for more transparency to the public, governments, federal[1], provincial [2], and even local [3], share their data and is now accessible via online portals. In addition, organizations publish their data for people across the globe to run analyses and infer insights. For instance, the World Health Organization [4] (WHO),

[1] https://www.data.gouv.fr/fr, https://open.canada.ca/en/open-data, https://www.data.gov/
[2] https://www.ontario.ca/page/open-government,https://www.donneesquebec.ca/,https://data.ny.gov/
[3] https://data.lacity.org/, https://donnees.montreal.ca/, https://opendata.vancouver.ca/pages/home/
[4] https://www.who.int/data/gho/

regularly publishes data related to COVID-19. Democratizing such data has enabled countries to learn more about the virus and better manage the pandemic. Furthermore, other portals like Kaggle [5] and OpenML [52] host data provided by several users. This data has enabled the improvement and development of new technologies in Machine Learning (ML) to ameliorate and devise new models. Other portals dedicated to specific domains also exist. The Scientific Data in Nature[6] or the Inter-university Consortium for Political and Social Research[7] (ICPSR) promote research reproducibility in the natural sciences and the social sciences respectively. These portals provide data of different formats (CSV, JSON, HTML) and are organized in different domains (Agriculture, Military, Law). They also include some helpful metadata such as the size of the data and the publisher. Moreover, the maintainers of these data portals assist users looking for a specific dataset and could not find it.

The presence of such portals can be regarded as a boon as they serve many goals such as more transparent governance and faster research reproducibility. However, dealing with big data also comes with a bane. Data discovery is time-consuming [12]: one has to look for data and then investigate to verify whether it suits their task. The data portals, unfortunately, do not equip the user with adequate operations to perform data discovery. They merely provide keyword search capabilities with some filter operations over the existing metadata, which is limited. For instance, on Kaggle, a user can look for a dataset and filter them based on its size, or the license attached to the data. The Canadian Open Portal provides similar functionalities. But in addition to that, it offers the option to find comparable records. Despite the usefulness of such an operation, the user cannot determine based on which metric the datasets are similar. Is it based on content similarity, same publisher, or even a combination of several variables? The limited metadata available on data portals and data lakes hinders the user's task to find data, leading to spending considerable time possibly ending up without finding what they look for. For this reason, by introducing KGLac, we aim to address this limitation and leverage the data hosted on these data lakes to derive metadata. Hence a user can perform data discovery in a faster and more effective manner.

[5] https://www.kaggle.com/
[6] https://www.nature.com/sdata/
[7] https://www.icpsr.umich.edu/web/pages/

2.2 Functionalites and Algorithms

The previous data portals represent tools to organize data lakes having basic search and filter operations. Nevertheless, to improve the data discovery user experience, the hosts need to integrate more adequate tools. In other words, these tools need to be scalable and fast to handle the voluminous data. For these reasons, several efforts have introduced new algorithms and operations to include in these data portals.

In [37], the authors suggest a new approach to organize data in data lakes. This structure represents a navigational structure to determine if two datasets belong to the same topic. The proposed organization utilizes a probabilistic Directed Acyclic Graph (DAG). The DAG is a Markov model that represents the metadata. Its nodes are the attributes extracted from the data lake, and the edges are the transitions from a parent, a set of attributes, to its children, subsets of these attributes. The leaf nodes refer to the actual data in the data lake. By extracting the attributes for each topic, the system calculates the corresponding embedding vector to determine how similar a given topic is to another one and construct the organization. While this organization is a helpful tool to find data belonging to the same topic, the construction of the model is contingent on the existence of the attributes in the data lakes. The creation of the model is time-consuming, and it is prone to any modification like the insertion or deletion of a dataset. On the contrary, KGLac does not depend on such metadata; it instead creates metadata for data lakes.

JOSIE [63], an algorithm that helps users find joinable tables in data lakes based on overlap set operations. The algorithm is scalable and optimized to quickly retrieve the top-k most similar candidates to a given column without investigating the columns in the data lake. KGLac, however, is a more general system. In addition to being a framework to find joinable tables, it provides operations to find unionable ones, and explore the data lake. Plus, KGLac deals with both numerical and textual data whereas JOSIE merely handles textual data.

D^4 [41] introduces a data discovery tool dedicated to infer semantic type for columns in tabular data. The main focus is to handle textual data posses semantics, while numerical ones can lead to false results as they can belong to various domains. The approach groups semantically related terms of a column and then verifies whether this grouping belongs to a given topic. In other words,

by investigating the values of the two columns, D^4 can report if they stem from the same variable or domain. In contrast, KGLac supports findings of semantically related columns based on the embedding of the content and the column name. Additionally, it also reports on the existence of primary key - foreign key between two columns which, in addition to being semantically related, an inclusion dependency should exist. Plus, given that numerical columns are considerably pervasive in data lakes, KGLac supports findings if two numerical columns are similar based on their contents or the semantic of their names.

2.3 Systems

The algorithms and approaches mentioned above are valuable to integrate into more comprehensive systems that provide a set of functions to perform data exploration and data discovery over data lakes. In this section, we will list several systems meant to tackle data discovery. We will compare them based on a set of features.

Metadata: It is essential to support data discovery operations over data lakes. Even the simplest operations such as keyword search and filter operations rely on the existence of such salient information like the title of the dataset or its format. Several systems aim to leverage this metadata to facilitate the task of finding datasets. [60] is a system to address tables relatedness. The authors investigate the techniques dealing with data values and data domain overlap to support various operations on tables. To do so, they extract metadata ranging from creating profiles for the tables, their schema, and descriptions if available and provenance. For the provenance, they rely on the sequential execution of the data in notebooks like Jupyter [45]. [12] collects metadata by profiling the datasets. Based on the content, it computes several statistics and embeddings to establish links between the data entities. [56] helps users build tables using Wikipedia and its KG DBpedia. They rely on the schema provided by the KG to find data used to enrich a given table and build it from scratch. [8] helps users find data based on keyword search and filters. It collects metadata by relying on markups and tags associated with the dataset when published online. In addition, the system utilizes Google KG to extract further metadata helpful to refine the search. [22] and Amundsen[8]

[8] https://www.amundsen.io/

connect to data stores like relational databases from which they extract the metadata. KGLac, similar to Aurum, extracts the metadata by leveraging the content of the data by building data profiles. It, however, supports inferring more information conveying information about data completeness.

Data Model: After extracting the metadata, it needs to be modeled and subsequently leveraged to support the data discovery operations. [60] stores the data on a postgreSQL [50] and models the inferred relationships, such as the provenance, between the data, cells, and notebooks on Neo4j [9], a property graph. [12] leverages the extracted profiles to build an in-memory graph where the nodes are the profiled data entities, and the edges are the relationships that might exist between the entities, such as primary key - foreign key relationships or schema similarities. Goods [22] uses HBase [10] to index its metadata as key-value tuples. Finally, Amundsen models the metadata and their relationships as a graph stored in Neo4j. KGLac models the metadata as an RDF-star [23]. To do so, we provide an ontology to model data stemming from data lakes. [42] presents an approach for a system capable of enriching structured and semi-structured data using external KGs both domain-specific and cross-domain. To enable enriching data, they first determine the domain of the tables to be treated. Then create an ontology modeling this data to match it with the domain-specific external ontology. In the end, they extract the module (a subset of the ontology) that contains richer information used to enrich the initial data. Hence, this approach models the data as a KG to improve the quality of the structured and semi-structured data. Hence, both the proposed system and KGLac would rely on building a KG to attain data enrichment. To build the graph, [42] models a table as an data entity with their associated columns as data properties. KGLac also structures its graph by building a hierarchical structure having three classes: Column, Table, and Dataset. In addition, [42] merely relies on semantic meaning while matching their KG with the external one. However, KGLac supports in addition to semantic similarity between the different columns on content similarity and the existence of a Primary key / Foreign key relationship which provides a richer and more connected graph.

Functionalities: By leveraging this metadata, the systems offer various operations. [60] creates metrics to determine table relatedness. In fact, given a table query and the type of relatedness,

[9] https://neo4j.com/
[10] http://hbase.apache.org/

the system can report the top-k results. The table relatedness help retrieve unionable, joinable, and cleaner versions of the given query tables. [12] allows the user to look for the data in the data lakes through a keyword search on the level of the source dataset, file, or attribute. Since it models the metadata as a graph, it also leverages the graph structure to discover related data through intermediate nodes along with a given relationship. Moreover, either by starting from scratch or adding a new field, a user uses the [56] interface to build tables; It provides union and join operations. to do so, the system internally converts the user selection of the data they are interested in from Wikipedia or DBpedia into SPARQL queries. To abstract these functionalities, KTabulator relies on KG querying and table transformation. [8] provides keyword search and filtering capabilities based on the extracted metadata. [22] and Amundsen also provide keyword search, but also provide user interfaces to present the extracted metadata for a given data entity. In addition, Goods provide a monitoring tool, where the user can track the status of a dataset across time and explore any other trend affecting the data, such as an increase in size. KGLac provides basic search and filter operations in addition to union and join ones. It leverages the graph structure to find the shortest path between two data nodes along a given edge. It also allows the users to annotate the data nodes with any feedback for users to refer back if they want to use the same data. Plus, it enables the users to write ad-hoc SPARQL queries in case it was not already pre-defined. Furthermore, [42] support data enrichment by matching the built ontology for the initial data with external cross-domain and domain-specific KGs. However, KGLac supports data enrichment using the data initially parsed and reflected in the KG.

Usability: The different systems have several interfaces to allow the end-user to perform data discovery. For [12] introduces a new language, the Source Retrieval Query Language (SRQL) to enable the user to interact with the graph with pre-defined operations. [8] provides a similar interface as data portals where you can search by keyword. The results offer the option to access the web page hosting the dataset in addition to some metadata. However, for KGLac interface services users only need to be accustomed to technologies like Jupyter [45] notebooks and frameworks like Pandas [11] which is the case, as the system is for data practitioners.

Table 2.1 summarizes the features supported by the different systems. We can notice that most

[11] https://pandas.pydata.org/

Table 2.1: Summary of the features supported by the systems

	Extracted Metadata		Supported Functionalities		User Interface
	Model	In-silo?	Keyword Search	Union and Join	
Juneau [60]	Property Graph	✓	✗	✓	Notebook Extension
Aurum [12]	In-memory Graph	✓	✓	✓	SRQL
KTabulator [56]	NA	NA	✗	✓	GUI
Google Dataset Search [8]	RDF	✗	✓	✗	GUI
Goods [22]	Key-Value tuples	✓	✓	✗	GUI
Amundsen	Property Graph	✓	✓	✗	GUI
Semantic Enrichment [42]	KG	✗	NA	NA	NA
KGLac	RDF-star	✗	✓	✓	Notebooks and IDEs

of them model the metadata as a graph. This structure is adequate to capture relationships between the data entities. In addition, it offers flexibility to add, update, and delete its nodes or edges. Furthermore, given that these systems perform data discovery, we need to investigate the different functionalities. It is worth mentioning that in addition to keyword search, union, and join operations, the systems provide additional functionalities. But because each system supports a different one, it is not feasible to reflect that in the summary table. For instance, Juneau supports the users to find cleaner versions for a given one while the others do not. Moreover, Goods enables the users to monitor the status of a given dataset if its size increases in a specific trend or a change in restriction. Last, we report the user interface each system supports. The majority of systems provide a Graphical User Interface (GUI). As this is helpful for non-technical people to use the system, they still need to learn how to use it. For instance, KTabulator for [56], the authors had to run training sessions for a group of users to evaluate the system. Hence, it is critical to provide an intuitive and user-friendly interface to allow the user to perform data discovery.

2.4 RDB to RDF

Table 2.1 shows that only the Google Dataset Search System, the Semantic Enrichment Platform, in addition to KGLac models the extracted metadata as a Resource Description Framework (RDF). This model allows the merging of several graphs. This is possible because nodes are unique via Uniform Resource Identifiers (URI). Hence if two nodes having the same identifier but belonging to different KGs, refer to the same entity after merging the graphs. That's why [8] leverages its internal KG to enrich the crawled metadata. RDF graphs have triggered an interest among researchers to map relational databases into RDF graphs. Tim Berners-Lee [5] suggests a simple mapping approach where the value in a column corresponds to a node, the predicate is the name of the column its cell value will be the value of the node in the graph. This approach can be generalized regardless of the domain the data belongs to. That's why it can miss semantics that these data may convey [47]. Another approach is to perform the mapping by relying on a domain-specific ontology. This approach requires either using a pre-existing ontology or building a new one. However, it allows capturing more semantically-rich mappings in addition to reducing redundant triples used to create the RDF graph [47]. The World Wide Web Consortium (W3C) formed the RDB2RDF Working Group to formalize a standard language to map relational data and schema into RDF. To perform the transformation, which sets the floor for more intricate ones, the group introduces Direct Mapping [1] (DM).In addition, they formalized R2RML [13] a language to enable the customized transformation of the relational data to RDF.

Besides, in [48], the authors report that businesses face several challenges such as accurately answering critical questions (daily sales) or optimizing business decisions. These challenges are due to the lack of a unified vocabulary that users across the different teams can understand without any ambiguity. Hence to overcome such limitation, a common ontology has to be introduced. And to do so, the authors present a methodology to build an Enterprise Knowledge Graph (EKG) from Relational Databases. It involves multiple actors including IT Developers, Business Users, and Knowledge Scientists. This methodology consists of three steps: Knowledge Capture when the different actors sit together to understand the requirements and disambiguate any notion discrepancies. The second is Knowledge Implementation. The scientist works on formalizing the ontology and

the KG coupled with the various queries. Lastly, it is the Self-Service Analytics phase: Business Users perform their tasks using the new unified model and approve its production if no problem is encountered.

Mapping relational databases to RDF offers several benefits as capturing semantic and adopting a more flexible schema. However, the mapping was mainly limited to mapping the actual data records into RDF nodes. Plus, work involving capturing the schema of relational data and model it as a KG does not exist to the best of our knowledge. So instead of mapping the column name to a predicate, it can be modeled as a node. This node can be linked to other attributes such as the type of the column or the total number of missing values, and the edges are the relationships between the different nodes. KGLac adopts such an approach and builds a KG for the metadata of relational data and provides a formal ontology to have a standard representation and a common vocabulary.

Chapter 3

Background

In the previous chapter, we reported several systems that work on top of the data lakes. One could notice that the majority model their data to perform data discovery as a graph, two of them specifically use KGs. Hence, in this chapter, we will continue this discussion to investigate KGs by reporting why they are adequate models to for data lakes. In fact, The Google Dataset Search engine [8] models the extracted information as a KG. To enhance the results reported to the user, the system also leverages the Google Knowledge Graph to infer more information. Google first introduced its KG in 2012 to improve its search engine. For instance, if you search ”Concordia University” you will find an infobox (fact panel) on the right, summarizing all the information about the institution like its address and notable alumni. This feature did not exist before 2012, but thanks to content captured in its KG, it has become possible. Even though Google did not release any technical details about the implementation of its KG, many other companies like LinkedIn, Microsoft, and Facebook, have also adopted this structure. The adoption of KG to model the voluminous, rapidly-growing, and various data, reflects the importance of this structure for these companies. This chapter presents a background about KG that sets the floor for the subsequent chapter. The following sections will present different definitions of what a KG is. Then they cover its structure and how ontology help appoort meaning to it. Then they end with listing KG characteristics and their applications.

3.1 KG: Importance & Definitions

3.1.1 KG Importance

As of 2012, KGs gained more attention and popularity and became the interest of several research labs and companies. Figure 3.1 shows the cumulative number of new Knowledge Graph publications every year between 1980 and 2020. The data were manually collected from Google Scholar using the keyword 'knowledge graph'. The figure shows that 2012 was a turning point as, before that date, the number of citations was not significant compared to 2012 onwards as the number exponentially increased. The graph shows that the number of publications since 2010 has doubled every two years. This observation indicates the growing interest in Knowledge Graphs. However, before digging deep into the importance of KG and the popularity it gained, it is crucial to define what a KG is.

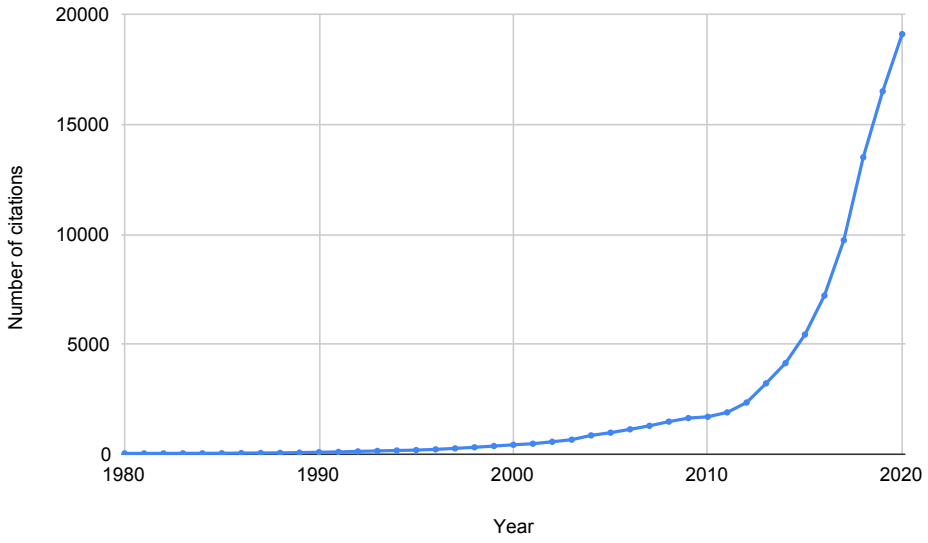


Figure 3.1: Cumulative number of KG publications from 1980 to 2020 on Google Scholar

3.1.2 Several Definitions

Defining KGs has been contentious [26]. Some tried to present them from a technical perspective describing the underlying structure, whereas others focused on their application. Paulheim lists four criteria to consider a system to be a knowledge graph. A knowledge graph (i) mainly describes

real-world entities and their inter-relations, organized in a graph, (ii) defines possible classes and relations of entities in a schema, (iii) allows for potentially interrelating arbitrary entities with each other, and (iv) covers various topical domains [43]. Paulheim lists these four criteria to consider a given system to be a Knowledge Graph: it has to be a flexible graph reflecting any real-world scenario given a defined schema. By describing real-world entities, there is a motivation to adopt KGs to model the content of data lakes and build GLac. In addition, a KG defines the different classes such as tables and columns and relationships such as primary-key/foreign-key between two columns and that a column belongs to a table. By definition, KGs offer the flexibility to include concepts to represent semi-structured data from data lakes. Details about the different supported classes are in section 4.1.2. Furthermore, this definition highlights that KG covers several domains. In other words, a KG can be a cross-domain one that offers the flexibility to not only model data in data lakes but also enrich it with content from other domains. However, this definition is just a list of characteristics of what a KG offers rather than what it is. It gives the minimum set of traits to distinguish KGs from Knowledge Collections [16].

Additionally, [31] reports that Knowledge Graphs are large networks of entities, their semantic types, properties, and relationships between entities. Reporting that these KGs are large means that they are scalable to model the content of the data lakes characterized as voluminous. Yet, the definition is vague: it misses providing details about the nature of the graph and what is meant by ”semantic types”. Other definitions gave a formal notation of KGs. For instance, [18] defines a Knowledge Graph as a Resource Description Framework (RDF) graph. An RDF graph consists of a set of RDF triples where each RDF triple (s,p,o) is an ordered set of the following RDF terms: a subjects $\in$ U $\cup$ B, a predicate p $\in$ U, and an object U $\cup$ B $\cup$ L. An RDF term is either a URI u $\in$ U, a blank node b $\in$ B, or a literal l $\in$ L. This definition is technical as it formally presents the various components of an RDF KG. However, it restricts KG to only being modeled as RDF. In fact, in addition, they can be represented as property graphs, which the definition misses. Despite serving the same purpose of modeling KG, each of these structures has its advantages and drawbacks. Ontotext, a global leader in enterprise KG and semantic database compares these data models in Table 3.1 [1]. Hence, while the adoption of either model depends on the use case, RDF is more

[1] https://www.ontotext.com/knowledgehub/fundamentals/rdf-vs-property-graphs/

Table 3.1: RDF vs Property Graphs

Feature	RDF	Property Graph
Expressivity	Arbitrary complex descriptions via links to other nodes; no properties on edges. With RDF-star [23], the model gets much more expressive than PG.	Limited expressivity, beyond the basic directed cyclic labeled graph. Properties (key-value pairs) for nodes and edges balance between complexity and utility.
Formal Semantics	Yes, standard schema and model semantics foster data reuse and inference.	No formal model representation.
Standardization	Driven by W3C working groups and standardization processes.	Different competing vendors.
Query Language	SPARQL specifications: Query Language, Updates, Federation, Protocol (end-point).	Cypher, PGQL, GCore, GQL (no standard).
Serialization format	Multiple Serialization formats.	No serialization format.
Schema Language	RDFS, OWL, Shapes.	None
Designed For	Linked Open Data (Semantic Web): Publishing and linking data with formal semantic and no central control.	Graph representation for analytics.
Processing Strenghts	Set analysis operations (as in SQL, but with schema abstraction and flexibility.	Graph traversal. Plenty of graph analytics and ML libraries.
Data Management Strenghts	Interoperability via global identifiers. Interoperability via a standard: schema language, protocol for federation, reasoning semantics.	Compact serialization. Shorter learning curve. Functional graph traversal language (Gremlin).
Main use cases	Master/reference data sharing in enterprises. Knowledge representation. Data Integration. Metadata management	Graph analytics and path search.

expressive and standardized. In addition, the user has the flexibility to serialize the content to be reflected in the KG without the need to learn different query languages. Yet, while Property Graphs are more suitable for graph traversal and analytic, some of the RDF data stores like Blazegraph [51] provide a set of analytical functionalities on top of the hosted RDF graph.

Furthermore, some other definitions highlighted how a KG potentially provides inference capabilities to extract more knowledge. [16] reports that a Knowledge Graph acquires and integrates information into an ontology and applies a reasoner to derive new knowledge. To elaborate, the author assumes that a KG is superior and more complex than a knowledge base (KB). In other words, for a KB to be a KG, it needs to be coupled with a reasoning engine to generate new knowledge and integrate it into the KB. Thus, reasoning capabilities can infer more relationships between the different KB components resulting in a richer and more connected KG. This continuous enhancement leads to better data discovery results. Yet, this definition is not formal and merely provides an overview of an architecture for a system that supports KGs. Hence several papers present various attempts to define KGs, but each tackles them from a particular perspective. A compilation of them illustrates the structure, schema, characteristics of KG, and some of its applications. The sections

will investigate these different definitions by going more in-depth by first explaining the structure of KG that is best suited to model the data lake content. Second, address the semantic of the different entities via the ontologies. Third, cover the numerous characteristics offered by KGa thanks to their structure and ontologies. And forth, report the diverse data discovery-related applications supported on top of KG via leveraging these characteristics.

3.2 KG: Structures & Ontologies

3.2.1 KG Structure

There exist two types of graphs for KG: RDF or property graphs. Table 3.1 compares between these two structures. Despite the possibility to convert RDF to property graph or vice-versa, KGLac will rely on RDF graphs to generate the KG. GLac is a metadata graph generated on top of data lakes to support data discovery and data integration. These purposes match the uses cases why to use RDF graphs. In addition, having one language query to interact with graph and having one standardization ensured by W3C does not overwhelm the user to learn several languages to write their queries nor understand the graph. Yet, it gives them the flexibility to import and serialize data in several formats. As a result, in the rest of this work, KG is modeled using RDF technologies. This section covers in more detail the various constituents of an RDF knowledge graph.

A Knowledge Graph or also a semantic network is a directed labeled graph. It consists of nodes N, relationships R, and labels L. Nodes represent data entities like students, professors, and countries. The relationships are the edges connecting these nodes. For instance, there is a relationship with the label ”is_from” between a student and a country. This triple (subject, predicate, object) conveys simple information such as a student S is from a country C. The set of triples (N * L * N) constitutes the graph to model any real case situation. This simplicity is one of the reasons why these structures are suitable to model knowledge, hence the name Knowledge Graph.

There are three types of nodes: IRI-based nodes that refer to actual entities, with IRI referring to the Internationalized Resource Identifier. Usually, they contain an IRI for a given data entity on the internet. For instance, on DBpedia, a KG for Wikipedia, the IRI referring to Concordia University is https://dbpedia.org/resource/Concordia_University. This IRI plays the role of an identifier allowing

the user to distinguish Concordia University from any other entity in the KG having the same name such as Concordia University Chicago with the IRI https://dbpedia.org/resource/Concordia_University_Chicago. The IRI-based node can be either a subject, predicate, or object. Second, nodes can be literals; they contain attributes associated with the IRI-based nodes, such as the name or faculty size. For instance, "Concordia University (en)" is a string and represents the name while its faculty size is 2419 (xsd:integer). A literal node can only be an object in the triple. Third, a node can be blank, representing a non-distinguishable entity with no specified IRI. Technically, these blank nodes can serve in representing containers like bags and sequences. The blank node can either be a subject or an object. Fig 3.2 depicts the structure of the sample graph showing information about Concordia University. The information is from DBpedia [3]. The nodes colored in blue represent IRIs for the different data entities, like the ones corresponding to Concordia University or its alumni. The red colors represent the literals containing the attributes like the name or the nationality associated with the IRI nodes. It is worth mentioning that a literal comes with their types (xsd:integer) or the language of the string values such as (en) for English. In addition, the blank in the figure of the type bag is a container that connects the IRI node representing Concordia University with those of its alumni.

This section illustrates the distinct constituents of KGs, what type of nodes to use to reflect the actual data entities, their attributes, and how to establish links between them. However, [43] mentions that KG should reflect real-world entities. So while creating several constituents to model the various data entities, specifying their semantic is equally crucial. To do so, KGs instrumentalizes ontologies which the next section covers.

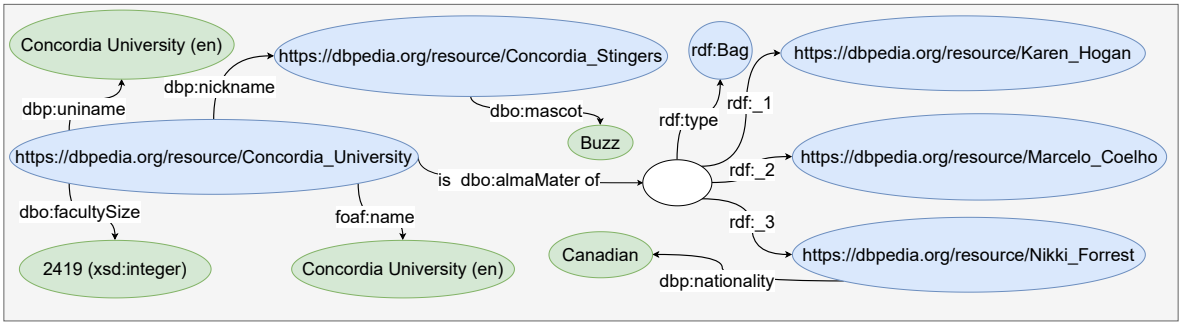


Figure 3.2: RDF structure representing information about Concordia University

3.2.2 Ontologies

Ontology is originally a branch of philosophy that studies the theory of being. It aims to develop a system with categories and reflects the intrinsic relationships between them. KGs adopted ontologies to embed meaning to the modeled data. This section defines what an ontology is and how they represent the being of KG by carrying semantics to the different entities in the KG.

[20] defines an ontology as an explicit formal specification of share conceptualization. This definition can be a bit shallow and does not provide further details about what properties an ontology can offer. Thus, one should examine this definition in detail. First, conceptualization represents the concepts of the domain given ontology tackles and the relationships between them. In other words, if the ontology provides terms about geography, it has to take into consideration the concepts of a country, city, currency, language, plus the relationships between them such as capital_of, located_in. Plus by being explicit, the ontology has to unambiguously convey the meaning or semantics to the entities it describes. For instance, if you say Carthage, does it mean the city in Canada or Tunisia? Does it even refer to the civilization in Tunisia? Or maybe Carthage the novel? An ontology must disambiguate such confusion and provide the user with the meaning of the various entities in the graph. Furthermore, it has to be formal. As mentioned in the previous section, a KG supports inference via the reasoning engine. Hence, a machine does not only need to read the data but also has to understand its meaning. For instance, if we consider data about the population of Canada stored in CSV files without headers. The machine will be able to read the content of the different columns but cannot tell which column represents the first name and which one is the last name. Hence, for a machine to understand the meaning of the data, its ontology has to be formally presented. Finally, The ontology has to be shared, meaning that others can use it to model and describe the data they have without creating their own. For instance, to represent data about social networks in a KG, the user does not need to come up with their terms, they can reuse the ontology *foaf* [9] (The Friend Of A Friend), which is suitable as it provides terms about the concepts and relationships about people and the interactions between them.

Furthermore, the observation shows that the terms ontology and vocabulary are used interchangeably. There is no quite a clear distinction between them. They both provide terms (concepts

and relationships) for a specific domain. However, ontology is usually associated with a more formal and complex definition of their terms while vocabularies offer more loose ones [2]. Another concept that is also confused with ontology is taxonomy. A taxonomy represents the hierarchy between the different classes in ontology. For example, in Figure 3.3, the dbo:Singer, dbo:musicalArtist, and dbo:Artist represent some of the classes the DBpedia ontology defines. However, the hierarchy between these classes where dbo:Singer is a subclass of dbo:musicalArtist also a subclass of dbo:Artist represents the taxonomy.

Hence, an ontology provides the set of terms and relationships related to a given domain. However, with this, there is the limitation of how to materialize and describe these different terms. For a further illustration, consider the following scenario: How to specify that Canada and Tunisia belong to the same class country. And how to specify that the predicate capital takes as a domain a city and a range country. The limitation lies with the need to have a mechanism to describe such information in the KG. For this reason, W3C introduced RDF Schema [21] (RDFS) as a language to describe groups and relationships. It consists of a set of terms for data modeling and description. Hence, RDFS is an ontology and serves the purpose of a language to specify the schema for a given ontology. To specify the domain and range of a predicate, RDFS provides the terms rdfs:domain and rdfs:range. In addition, thanks to RDFS, rdfs:property specifies that a given relationship is of type property. By including these descriptions, the KG can capture the semantics and meaning of its content. RDFS also provides a list of other terms to describe classes, such as rdfs:Class, rdfs:Resource, or even rdfs:Literal. In addition, some terms describe the predicates, so in addition to rdfs:domain and rdfs:range mentioned earlier, there are rdfs:subClass and rdfs:subPropertyOf. These terms define the foundation for specifying taxonomies and supporting inference. Similar to RDFS, there exists OWL [4] the Web Ontology Language, also introduced by W3C to make web content more accessible and interpretable to machines. In other words, it enables processing the data and not just present it on the web. OWL differs from RDFS by being more expressive and richer as it consists of a richer vocabulary to describe the entities in the KG. [33] presents a list of the different terms OWL defines.

This section defines what an ontology is, its features, in addition to the languages (RDFS and

[2] https://www.w3.org/standards/semanticweb/ontology

OWL) used to describe the distinct concepts. So far, this chapter covers the structure of a KG and how to embed meaning to its various constituents. The subsequent section lists the diverse characteristics of KGs, illustrating why they are adequate structures to model meta-data of a data lake content.

3.3 KG: Characteristics & Applications

3.3.1 KG Characteristics

After exploring the structure of RDF-based KGs and the ontology accompanying them, this section lists a myriad of their characteristics. Based on the definitions in section3.1.2, it explains how they help mitigate the different challenges faced when capturing meta-data for the content of data lakes. The content of data lakes is voluminous and rapidly changing and presents several constraints that data stores have to accommodate. These challenges are The scalability of the system, the dynamic schema, and the need to capture relationships between the data entities.

Scalable data stores: Data lakes host humongous data. The US open portal hosts approximately 300K datasets, impossible to store on a single machine. Hence for a system to accommodate this constraint has to offer horizontal scalability. Yet, vertical scalability is not adequate because the data is not only large but also continuously increasing. Hence, the amount of data will eventually exceed the capacity of the host and can lead to missing data. Accounting for such constraints is significant for companies as it enables them to meet the growing usage of their system. For example, Facebook in 2014 reports generating 4 petabytes of data a day [54] which is equivalent to 1460 petabytes a year, which a single machine cannot store. **Schema:** The data in data lakes is not homogeneous as it comes in different formats (structured, semi-structured, or unstructured). In addition, the content is dynamic. In other words, data keeps changing with the insertion or deletion of a data entity. Data stores have to accommodate such constraints without performing labor-heavy and time-consuming tasks to adjust the schema with every modification. For instance, for a relational database, the insertion or the deletion of new data will potentially require resetting the constraints and redesigning the tables. Hence, the data store has to support a malleable schema. **Relationships between data**

entities: Despite the heterogeneity of data in data lakes, there exist relationships between its entities. Such information is valuable as combining information extracted from the various sources can be complementing and enriching. For example, [8] leverages its Knowledge Graph to enrich the information extracted about the datasets after crawling the web pages. Hence, the adequate stores have to model the data in a structure that preserves this inter-connection.

KGs are adequate to address these constraints. Large companies employ them to model their data. In [40], different KGs curated by large companies were reported. All of them contain millions of entities. For instance, Microsoft is building a graph with 2 billion entities and 55 billion facts, while Google KG has 1 billion entities with 70 billion assertions. A single machine cannot store such voluminous structures. More suitable distributed and scalable systems exist to address this limitation. We will provide more details about them in section 3.3.2. As for the schema, KGs do not impose on the user to define the schema before loading the data. It can be defined and easily changed at any time. Unlike RDBs, when an insertion or deletion of a data entity occurs, it does not engender costly schema updates. To define the schema, users can use some of the already existing ontologies like RDF and schema, as detailed in 3.2.2. Lastly, to model the relationships between the data entities, KGs capture such information and model them as predicates. Hence, KGs appropriately address the different constraints faced when modeling data stemming from data lakes. Plus, a KG is extendable. In fact, by identifying the same data entity with the same IRI, merging two KGs lead to aggregating the information associated with them into a unified KG. For instance, let us consider that ministry of higher education of Quebec models its data as a KG and refers to Concordia University by https://dbpedia.org/resource/Concordia_University. Then by linking it with DBpedia, the ministry can enrich its knowledge base about Concordia University as the IRI node referring to the university will have additional information that the ministry would not have before.

This section presents the diverse characteristics of KGs used to address the different constraints associated with representing data in data lakes. By being a flexible, scalable, and extendable model, several companies have adopted the Knowledge Graph as a structured model to create a unified framework supporting their multiple services.

3.3.2 KG Applications

By leveraging the different characteristics of KG, the development and enhancement of many applications in Artificial Intelligence (AI) and data science have become possible. However, to support these applications, there is a need to host these KGs on dedicated data stores to enable them to interact with these semantic networks. Thus, this section will first cover some of the open data stores that can host KGs and provide operations on top of them. Then, it will list several applications in several domains like medicine, security, and finance.

To interact with the KG, users need to host it on supporting data stores like Apache Jena [11] and Blazegraph [51]. These stores host the data and enable the users to write queries. SPARQL is a query language for RDF KGs. It is similar to SQL yet relies on matching graph patterns to report the results. In addition, these stores support the latest enhancement to represent KG: Both Apache Jena and Blazegraph support RDF-star and SPARQL-star. In addition, the latter provides a set of functions as an abstraction of writing more complex queries by supporting the traversal and analysis of the KG. Some of the functionalities are finding the shortest path between two nodes and reporting the intermediate nodes coupled with their associated attributes. In addition, they support the Page Rank and finding connected components algorithms. This continuous support for KGs and the growing community reflects the importance KGs gained in the last few years.

These structures represent the intermediate between KGs and the users to develop various applications. In fact, Similar to word embedding, now many techniques using Graph Neural Networks [55, 61, 62], generate **Knowledge Graph Embeddings** (KGE). Hence, regardless of the size of the graph, edges and vertices will be mapped into latent rich and compact vector representations of the KG used by the models to render better results. For instance, in unsupervised learning, these representations can be beneficial to carry node clustering. The result can be helpful to perform merging two KGs by establishing relationships between nodes that are similar. By leveraging these embeddings, the user is equipped with enhanced capabilities to perform data discovery. Specifically, in the field of biomedicine, researchers started investigating leveraging KG embeddings to perform drug discovery techniques [6].

In addition, a KG is adequate for supporting **inference** which can benefit data discovery. A

Knowledge Graph in the literature is a knowledge-base system that is composed of a knowledge-base and a reasoning engine to infer new information[16]. By creating rules and relying on mathematical logic, the reasoner uses them to infer more relationships between the nodes. To apply inference over the KG, the user will need to specify the schema of the different entities in the KG. Fig 3.3 present a snapshot from DBpedia showing actors starring in the movie Titanic with some attributes associated with them. To differentiate the IRI nodes from the literals, we use the blue and green colors respectively for the nodes. The figure shows the absence of relationships between the different actors even though they starred in the same movie. To enrich the graph, we can establish links between them by leveraging the inference capabilities supported on top of KG. In a real case scenario, if two actors act in the same movie, then they should know each other. To reflect this in the graph, by relying on the rule-based approach, the introduction of a rule can indicate that if two actors starred in the same movie, then they know each other. Formally, $\forall A1,\ A2,\ M\ if\ (A1\ dbo:starring\ M) \wedge (A2\ dbo:starring\ M) \rightarrow (A1\ foaf:knows\ A2) \wedge (A2\ foaf:knows\ A1)$. Hence, using the reasoning engine, we can create a link between the actors in the figure to indicate that they know each other. This information can be helpful to understand the relationships between actors. For instance, in the case of a larger sample, we can create a rule to establish stronger relationships than *foaf:knows* between actors who performed in more than one movie. Another approach to further infer insights from the KG is by leveraging the class hierarchies. In Fig 3.3 Celine Dion belongs to the class Singer and the Singer class is also a subclass of musicalArtist, which is also a subclass of the class Artist. In fact, by understanding this hierarchy, one can conclude that Celine Dion is also an artist. Formally, $\forall A\ C1\ C2\ C3\ if\ (A$ rdf:type $C1) \wedge (C1$ rdfs:subClassOf $C2) \wedge (C2$ rdfs:subClassOf $C3) \rightarrow (A$ rdf:type $C2) \wedge (A$ rdf:type $C3)$. In other words, Celine Dion is also a musicalArtist and an Artist. This information is helpful as it enhances the search results for entities belonging to the class Artist. It can also be helpful to establish links between artists of different domains like Celine Dion and Kathy Bates.

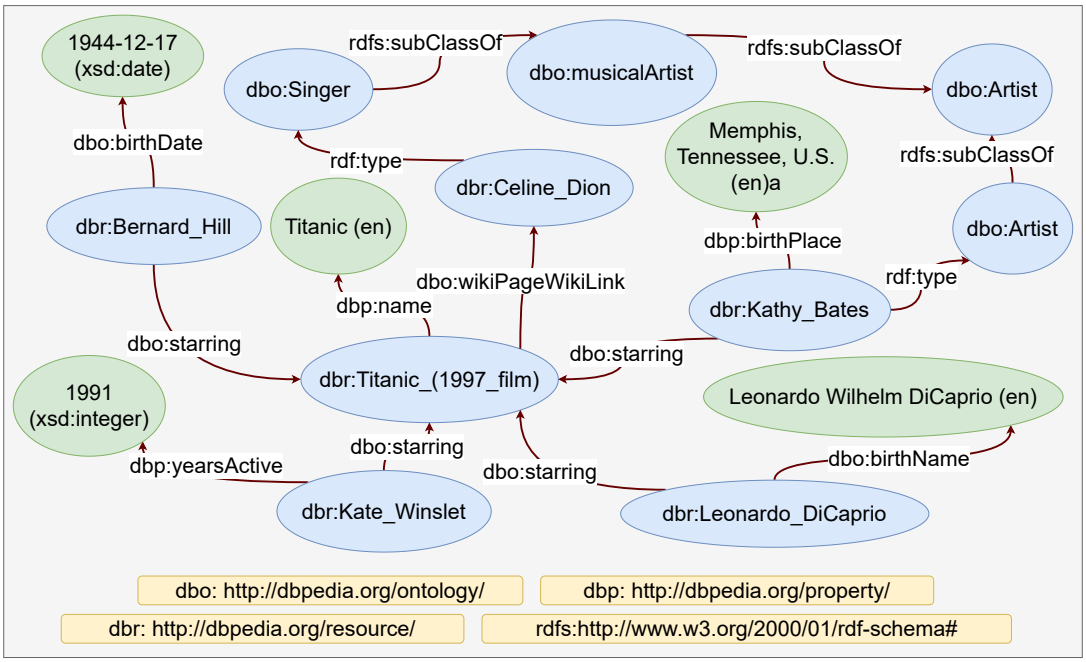


Figure 3.3: Reasoning engine infers foaf:knows between actors starring in the Titanic

This section reports the different stores where a user can host KGs and how a user can leverage these networks to perform several data discovery operations. In conclusion, KGs combine a malleable structure with semantically rich constituents. These traits offer several characteristics used to support many applications. All these features make KGs fit to capture meta-data for data lakes. The next chapter will cover KGLac, the system responsible for creating the KG. It will go into detail into the pipeline, starting from loading the raw data until loading the KG into an RDF store. In addition, the next chapter will provide details about how the end-user can interact with GLac via the supported interface services. In addition it will present *LAC*, a data lake specific ontology and GLac, the highly interconnected KG representing the caputred schema of data lakes.

Chapter 4

A Data Lake ontology & A Proof of Concept System

The previous chapter presents the importance of KGs and data lakes can benefit from them. To capture the schema and model it as a GLac, the KG, we developed a system that processes raw data to create the KG. To this end, this chapter presents KGLac, a proof of concept system responsible for creating GLac and providing a set of interface services for the user to perform data discovery and data integration on top of the KG. KGLac starts by sharding the different tabular data and create data profiles stored on a document database. It then loads these profiles to model the data nodes in GLac. It also instrumentalists these profiles to establish the various relationships between the different data entities. After hosting GLac on a suitable RDF-store, KGLac equips the user with a set of APIs to perform data discovery and data integration operations ranging from a simple keyword search to finding the shortest path between two nodes given a specific relationship. For further details, this chapter will start by presenting the overall architecture of KGLac and its different components. It will also present *LAC*, the data lake specif ontology used to create GLac. Second, it will explain the various technology choices used to build the components, and last, it will highlight the contributions of KGLac.

4.1 System Overview

Figure 4.1 consists of two main components: the GLac construction and Interface Services. The former is responsible for data profiling and store the results on a document store, then leverage it to build GLac based on learned representations (embeddings). The latter provides discovery operations based on SPARQL queries and embedding similarity. Figure 4.2 shows the workflow between these different components. First, the profiler will load the raw data from the local data lake. It will create profiles for the columns and then stores them on a document database. The GLac builder will then load these profiles to generate GLac, the highly interconnect KG, and then hosts it on RDF-star store. The last step consists of equipping the user with APIs to interact GLac to perform data discovery and enrichment. The user can also update it by providing insights and feedback. The user also can query the raw data if they implicitly configured the profiler to store it. This section will go over each component in detail. It starts by presenting the data profiler, how it processes the data, what metrics it computes to create data profiles and how it stores them.

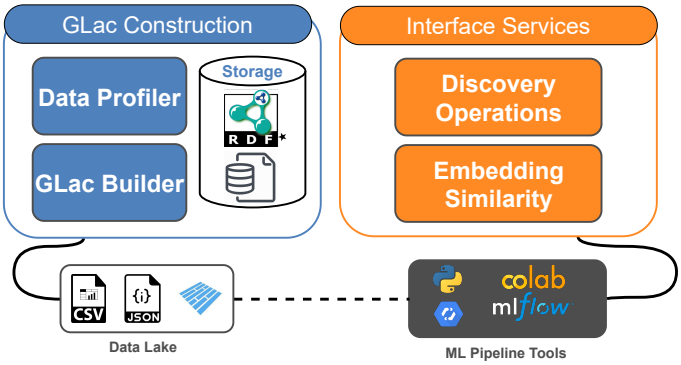


Figure 4.1: The proof of concept system architecture, where KGLac gets access to a local data lake, e.g., sets of files or databases, to construct GLac. Then, different ML pipeline tools can communicate with KGLac to facilitate data discovery. KGLac tracks the use of datasets.

4.1.1 Data Profiler

The data profiler is one of the essential components of KGLac: it aims to create data profiles for tabular raw data. Data profiling refers to the activity of creating small but informative summaries of a database. In our case, data profiling will collect statistics and information on the different columns in the database [28]. The content of the produced profiles contains viable metrics to create

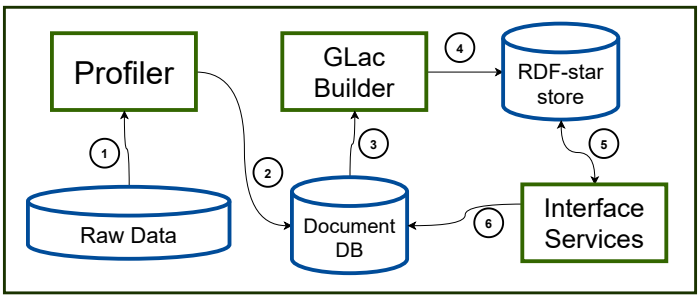


Figure 4.2: KGLac workflow: 1-Profile the raw data. 2- Store the profiles and the raw data on a document database. 3- Load the profiles. 4- Host GLac on an RDF-star store. 5- Query and update GLac. 6- Query the raw data.

the meta-data for the data lakes. These metrics range from simple statistics such as the number of null values and distinct values in a column, its data type, or the most frequent patterns of its values [39]. However, to profile the raw data, the user would need to specify the different datasets to parse. To do so, the user would first need to create a YAML file where they need to provide the name, path to the dataset in addition to the type of data it contains. For now, KGLac would require the data type to be in a tabular format, such as CSV, JSON, or Parquet. Once the configuration file is ready, the user can start the profiler to parse the YAML file and extract the content of this dataset. The content encompasses numerous tables, for each KGLac creates a separate class to include its path and data type and then appends to a queue [1].

The used array is synchronized as it supports multithreading. The profiler must be scalable to handle the voluminous data via using multithreading. The number of threads or workers is configurable. In other words, the user can change the number, which is by default set to 4. The threads dequeue a table as KGLac enqueues them while parsing the configuration file. Each worker will perform three tasks: First, it will interpret the table. In other words, it will shard the table into a set of columns. Then, it will infer the data type of each of these columns. Second, based on the inferred data type, it will profile each column to extract suitable statistics and information. Last, the thread will index the profile on the document database. KGLac can also store the raw data. However, the user has to enable this functionality before running the profiler. Once the thread finishes with these three tasks, it dequeues another table if the queue is not empty.

The loaded tabular data usually lacks schema. Hence, it is not straightforward to determine the

[1] https://docs.python.org/3/library/queue.html

type of data a column contains. The data type is essential because, based on it, KGLac decides which statistics and information it needs to compute. For instance, to calculate the interquartile range (IQR), the difference between the 75th and 25th percentiles to reflect the statistical dispersion, the values need to be numerical. Such metric is not applicable on a textual column containing first names, for example. Thus, KGLac parses the data to determine the data type, which is either textual or numerical. For this, each thread will load the table in a PySpark dataframe. PySpark serves two purposes: first, it reflects another aspect of the profiler scalability [57, 58] and second, it determines the data type of the columns. In fact, while loading the data, a user can pass the argument inferSchema to obtain the data type for each column. Pyspark does determine whether a column is numerical or textual. Instead, it reports data types of a lower granularity like int, float, or smallint. For this reason, KGLac aggregates these numeric types (int, float, double, bigint, smallint, tinyint) under the class numerical, and all the other types will be of type textual.

After determining the data type of each column, the profiler will profile them. It is worth mentioning that the profiling will be merely on the level of the columns. In fact, for each table, the thread separates the numerical from the textual ones. Then it will run different analyzers to calculate the statistics and retrieve the information that will constitute the profile. Regardless of the column data type, the profiler determines the following attributes: the total number of values, the number of distinct values, and the number of missing values. These metrics are insightful to perform data discovery by telling the size of the data or its completeness. KGLac uses PySpark to calculate these metrics. In addition, the profile will include the path of the table it belongs to, the data type, its name, and the name of the table and the datasets. One can observe that the last three attributes are unique. Hence, the profiler uses then to generate an id to uniquely identify the column using the zlib [2] library. Furthermore, for numerical columns, the profiler will compute the minimum, maximum, and mean values. In addition, it will approximately calculate the quartiles in calculating the median and IQR with a configurable error set to 0.03. As for textual columns, the profiler computes its column embedding using minHash using datakstch [3]. These compact and latent representations with size 512 are suitable to determine how similar two textual columns are.

[2] https://docs.python.org/3/library/zlib.html
[3] https://github.com/ekzhu/datasketch

After creating the profiles, the thread indexes them on a document database, namely Elasticsearch [4]. This is a distributed store, which is adequate to handle the content of data lakes. For this reason, the worker will connect to Elasticsearch, serialize the profile and then index them. In addition, if the user enables it, the worker can store the raw data on the document database. Having access to this information helps the users if they want to find data based on the raw data rather than the captured schema in the profiles. Given that the columns can be large, KGLac divides the content into smaller chunks of size equal to 1000. After indexing all the profiles, the index will handle another table from the queue if it is not empty. In conclusion, the profiler shards the raw data into columns, and for each, it creates a data profile and indexes it on a document database. Figure 4.3 summarises the profiler workflow. The next step consists of leveraging these profiles to create GLac, thanks to the GLac builder component.

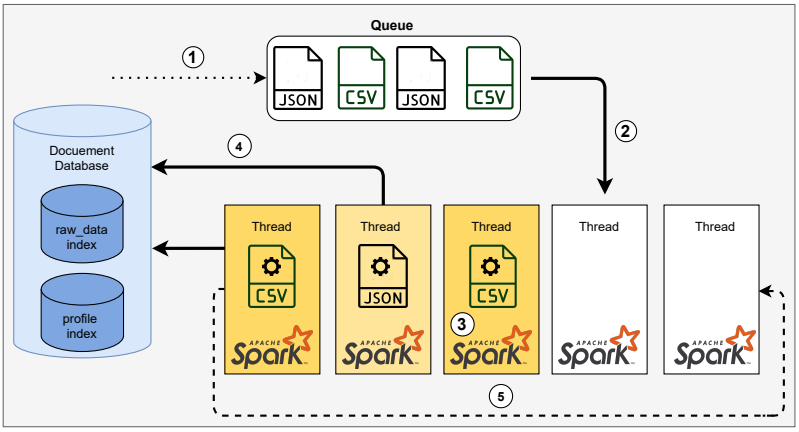


Figure 4.3: Profiler Workflow: 1-Load the tables into a priority queue. 2- Each thread loads a table. 3- Each thread determines the data type of each column and then profiles it. 4- Each thread loads the profiles and the raw data into a document database on a separate index. 5- Thread waits to process another table when available.

4.1.2 LAC: Data Lake Ontology

To build GLac, KGLac harness several existing ontologies such as RDF and Schema. It also utilizes *LAC*, an ontology that provides terms to represent various concepts for tabular structured and semi-structured data in data lakes which this chapter thoroughly presents.

[4] https://www.elastic.co/

To convey semantics to data, KGs leverage ontologies. Several cross-domain and domain-specific ones exist. For instance, DBpedia [3] provided several terms to represent information extracted from Wikipedia while *foaf* describes social networks and their activities concepts. For this reason, to represent the different entities stemming from the data lakes, a set of terms reflecting data lake concepts should be used. Such ontology does not exist. Yet several efforts have tried to provide vocabularies to represent data entities from structured data such as R2RML [13], a language used to map entities from relational databases to RDF resources. However, these efforts do not cover all the concepts within data lakes, such as semi-structured and unstructured data. Thus, we introduce our ontology *LAC* which stands for lake in French that provides terms to model schema entities for structured and semi-structured data from data lakes. It is important to note that currently *LAC* does not support unstructured data. However, given that ontologies are extendable, the ontology can include terms for this type of data in the future.

To start formalizing *LAC*, we examined the different data portals mentioned in 2.1. They provide access to several datasets containing data in different formats. Hence our terms should include the notion of a dataset. It is a resource that contains a set of data entities that belong to a common domain. These entities can either be a set of files of different formats or datasets. Hence, the structure is similar to a folder in a file system where a folder contains files or folders. Furthermore, the different files published within a dataset come in different formats that the user is responsible for parsing if they want to explore them. Some of these types are relational data that a user can host on a relational database management system to explore. In addition, there exist JSON files that a user should parse to access its content or a PDF that a user would need to read or leverage Natural Language Processing techniques to process and understand. *LAC* will be, for the moment, limited to tabular-like structured and semi-structured data. Some of these data formats are relational tables, CSV, JSON, XML files, and excel sheets. Although processing these numerous formats is different from one to the other, they have a tabular-like structure. Even though JSON files are documents, they offer the flexibility to be rendered and represented as tables. Hence, to simplify the ontology, all these files of different types will be considered tables and expressed as resources of type lac:Table in the ontology. So, *LAC* should represent these resources to belong to a class of type 'table'. Tables can be further broken down into columns or rows. But since *LAC* intends to reflect the schema for

entities in data lakes, then columns are adequate to consider because a column naturally comes with a set of attributes such as name, type of instances, which can constitute a schema. In addition, these columns contain instances that stem from the same variable or domain that *LAC* needs to infer. In addition, let us assume we reflect rows instead of columns, then the KG has to represent data on the level of instances which is not the purpose of *LAC* which intends to provide terms to represent the schema for data in data lakes and not the data itself. In addition, working on the level of rows would require modeling hundreds if not thousands of rows which would overwhelm the KG without bringing much semantic to reflect.

Hence, the granularity *LAC* would support modeling of the schema for structured and semi-structured data consists of the classes dataset, table, and column. These classes represent a taxonomy where a column is part of a table, a table is part of a dataset, and a dataset can also be self-contained KGLac will render these resources as nodes. Between these nodes, there exist several relationships that *LAC* should capture. The Canadian Data Portal provides functionality to look for similar datasets. Even though there is no indication of what kind of similarity exists between two datasets, *LAC* should reflect such insightful information to create an interconnected KG. KGLac establishes relationships between the resources based on their content and schema. For instance, by considering two columns, there is potentially a content similarity relationship if the raw data values of the columns belong to the same domain. For example, if two columns contain names of countries, then KGLac can establish such a relationship. In addition, each column has a name that KGLac can harness to measure if there is a semantic relationship between two columns. For instance, region and country can be semantically close. It is worth noting that these two relationships are independent; One does not entail the existence or the absence of the other. In addition, similar to relational databases, there can exist a primary key / foreign key (PKFK) relationship between two columns. Thus, in addition to providing terms to model these similarity relationships, *LAC* can provide terms to reflect the PKFK edge. Hence KGLac supports four types of relationships: content similarity, semantic similarity, PKFK, and inclusionDependency. The first three relationships can be generalized to link between tables and datasets by leveraging the edges of their columns and tables. The last indicates that the values of a column are a subset of the content of the other. The following chapter will provide more details about how to establish these relationships. To create

these various relationships, KGLac calculates several similarity metrics. These measures reflect the certainty of establishing such links between the different data entities. To include these certainty metrics in the KG, KGLac uses RDF-star to annotate the edges. For this reason, *LAC* introduces the edge lac:certainty that accepts as domain a triple and a certainty score as a range.

Table 4.1: LAC different classes

Name	Type	Part Of	Description
lac:Dataset	rdfs:Class	lac:Dataset	A class to model a dataset in data lakes. A dataset contains a set of data that belong to the same domain.
lac:Table	rdfs:Class	lac:Dataset	A class to model tabular-like data files such as CSV and JSON.
lac:Column	rdfs:Class	lac:Table	A class to represent a column in a table class.

The different resources in the KG come with a set of attributes. To reflect these attributes, KGLac uses some of the core ontologies such as RDF and SCHEMA. But for others, it is not feasible to reflect without *LAC*. Some of these attributes are annotations. On data lakes, users can comment on a dataset or annotate it, which is insightful for the community. In other words, by reading the multiple comments and annotations, a user can understand the context of the dataset or the different projects that used the data. To model such information, *LAC* provides lac:insights and lac:used_in. Each of these edges leads to a bag where to append the different annotations and usages. Even though both of these attributes provide insightful information for research reproducibility, there is a semantic difference between them. Lac:insight allows users to provide a short description of the dataset based on their experience. Lac:used_in allows the mention of the usage of such data, for instance, the link to the project using it. Another useful attribute to reflect is the path of the dataset or the table. Thus, *LAC* provides the term lac:path using which a user can easily load the content of the resources and use it. *LAC* provides terms to model statistical attributes extracted from the data entities. Lac:totalVCount, lac:distinctVCount, and lac:missingVCount respectively associate the data entities with literals for the total, distinct, and missing number of values a column contains. The metrics provide the user information on whether the data is adequate to join and union with or even data completion. Finally, Table 4.1 and 4.2 summarise the different terms of both classes and properties (edges) to model table-like structured and semi-structured data. In addition, Fig 4.4 provides the schema of *LAC* whose terms are colored.

LAC is an effort that aims to put the building stones for an ontology to model both the entities

Table 4.2: LAC different properties

Name	Domain	Range	Type	Description
lac:semanticSimilarity	lac:Column	lac:Column	rdf:Property	Indicates that the two columns are semantically similar based on their names.
lac:contentSimilarity	lac:Column	lac:Column	rdf:Property	Indicates that the two columns have similar content based on their instances.
lac:inclusionDependency	lac:Column	lac:Column	rdf:Property	Indicates that the content of a column is contained in other's.
lac:pkfk	lac:Column	lac:Column	rdf:Property	Indicates that a column is foreign key to primary key column.
lac:used_in	lac:Dataset lac:Table lac:Column	rdf:Bag	rdf:Property	Indicates the different usages of the data entity resource.
lac:insights	lac:Dataset lac:Table lac:Column	rdf:Bag	rdf:Property	Indicates the different insights provided by the user on the data entity resources.
lac:path	lac:Dataset lac:Table	rdfs:Literal	rdf:Property	Provides the path of the dataset or table in the file system.
lac:totalVCount	lac:Column	rdfs:Literal	rdf:Property	Provides the total number of instances in a column.
lac:distinctVCount	lac:Column	rdfs:Literal	rdf:Property	Provides the number of distinct instances in the column.
lac:missingVCount	lac:Column	rdfs:Literal	rdf:Property	Provides the number of missing instances in the column.
lac:certainty	Triple	rdfs:Literal	rdf:Property	Annotates a triple to reflect how certain of an similarity, inclusion dependency, or pkfk edges to be established.

and their relationships from data lakes in a KG. This effort is prone to continuous modification and improvement depending on the requirements and the data to represent in the graph.

4.1.3 Glac: A Highly Interconnected KG

KGLac generates GLac, a highly interconnected graph to represent the schema for the content of data lakes. While. *LAC* is a core component used to create GLac, KGLac, utilizes other already defined ontologies. Reusing them is cost-effective because there is no need to engineer and maintain a new ontology that models terms that other efforts have already addressed. In addition, ontologies are understood to provide a common knowledge representation. Hence all the different actors operating on the KG will have the same understanding of the content. As a result, in addition to being cost-effective, reusing ontologies promotes the interoperability of the application [7], GLac in our case. KGLac uses different terms provided by RDF[5], RDFS[6], Schema[7], OWL[8], and Dct[9]. Since GLac captures the meta-data for the content of the data lake, SCHEMA provides several terms that are used to associate the different core classes (column, table, and dataset) with their respective

[5]http://www.w3.org/1999/02/22-rdf-syntax-ns
[6]http://www.w3.org/2000/01/rdf-schema
[7]http://schema.org/
[8]http://www.w3.org/2002/07/owl
[9]http://purl.org/dc/terms/

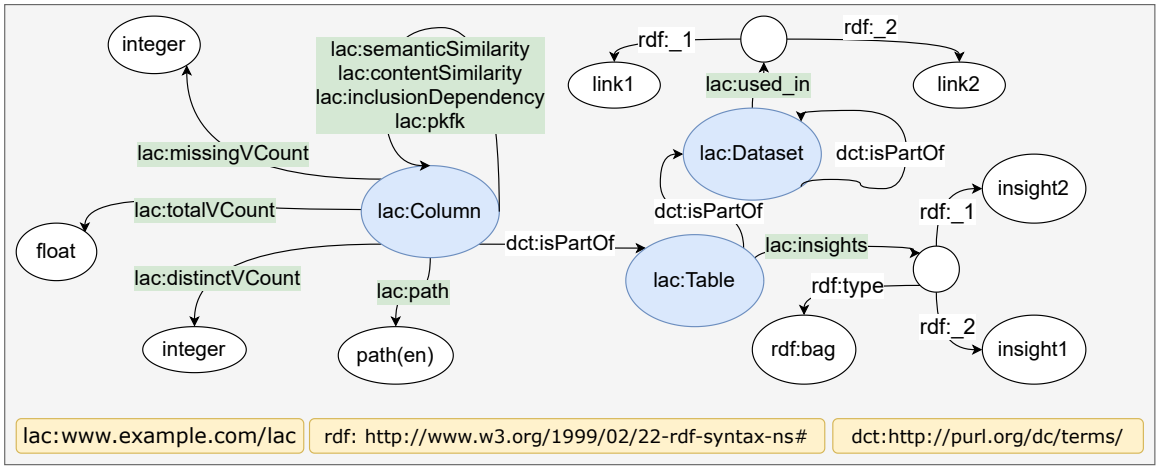


Figure 4.4: Schema of *LAC*

attributes, such as the name of the column and some statistics about their content. In addition, the relationships between the core classes in GLac constitute hierarchical relationships where a column is part of a table and the table is itself a part of a dataset. For this, dct provides the property isPartOf. This relationship is intended to be used with non-literal values and describes a resource in which the described resource is physically or logically included. Furthermore, to specify the classes, KGLac uses RDF:type. Plus, to include the cardinality of a column (the ratio of the number of unique values and the number of total values), and create a human-readable name, owl:cardinality and rdfs:label are used respectively. Table 4.3 summarises the terms from ontologies other than *LAC* used to create the properties in GLac. Figure 4.5 provides a instance for GLac using all the different ontologies with the terms from *LAC* colored in blue.

Hence, we cove GLac, a highly interconnected KG, to represent the meta-data for the content of data lakes. GLac instrumentalizes the concepts covered several already-defined ontologies such as RDF and Schema in addition to *LAC*, an ontology to model concepts captured in data lakes. To build GLac, our system, KGLac processes and leverages the raw data. So far, we have covered the architecture of the proof of concept system and presented the GLac in addition to the used ontology. The next section will present GLac builder, the component responsible for creating GLac by instrumentalizing *LAC*.

Table 4.3: Terms from other onotology used to build GLac

Property name	domain	range	Description
schema:name	lac:column lac:table lac:dataset	rdfs:Literal	State the name of the resources
schema:type	lac:column	rdfs:Literal	State the type of the column content (Textual or Numerical)
schema:median	lac:column	rdfs:Literal	State the median of the numerical column
schema:minValue	lac:column	rdfs:Literal	State the minimum value in the numerical column
schema:maxValue	lac:column	rdfs:Literal	State the maximum value in the numerical column
dct:isPartOf	lac:column lac:table lac:dataset	lac:column lac:table	A related resource in which the described resource is physically or logically included.
rdf:type	lac:column lac:table lac:dataset	rdfs:Literal	State that a resource is an instance of a class
rdfs:label	lac:column lac:table lac:dataset	rdfs:Literal	Provide a human-readable version of a resource's name
owl:cardinality	lac:column	rdfs:Literal	Restricts a class to a data value belonging to the range of the XML Schema datatype nonNegativeInteger

4.1.4 GLac Builder

After creating the profiles, KGLac is now ready to build GLac via the GLac builder. This component harnesses the content of these profiles to create the various nodes and their attributes, in addition to statistical and deep learning techniques to establish links between them. The GLac builder starts by loading the profiles from elasticsearch. For each profile, it creates the column, table, and dataset IRI-based nodes. The GLac builder needs to assign a unique id for each of these nodes. That's why it uses the id, the profiler generated for the column. As for the table, the GLac builder generates an id using its name and the name of the dataset it belongs to using zlib. The same applies to the dataset node with just using its name. After creating the various first-class nodes, the component creates a hierarchy between these different classes using dct:isPartOf. In the end, for each column, the builder associates it with its attributes such as the name, data type, and cardinality, which reflects the uniqueness ratio; it materializes them as literal in the GLac and using the terms covered in section 4.1.2. By the end of this step, the GLac builder creates the different vertices GLac contains. Next, it interconnects the column nodes via establishing similarity relationships which the remaining section covers in detail.

The GLac builder starts by creating semantic relationships between the different columns. The existence of a semantic relationship entails that two columns names have a close meaning. To

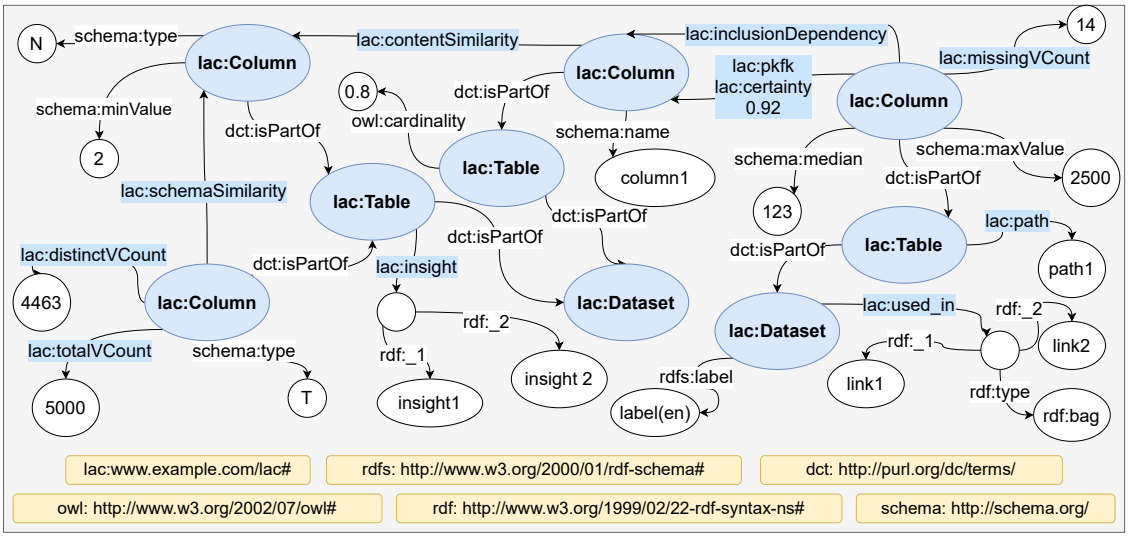


Figure 4.5: Instance of GLac

determine such affinity between them, the GLac builder computes an embedding for the column labels using the approach mention in [32]. Such techniques are adequate to apply to tabular data. The method relies on word embedding [35, 44]. Word embedding has been to determine how similar two words are because it takes into consideration the context of the word in the document. The GLac builder uses the Glove [10], a pre-trained word vector using Wikipedia data with a vector size equal to 50. In addition, to create the relationship, the affinity has to be above a particular limit. Both the pre-trained model and the threshold are configurable. The GLac builder forms a triple having both the subject and the object the nodes representing the column with predicate having lac:semanticSimilarity as a label. Furthermore, The semantic affinity between the two columns reflects the certainty of the existence of the relationship. Hence, GLac captures such metric and uses it to annotate the predicate between the column using RDF-star, a more compact representation to make statements about statements [2]. For instance, if we consider two columns in GLac if ids 1 and 2 having a semanticSimilarity relationships with certainty 0.75, then

<<lac:1 lac:semanticSimilarity lac:2>> lac:certainty 0.75

is how the GLac builder would serialize it in turtle-star with lac being a prefix for www.example.com/lac.

[10] https://nlp.stanford.edu/projects/glove/

The GLac builder also leverages the values in each of the profiled columns. Depending on the data type of the column, the component utilizes a different approach. For textual columns, the profiler computed their embedding using MinHash [10]. Hence, the GLac builder will query the MinHash vectors from elastic search in addition to the column id. To calculate the similarity between the column using the MinHash and the Jaccard similarity metric, comparing each column with the rest is not scalable and is time-consuming. To mitigate this barrier, the GLac builder uses a two-step approach: First, it uses the MinHash Local Sensitive Hashing [29] (LSH) to index the different MinHash vectors with the corresponding column id. The LSH will cluster similar columns by hashing the fed vectors with a high Jaccard distance into the same bucket. Second, for each column, the builder will pass its MinHash vector as a query to the index, which will return the ids associated with the MinHash vectors. The returned vectors have a Jaccard similarity above a specified threshold. By using this approach, the system infers a content similarity relationship between two textual columns. Similar to the semantic affinity between the column, the Jaccard similarity score between the two vectors reflects the certainty behind establishing the predicate that GLac captures through RDF-star.

Furthermore, the GLac Builder creates content similarity relationships between numerical columns. It applies the same approach as [12]. For each, it connects to Elasticsearch to load the associated id, the max and min, median, and IQR values. Then it approximately computes the domain the column values cover by subtracting and adding the IQR to the median. Then, it interconnects the columns by creating content similarity edges between them. Column A has a content similarity relationship with column B means that the overlap between their domains is above a certain threshold. The overlap is similar to the Jaccard similarity, as it reflects the certainty that the content similarity relationship exists. In addition, if the range of the domain is equal to 0, then the GLac builder clusters these single points using DBSCAN [11]. For all the columns in the same cluster, the GLac builder creates a content similarity relationship with certainty equal to the overlap threshold. Hence, for both numerical and textual columns, the GLac builder calculates the overlap between the pair to determine if there is a content similarity between them. Yet, the approaches are different: for the textual, it uses the Jaccard similarity coefficient, as for numerical, it computes the domain overlap.

[11] https://scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html

Similar to relational databases, there exist primary key/foreign (PKFK) relationships between columns. For this reason, the GLac builder will leverage the different data profiles to establish such edges between the various column in GLac. For the GLac builder to determine such a relationship, there are two conditions to satisfy: First, a primary key column must have a high uniqueness ratio. Second, the foreign key column must have an inclusion dependency (IND) relationship with the primary one. An IND condition between two columns indicates that all the values of one are contained in the values of another one [15]. For the first condition, the GLac builder initially computed this ratio named cardinality by dividing the number of distinct values by the total number of values. The cardinality has to be above a configurable threshold. For the second condition, the GLac builder treats columns depending on their data type. In other words, for the textual one, it is sufficient to check whether there is a content similarity between a pair of columns based on the MinHash. While for the numerical values, the condition is more restrictive than the content similarity. It requires that the range of the values of the contained column is within the container's. In addition, the overlap between the column domain needs to be above a certain threshold [12]. Thus, the KGBuilder needs first to generate inclusion dependency edges between the numerical columns. Then, to have a PKFK between a pair of numerical columns, it is sufficient to find an inclusion dependency relationship between them to meet the second condition. To determine the inclusion dependency constraint between two columns, the GLac builder relies on these heuristics to simultaneously comply with the definition that all the values of a column are a subset of the other one and offer a scalable system. Like all edges, GLac also captures the certainty for PKFKs relationships which is the highest cardinality between the two columns.

In conclusion, the GLac builder leverages the profiles to create the nodes in GLac and interconnect its columns by generating four types of edges: semanticSimilairy, contentSimilarity, inclusionDependency, and PKFKs. These edges are helpful to support data discovery and data enrichment operations supported by the interface services, which the next section covers.

4.1.5 Discovery Operations

After creating GLac and hosting it on an RDF store supporting RDF-star, the user can performing data discovery and enrichment operations using the interface services component. It equips the

user with a set of APIs to connect with both the RDF host and Elasticsearch. These APIs include discovery operations that query the graph based on SPARQL-star, an extension to SPARQL supporting RDF-star. In addition, they support fetching data based on embeddings to enrich data via union and join operations. For further details, this section lists the various discovery operation the POC system supports.

Given the large size of data lakes, the user typically does not know its content. Yet, they usually have an idea about what they need to find. That's why they use keyword search as an initial step. For this reason, the interface services component is not different. It provides a set of basic operations to explore the content of the data lakes. A user can search the content of the graph on various granularities. They can get the different datasets and all the tables or the tables in a particular dataset. Also, they can look for a specific column. For all of these operations, the user needs only to specify the entity name or a regular expression to obtain the matching results. In addition, if a user is interested in looking for a table having some specific columns, they can search for a table based on a set of conditions. These APIs return rich content containing the name of the table, which dataset it belongs to, in addition to the number of columns and rows. As for the ones operating on columns, the result contains several attributes, such as the number of missing values, the cardinality, or the median. The APIs encapsulate the output in Pandas dataframes. Supporting such features serves many objectives: Usability, Pandas is a common framework in data science, so the user does not need to learn a new data structure to manipulate the data. For example, assume the user searches for tables given a regex and obtains several results. Yet, they want to find the ones with the number of rows above a particular threshold. They can, in this case, harness the dataframe APIs to sort or filter the results. Interoperability, the user can easily convert these dataframes to others like PySpark dataframes when the content is voluminous. Data Representation, Pandas is adequate to present tabular data, which is the type of data KGLac. Hence, to explore data lakes, the user can utilize these APIs as initial steps.

4.1.6 Embedding Similarity

In addition to data discovery, the interface services equip the user with APIs to enrich their data using embedding similarity techniques. Data enrichment is crucial to the data-science life-cycle

because the new data contribute to uncovering new insights, highly predictive features to enable answering interesting questions [53]. KGLac supports data enriching by finding unionable and joinable columns in the data lakes. The former augments the content of the data by inserting more rows, while the latter adds more features by appending more columns. To union two tables, KGLac relies on semantic similarity between the column names [46]. In other words, there must be similar columns to union on. As a result, KGLac offers an API that takes two dataframes and reports back a Pandas dataframe containing pairs of the column names to conduct the union. This approach takes into consideration semantic similarity between the names. The API does not depend on GLac to perform this operation because the user can rename the columns. Hence, the service extracts the column names and uses the same approach as GLac builder to create semantic similarity between two columns. It then keeps track of the result whose affinity is above a threshold to build a dataframe and return it to the user. Ultimately, the user chooses from the reported pairs, based on which, they merge the two dataframes.

Furthermore, KGLac supports finding joinable columns using their content. To join two tables, each should have a column with similar content. KGLac harnesses the content similarity and PKFKs relationships to find such columns. For instance, assume a user has a table about world happiness and aims to understand factors contributing to it. By looking for neighbors of the column countries along a PKFK or content similarity edges, the user finds a table containing general information about each country like the capital, demographics, and GDP. The last two columns can be insightful features the user can add. However, this approach assumes that the tables are already materialized in GLac. In other words, GLac contains only the entities or relationships modeling data the profiler processed. To mitigate this limitation, KGLac adopts a two-step approach: First, it processes the new data to generate embedding for the textual column or calculates the domain for the numerical ones. Second, similar to GLac builder, it compares these computations with those of the columns loaded from ElasticSearch to determine content similarities. The returned result consists of pairs of columns the user can choose from to merge the dataframes. At this stage, the user needs to interfere to enrich the tables. But the advantage KGLac offers is limiting the number of columns they need to manually investigate. In conclusion, the union and join APIs simultaneously rely on embedding queries and statistical measures to find matching columns.

Table 4.4: Different APIs supported by the Interface Services

API	Description	Arguments
get_number_of_datasets	Get the number of datasets in the GLac.	- show_query: Print SPARQL query.
get_tables_in	Get the tables in a dataset.	- dataset_name: Name of the dataset. - show_query: Print SPARQL query.
get_all_tables	Get all the tables in GLac.	- show_query: Print SPARQL query.
search_columns	Search for columns in GLac.	- keyword: Regex for the column name. - show_query: Print SPARQL query.
search_tables	Search tables by name.	- table_name: regex for the table name. - show_query: Print SPARQL query.
search_tables_on	Search tables containing specific column names.	- conditions: List of conditions for the column names. - show_query: Print SPARQL query.
get_shortest _path_between_columns	Get the intermediate data entities between the starting column and the target one. It uses the GAS service provided by Blazegraph.	- col1_info: Pandas series representing the starting column. - col2_info: Pandas series representing the starting column. - via: String to specify the relationship. - max_hops: Number of maximum hops to consider. - show_query: Print SPARQL query.
get_content_similar_to	Get columns having content similarity with the input.	- column: Pandas Series representing the column. - show_query: Print SPARQL query.
get_semantically_similar_to	Get columns having semantic similarity with the input.	- column: Pandas Series representing the column. - show_query: Print SPARQL query.
get_pkfk_of	Get columns having PKFK edge with the input.	- column: Pandas Series representing the column. - show_query: Print SPARQL query.
add_dataset_insight	Comment on the dataset.	- dataset_name: Name of the dataset to annotate. - insight: Comment on the dataset. - show_query: Print SPARQL query.
add_table_insight	Comment on the table.	- dataset_name: Name of the dataset the table belongs to.. - table_name: Name of the dataset to annotate. - insight: Comment on the table. - show_query: Print SPARQL query.
get_dataset_insight	List the comments on the dataset.	- dataset_name: Name of the dataset to annotate. - show_query: Print SPARQL query.
get_table_insight	List the comments on the table.	- dataset_name: Name of the dataset the table belongs to. - table_name: Name of the dataset to annotate. - show_query: Print SPARQL query.
add_dataset_usages	Add the link to the project using the dataset.	- dataset_name: Name of the dataset to annotate. - used_in: the link to the project. - show_query: Print SPARQL query.
add_table_usages	Add the link to the project using the table.	- dataset_name: Name of the dataset the table belongs to. - table_name: Name of the table to annotate. - used_in: the link to the project. - show_query: Print SPARQL query.
get_dataset_usages	List the projects that use the dataset.	- dataset_name: Name of the dataset to annotate. - show_query: Print SPARQL query.
get_table_usages	List the projects that use the table.	- dataset_name: Name of the dataset the table belongs to. - table_name: Name of the table to annotate. - show_query: Print SPARQL query.
execute_sparql_query	Execute an ad-hoc query.	- query: SPARQL query - method: Type of the query (Get, Post)
get_joinable_columns_between	Get all the paths between a starting table and the target one.	- table1: Pandas dataframe representing the starting table. - table2: Pandas dataframe representing the target table.
get_joinable_columns	Given data, find joinable columns from the data GLac captured. The input data can be new data or already in GLac.	data: Pandas series or dataframe to find joinable columns for.
get_unionable_columns	Given two tables, find unionable columns based on column name semantic affinity.	- df1: Pandas dataframe to represent the first table. - df2: Pandas dataframe to represent the second table. - threshold: semantic affinity threshold to consider two column name unionable.

In conclusion, Table 4.4 summarises the various APIs the interface services component supports. Using these APIs, the user can perform data discovery and data enrichment operations. Section 5.1.2 presents a use case showcasing how to use them to compile a dataset from scratch. Hence, KGLac offers numerous APIs that operate on top of the GLac, our schema KG for data lakes. As a result, the user can explore data lakes via functionalities that leverage the semantic and content of the data they contain. Implementing the system could not be possible without using certain technologies. The subsequent chapter will cover the reason behind using these technologies.

4.2 KGLac characteristics

KGLac instrumentalizes several technologies to build the GLac and allow the user to interact with it. These technologies promote four features of the platform: scalability, reliability, comprehensibility, and reusability. This section will list the different technologies and show how they contribute to these various characteristics.

4.2.1 Apache Spark

First, KGLac has a scalable profiler thanks to using PySpark. The profiler has to process the different datasets data lakes contain. So for KGLac to handle the data, it requires adequate tools. Hence the profiler leverage PySpark capabilities to address the scalability aspect. PySpark is a python library supporting Apache Spark, an engine for large-scale data processing. Apache Spark outperforms several state-of-art tools like Hadoop MapReduce to deal with big data [58]. It introduces the resilient distributed dataset (RDD), a collection of objects distributed across different machines, to process data in parallel. Hence, Apache Spark supports parallel computing by executing its two operations (transformations and actions) on each partition [59]. Additionally, Apache Spark reports the execution of the various transformations such as map, flatMap, and union, until an action like take, collect, and count is triggered. This Lazy Evaluation contributes to making KGLac scalable: It reduces communication between the different workers and the driver. In addition, it optimizes these operations thanks to its Catalyst Optmizer[12] when using the dataframes.

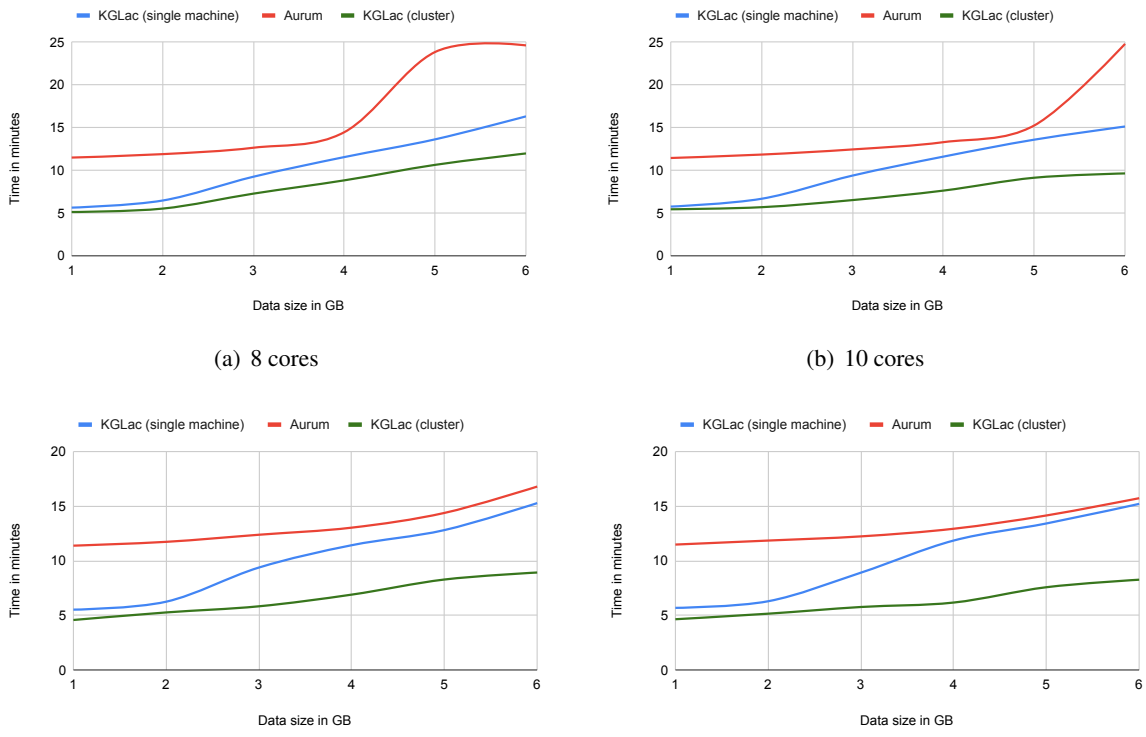
[12]https://databricks.com/glossary/catalyst-optimizer

Furthermore, Apache Spark reduces the execution time by adopting in-memory data processing. This feature allows avoiding fetching data from the disk, which is time-consuming. Thus, thanks to these characteristics, KGLac is scalable. Moreover, the profiler leverage Apache Spark dataframe to interpret the data type of the different columns of the tabular data. In fact, by using the native APIs of Apache Spark, after loading the data, by passing the inferSchema argument, PySpark can infer the data types of the columns. Such an operation will require going over the data, which takes time if the table is large. However, parsing the content of the columns is necessary to obtain the data type. In this case, it is better to re-use what PySpark supports than creating our own. In conclusion, using Apache Spark serves two objectives: scalability as it is adequate to process voluminous data, and reliability as it supports determining the data type, crucial information to profile the columns. To store the profiles, KGLac uses Elasticsearch. The next chapter will present the reasons behind choosing it.

To evaluate the performance of processing the data and generating the profiles, we compare KGLac and Aurum profilers. The setting of the experiments consists of evaluating both KGLac and Aurum on a single machine with 16 cores. In addition, the evaluation includes running KGLac on a cluster of 4 nodes, each with 16 cores. It is worth mentioning that, unlike KGLac, Aurum was not assessed on a cluster of machines because it does not support it. Furthermore, since there is a difference in what metrics each profile extracts, the evaluation merely considers the common ones. In this case, KGLac does not extract the number of missing values in each column. Furthermore, on the cluster of machines, the system dumps the data in a text file instead due to technical difficulties, Compute Canada [13] support reported. The assessment includes four sets of experiments, each for a different number of threads $\lambda \in [8, 10, 12, 14]]$. Figure 4.6 shows evolution of the running time in minutes of each system and for each λ for different dataset sizes ($\eta \in [1, 2, 3, 4, 5, 6]$), extracted from online portals. First, for Aurum, the runtime monotonically increases, for every λ as η grows. For $\lambda \in [12, 14]$ the runtime starts at $\tilde{1}1$ minutes to reach $\tilde{1}5$ minutes. However, for the remaining λ values, the observation shows the existence of two phases: The runtime starts with 11 minutes, continues to steadily increase until reaching $\eta\prime$ where it starts to drastically grow, attaining 25 minutes. $\eta\prime$ varies based δ ($\eta\prime$ = 4 and 5 for λ = 8 and 10, respectively). Second, KGLac (single

[13] https://www.computecanada.ca/

machine) outperforms Aurum. The four graphs show a monotonically increasing runtime starting with 5 minutes at $\eta = 1$ to reach approximately 15 minutes at $\eta = 6$. Third, for KGLac (cluster), it outperforms both Aurum and KGLac (single machine). The runtime starts at 5 minutes to end with 9 minutes on average. Hence, these graphs show that KGLac outperforms Aurum on a single machine, and the performance becomes greater if KGLac runs on a cluster thanks to using Apache Spark.



(a) 8 cores

(b) 10 cores

(c) 12 cores

(d) 14 cores

Figure 4.6: KGLac and Aurum profilers scalability.

4.2.2 Elasticsearch

KGLac also uses Elasticsearch, a document database, to store the different profiles and raw data. Elasticsearch is an integral part of KGLac because it ensures scalability on two different levels: Storage and search. Elasticsearch stores data in clusters and each one of them is a set of nodes.

Elasticsearch is a distributed store because it accommodates adding and removing nodes and automatically redistributes the data [14]. The data consists of documents stored under an index, a logical grouping composed of several shards. A shard is a self-contained index distributed across the different nodes. Hence, Elasticsearch spread the documents into the multiple shards hosted on the various nodes. This storage model permits housing large data, including the profiles and the raw data, the profiler component generates. In addition, Elasticsearch offers scalable search thanks to having two types of shards, primary and replica. Although these characteristics lead to data redundancy, they ensure fault-tolerance and increase data query capacity [17]. As a result, Elasticsearch simultaneously offers scalable storage allowing the profiler to load documents computed from data lakes and scalable data search allowing the GLac Builder to query them to build GLac. GLac is an RDF KG that captures predicate annotations by the means of RDF-star. The next section will explain why this technology is adequate to model GLac.

4.2.3 RDF-star

In addition, the GLac Builder generates an RDF KG to represent the schema for the content of data lakes. In fact, to capture statement-level annotation, the build harness RDF-star as a representation to tag the edges. RDF-star is an extension to RDF to embed descriptions to the predicates. For instance, GLac captures the certainty that relationships exist between two columns. RDF-star is an alternative to reification techniques to model these annotations. Yet, These approaches present shortcomings and W3C has not adopted any of them as a standard [23]. The limitations lie within an increase in the number of statements to capture this information and the cumbersome query a user needs to write. Ontotext[15] lists these different approaches coupled with the advantages and drawbacks they offer. They also compared the performance of each of them with RDF-star using a benchmark subset of Wikidata. Table 4.5 summarizes the results and shows that RDF-star represents the same data with 50 million fewer statements than the best of the investigated reification methods. In addition, it shows that even to load data, RDF-star outperforms the others by taking just two-thirds of the time of the best one. Plus, to permit querying RDF-star, one can use SPARQL-star,

[14] https://www.elastic.co/guide/en/elasticsearch/reference/master/add-elasticsearch-nodes.html
[15] https://www.ontotext.com/knowledgehub/fundamentals/what-is-rdf-star/

Table 4.5: Performance comparison between RDF-star and the other approaches

Modeling Approach	Total Statements	Loading time (min)
Standard reification	391,652,270	52.4
N-ary relations	334,571,877	50.6
Named graphs	277,478,521	56
RDF-star	220,375,702	34

an extension to SPARQL, to seamlessly access these annotations. Hence, RDF-star supports a more compact representation promoting, as a result, GLac comprehensibility. Next, we will explain why this technology is adequate to model our KG.

4.2.4 Pandas

Last, the interface services leverage Pandas dataframes to report the results. Pandas is a commonly-used tool in data science to manipulate tabular data. Hence, the user does not need to get accustomed to using a new data structure to explore the result of the APIs. The user can use the native Pandas functionalities to operate on the results. For instance, if the user is concerned about data completeness, they can sort the result by the number of missing values. Or, if they want to find large tables they can filter based on the total number of values. Hence, by instrumentalizing Pandas, the interface service provides a user-friendly interface promoting reusability. In conclusion, this section lists the different characteristics of the tools used to implement a scalable, reliable system that produces a GLac, comprehensible KG using Lac whose schema and vocabulary is available in Appendix A and Appendix B, respectively. In addition, one can interact with GLac through a user-friendly interface. The next chapter will present several use cases where we ran our platform to perform data discovery and data enrichment.

Chapter 5

Use Case & Evaluation

The last chapter covered KGLac different components and the pipeline on how to process raw data, create GLac, and allow the user to interact with it. The user will merely need to call the various APIs to perform data discovery and data enrichment. Hence, this chapter will present a use case where KGLac could be seamlessly integrated into the data science pipelines to build a dataset to investigate the gender-based pay gap in the Information systems field. Subsequently, the chapter will evaluate the performance of the KGLac in determining predicates between the nodes and compare it with Aurum, a state-of-art system using the same dataset.

5.1 Use Case

This section presents a use case where KGLac is integrated into the data science pipeline to build data, which a data scientist can use to explore the compensation gap in the IS domain based on gender. In this scenario, we assume that a data scientist possesses a locally stored data lake and uses KGLac to process its content to build GLac. Additionally, KGLac will host the KG on Blazegraph [1] and the profiles and raw data on Elasticsearch. Then, the user creates a Jupyter notebook where they will call the interface services to compile their data. Plus, we assume that the data the user aims to form is a dataframe comprising columns about gender and salaries. But before going over the end-to-end scenario, the section will provide details about the data lake KGLac processed.

[1] https://blazegraph.com/

5.1.1 Data Lake

Before leveraging the different APIs to explore the knowledge graph and build the data, the user will need to specify the data lake KGLac will process. This scenario is consists of over 125 real datasets [2] containing more than 380 tables and consisting of more than 6500 columns with a 2 GB in size. The different datasets were collected from several sources, including but not limited to Kaggle[3] and open government portals. The datasets cover several topics like COVID-19, sports, crimes, employment, and real estate. This data lake serves two objectives: First, building GLac to investigate the pay gap between genders. Second, comparing the performance of KGLac with Aurum in terms of inferring relationships (semantic similarity, content similarity, and PKFKs). The generated GLac is composed of over 3.5 million statements.

5.1.2 Scenario

This chapter will cover the full pipeline a data scientist goes through to build their data using the different APIs supported by KGLac. First, the user starts the interface services by specifying Blazegraph endpoint and Elasticsarch to query GLac and the raw data, respectively. Second, the user will start looking for data related to salaries. In this case, they can use the *search_table_on* API to discover tables given a set of conditions. KGLac will match the different conditions with the names of the columns using regular expressions. In this use case, the user is interested in searching for tables with column names having the salary, pay, or earning tokens. The result is a pandas dataframe with 21 rows. Each represents a table meeting the specified conditions coupled with its dataset and the number of columns and rows. However, the user is keen on starting with a large table with loads of rows. Thus, they leverage pandas structure to sort the values based on the number of rows. Table 5.1 shows the first five rows where the first row corresponds to the table 'employee_salaries.csv' with approximately 150 000 rows. The second-largest table contains only about 40 000. With these figures, the user chooses to load the first table. Figure 5.1 shows the APIs calls, a user would perform to get get the sorted dataframe.

[2] https://github.com/CoDS-GCS/KGLac_materials
[3] https://www.kaggle.com/datasets

```
salaries_tables = KGLac.search_tables_on([['salary', 'pay', 'earning']])
salaries_tables.sort_values('number_of_rows', inplace=True, ignore_index=True, ascending=False)
```

Figure 5.1: Search for tables with columns having salary, pay, or earning using KGLac search_tables_on API. Then sort the results in descending order based on the number of rows.

Table 5.1: The first five rows of the salaries_tables with the highest number of rows.

	table_name	dataset_name	number_of_columns	number_of_rows	path
1	employee_salaries.csv	san francisco county	8.0	148654.0	/data/us_counties_cities/san francisco county/ employee_salaries.csv
2	san-francisco-2019.csv	san-francisco-salaries19	13.0	41136.0	/data/transparent_california/ san-francisco-salaries19/san-francisco-2019.csv
3	san-francisco-2017.csv	san-francisco-salaries19	12.0	41127.0	/data/transparent_california/ san-francisco-salaries19/san-francisco-2017.csv
4	san-francisco-2018.csv	san-francisco-salaries19	13.0	35212.0	/data/transparent_california/ san-francisco-salaries19/san-francisco-2018.csv
5	LA_County_Employee _Salaries.csv	la_county	20.0	35196	/data/us_counties_cities/la_county/ LA_County_Employee_Salaries.csv

Table 5.2: The first five records in employee_salaries.csv

	EmployeeID	JobTitle	BasePay	OvertimePay	OtherPay	Benefits	TotalPay	TotalPayBenefits
1	E2537827072	GENERAL MANAGER-METROPOLITAN TRANSIT AUTHORITY	167411.18	0.0	400184.25	NaN	567595.43	567595.43
2	E8426241129	CAPTAIN III (POLICE DEPARTMENT)	155966.02	245131.88	137811.38	NaN	538909.28	538909.28
3	E5931786714	CAPTAIN III (POLICE DEPARTMENT)	212739.13	106088.18	16452.6	NaN	335279.91	335279.91
4	E2661881086	WIRE ROPE CABLE MAINTENANCE MECHANIC	77916	56120.71	198306.9	NaN	332343.61	332343.61
5	E3902371237	DEPUTY CHIEF OF DEPARTMENT, (FIRE DEPARTMENT)	134401.6	9737.0	182234.59	NaN	326373.19	326373.19

After finding data related to salaries, the user loads the table into a pandas dataframe using the path pointing to its location in the data lake. Table 5.2 shows the first five rows of the table. It contains several attributes like the ids, job titles, and the employees' salaries. However, it misses sex, a crucial component for investigating the gap. Hence, in the next step, the user is interested in finding data containing information about gender. To this end, using the same API to find data with salaries, they specify 'gender' and 'sex' as the conditions. Figure 5.2 shows the different calls the user made to find the table. After sorting, the user loads the fourth table having more than 100 000 records, the closest to the salaries table in terms of the number of rows. Table 5.3 shows that each row reports the name of the person, their gender, and the name count for that gender. Now, the user has two tables, one for the employees' salaries in San Francisco county and the other for the gender for each name. However, there are two issues. First, for some names, the gender can map to either

Table 5.3: The first five records in names_gender.csv

	table_name	dataset_name	origin	number_of_columns	number_of_rows	path
1	SeoulFloating.csv	kimjihoo_coronavirusdataset	kaggle	7.0	1084800.0	/data/kaggle/kimjihoo_coronavirusdataset/SeoulFloating.csv
2	database.csv	nhtsa_safety-recalls	kaggle	39.0	638454.0	/data/kaggle/nhtsa_safety-recalls/database.csv
3	covid.csv	tanmoyx_covid19-patient-precondition-dataset	kaggle	21.0	566602.0	/data/kaggle/tanmoyx_covid19-patient-precondition-dataset/covid.csv
4	names_gender.csv	names_gender	data_world	3.0	109173.0	/data/data_world/names_gender/names_gender.csv
5	population.csv	xvivancos_barcelona-data-sets	kaggle	8.0	70080.0	/data/kaggle/xvivancos_barcelona-data-sets/population.csv

male or female. For instance, Aadaya is a name for both genders with 4828 males and 8 females. Second, the user cannot concatenate the two tables because there is no common column to append one to the other. To mitigate the first problem, we assume one name only corresponds to one gender based on the highest count. For Aadaya, the user will consider it a name for a male. Second, the user will further explore the data lake to discover a table that apports information about the names of the employees.

```
gender_tables = KGLac.search_tables_on([['gender', 'sex']])
gender_tables.sort_values('number_of_rows', inplace=True, ignore_index=True, ascending=False)
```

Figure 5.2: Search for tables with columns having gender or sex using KGLac search_tables_on API. Then sort the results in descending order based on the number of rows.

The user at this stage calls the API *get_path_between_table* and specifies the salaries table as a starting point and the gender one as the target with two hops, as shown in Figure 5.3. The API returns a graph depicting the different paths between the tables. Figure 5.4 shows a snapshot of GLac where the starting table, dataset, columns are colored in blue while the target ones are in orange. In the graph, there are two intermediate tables, employee_info.csv, and san-francisco-2019.csv. The user is interested in concatenating both tables by finding a common column. The figure reveals that the former belongs to the same dataset as the salaries tables, indicating that the tables are related. Moreover, the column EmployeeID uniquely identifies the employee in both tables. So the user can merge the tables on that column to apport the first name for each employee. Later, by using this new information, the data scientist can determine the gender of the employee via concatenating the two tables on the firstName and name columns. Hence, the employee_info table is suitable to compile a dataframe having the salary and gender information. Figure 5.5 shows the different steps used to

obtain the resultant data. The user first loads the employee-info table and then merges it with the salaries one on the EmployeeID column. Next, the user concatenates the result with the gender table to have a dataframe containing more than 140 000 rows. Table 5.4 shows 5 rows of the result. For each employee, the data scientist knows their name, salary, and gender.

```
KGLac.get_path_between_tables(salaries_tables.iloc[0],gender_tables.iloc[3], hops=2)
```

Figure 5.3: Get the paths between the salary table and the gender one by calling KGLac get_path_tables API.

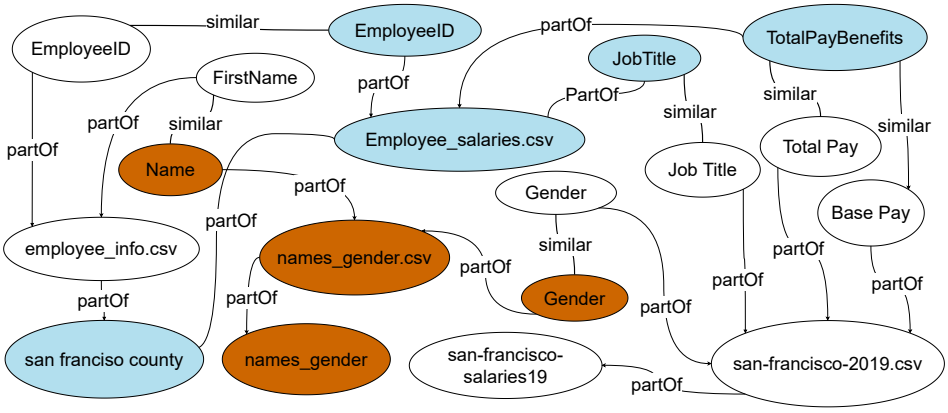


Figure 5.4: There are two paths between the selected tables.

Table 5.4: The first five records in salaries with gender table

	EmployeeID	JobTitle	BasePay	OvertimePay	OtherPay	Benefits	TotalPay	TotalPayBenefits	FirstName	Gender
0	E2537827072	GENERAL MANAGER-METROPOLITAN TRANSIT AUTHORITY	167411.18	0.0	400184.25	NaN	567595.43	567595.43	christianne	F
1	E2922245454	IS BUSINESS ANALYST - SENIOR	100039.21	0.0	0.0	NaN	100039.21	100039.21	christianne	F
2	E2602716442	Management Assistant	71945.8	0.0	0.0	29594.18	71945.80	101539.98	christianne	F
3	E8426241129	CAPTAIN III (POLICE DEPARTMENT)	155966.02	245131.88	137811.38	NaN	538909.28	538909.28	otoniel	M
4	E1685310364	EMPLOYMENT AND TRAINING SPECIALIST III	60326.15	0.0	525.58	NaN	60851.73	60851.73	otoniel	M

Furthermore, the user is eager to augment the data by adding more rows. To do so, they load 'san-francisco-2019.csv' the second largest as shown in Table 5.1. The table contains over 40 000 instances. The table has many attributes, including the full name of the employee, their sex, their

```
# read employee_info.csv
employee_info_table = KGLac.search_table_by_name('employee_info.csv')
employee_info = pd.read_csv(employee_info_table.path[0])

# Join salaries_kaggle and employee_info on id
salaries_with_name = salaries_kaggle.merge(employee_info[['EmployeeID', 'FirstName']],
                                left_on = 'EmployeeID', right_on = 'EmployeeID')

# change FirstName to lower case before joining
salaries_with_name['FirstName'] = salaries_with_name['FirstName'].str.lower()

# change name to lower case and keep duplicated names with higher counts, if any.
gender_by_name['Name'] = gender_by_name['Name'].str.lower()
gender_by_name = gender_by_name.sort_values('Count').drop_duplicates('Name')

# add gender to the salaries dataframe
salaries_with_gender = salaries_with_name.merge(gender_by_name[['Name', 'Gender']], left_on='FirstName',
right_on='Name').drop('Name', axis=1)
```

Figure 5.5: Merge the salary table with the gender one using the employee_info table.

job title, and information about their salaries. In other words, the table contains core information about investigating the compensation gap between the genders. However, to append the two tables, the user needs to pick unionable columns. This task is cumbersome because the user needs to manually compare all the different column pairs of both tables. In fact, the arities of the table are ten and thirteen, resulting in a total of 130 comparisons. Hence, to mitigate this limitation, the user intrumentalize KGLac, by calling *get_unionable_column* and passing the 'salaries with gender table' along with 'salaries in san francisco' one as shown in Figure 5.6. KGLac will leverage embedding to find suitable columns to union on. The result is a pandas dataframe with each row containing a pair of columns the user can use to union the two tables. This API helps the user to reduce the number of pairs to check to determine the unionable columns. In this case, the result return 12 results, as Table 5.5 shows. Hence reducing the number to less than 10% of the number of combinations they would have investigated without KGLac. To union the two tables, the user only selects the 'Job Title', 'Gender' and 'Total Pay' columns. They assume that at this stage, these are the features relevant to investigating the pay gap. After the union (Figure 5.7 shows the steps), the new dataframe consists of more than 180 000 rows and three columns. The first five rows are shown in Table 5.6.

```
KGLac.get_unionable_columns(salaries_with_gender, salaries_san_francisco)
```

Figure 5.6: Get unionable columns between the salary_with_gender dataframe and the salaries_san_francisco one using KGLac get_uninable_columns.

Table 5.5: Unionable columns between the salaries_with_gender dataframe and the salaries san Francisco one.

	First dataframe columns	Second dataframe columns
0	JobTitle	Job Title
1	BasePay	Base Pay
2	OvertimePay	Overtime Pay
3	OtherPay	Other Pay
4	Benefits	Benefits
5	Benefits	Total Pay & Benefits
6	TotalPay	Total Pay
7	TotalPay	Total Pay & Benefits
8	TotalPayBenefits	Benefits
9	TotalPayBenefits	Total Pay
10	TotalPayBenefits	Total Pay & Benefits
11	Gender	Sex

```
# select only name, job title, year, and total pay columns
salaries_with_gender = salaries_with_gender[
   ['JobTitle', 'Gender', 'TotalPay']
]
salaries_san_francisco = salaries_san_francisco[
   ['Job Title', 'Sex', 'Total Pay']
]
# rename the columns from TransparentCalifornia
salaries_san_francisco.columns = ['JobTitle', 'Gender', 'TotalPay']
salaries = pd.concat([salaries_with_gender, salaries_san_francisco])
salaries
```

Figure 5.7: Steps to union the salaries_with_gender and the salaries_san_francisco dataframe.

The data scientist can use the resultant dataframe to investigate the pay gap between the gender. However, they are merely interested in exploring it in the domain of IS. For this reason, they also included the job title column to union the two dataframe in the previous step. Yet, to only include

Table 5.6: The first five records in the salaries table after the union

	JobTitle	Gender	TotalPay
0	GENERAL MANAGER-METROPOLITAN TRANSIT AUTHORITY	F	567595.43
1	IS BUSINESS ANALYST - SENIOR	F	100039.21
2	Management Assistant	F	71945.80
3	CAPTAIN III (POLICE DEPARTMENT)	M	538909.28
4	EMPLOYMENT AND TRAINING SPECIALIST III	M	60851.73

the rows corresponding to individuals working in IS, the user will need to filter the content of the dataframe. Yet, to do so, the user will need to do so based on the job title. In other words, the user will keep the rows whose title contains one of the keywords the user needs to set. Nevertheless, the user has to set an exhaustive list which is time-consuming. As an alternative, the user can search the data lake for tables having information, helpful to only keep the IS-related jobs. Plus, the table the user has to find should have a column having similar content to the Job Title column obtained in the last step. For this, KGLac equips the user with *get_joinable_columns* for which they specify the column to join on. KGLac generates column embedding for the passed input, retrieves matching columns, and lastly returns them to the user in the dataframe. Figure 5.8 and Table 5.7 show the API call coupled with the returned results with selected attributes, respectively. It shows that there are five matching columns in five different tables. The user is already using the last two. The second and the third rows should be similar to the san-francisco-2019. The only difference is that they contain data for 2018 and 2017. So the remaining table to explore is the first one. Table 5.8 shows the loaded table containing several columns. But what is interesting for the user is the existence of a Job Class for each title effective in 2017-18. This information is insightful as job titles belonging to the same domain would have the same class. In this case, the user merges the two tables based on the job title and filter the rows using the class 'Information System'. As a result, the data scientist has curated data potentially used to assess the compensation gap in the Information system domain. The data scientist avoids spending considerable time thanks to KGLac and the various APIs to perform data discovery and data enrichment. In the end, the user can add comments about the tables they used to compile the data. Such information can be an insightful reference for other users in the future.

```
columns = KGLac.get_joinable_columns(salaries['JobTitle'])
```

Figure 5.8: Find joinable columns with the JobTitle column belonging to the salaries dataframe in the data lake.

Table 5.7: Joinable columns between with JobTitle in the salaries dataframe.

	original_column	column_name	table_name	dataset_name	...	number_of_distinct_values	cardinality	column_data_type
0	JobTitle	Title	Hourly-Rates-of-Pay-by-Classification-and-Step-FY17-18.csv	classification		961.0	0.606692	T
1	JobTitle	Job Title	san-francisco-2018.csv	san-francisco-salaries19		975.0	0.027689	T
2	JobTitle	Job Title	san-francisco-2017.csv	san-francisco-salaries19		1016.0	0.024704	T
3	JobTitle	Job Title	san-francisco-2019.csv	san-francisco-salaries19		1049.0	0.025501	T
4	JobTitle	JobTitle	employee_salaries.csv	san francisco county		2159.0		

Table 5.8: First five records in Hourly-Rates-of-Pay-by-Classification-and-Step-FY17-18.csv

	Eff. Date	Job Code	Job Class	Title	Grade	Biweekly Min	Biweekly Max	Extended Range	Step 11	...	Step 20
0	7/1/2017	140	Management	Chief, Fire Department	0140N	$12,208	$12,208	NaN	NaN		NaN
1	7/1/2017	140	Management	Chief, Fire Department	0140N	$12,208	$12,208	NaN	NaN		NaN
2	7/1/2017	381	Police Services	Inspector 2	0381N	$5,476	$5,476	NaN	NaN		NaN
3	7/1/2017	387	Police Services	Crime Scene Invstgtn Mngr 3	0387N	$6,134	$6,134	NaN	NaN		NaN
4	7/1/2017	390	Management	Chief of Police	0390N	$12,426	$12,426	NaN	NaN		NaN

This section covered a use case where a data scientist is interested in investigating the pay gap between gender in the Information Systems field. Throughout the use case, the user leverages the interface service component to interact with KGLac. They called several data discovery and enrichment APIs to compile a dataset containing records about gender, salary, and job title for several employees. In addition, the use case shows how easy and seamless to integrate KGLac in a data science pipeline. Moreover, it shows how KGLac saves the user time as it reports different attributes related to tables, such as the number of rows and the names of the columns. Hence, the user does not need to open the various files to verify such information. Plus, the user does not need to compare columns to assess if they contain the same content or belong to the same domain. KGLac supports all of these operations in a rapid and user-friendly manner. However, several of these functionalities depend on how well the GLac builder infers the relationships. In the section, we use the same dataset to evaluate KGLac and compare its performance with Aurum's [12], a state-of-the-art data discovery system. The evaluations show that KGLac comparatively results in promising figures in detecting the relationships.

5.2 Evaluation

This section reports the evaluation of KGLac using a subset of the previous dataset described in 5.1.1. In fact, there is ground truth for the data lake that includes all the different relationships. So, we need to come up with our own. However, given the size of the data lake, it is not feasible to manually generate it. Hence, the experiments are limited to the datasets used in the use case, consisting of 4 datasets, 9 tables, and 110 columns. It consists of assessing how performance the KG builder is in establishing relationships between the different columns in GLac. The concerned relationships are semantic similarity, content similarity, and PKFKs. The evaluation did not take into consideration the inclusion dependency one because KGLac does not explicitly support a functionality based on it. It instead uses it to create PKFKs. Hence, the evaluation of PKFK would simplicity provide insights about the performance of creating these predicates. In addition, the section will compare the performance of KGLac with Aurum, a known system to perform data discovery for each relationship.

5.2.1 Semantic Similarity

Before evaluating KGLac and comparing its performance, it is crucial to generate the ground truth for semantic similarity relationships. First, we generated all the possible column pairs resulting in 11556 pairs. However, if column A has a semantic similarity with column B, then B has the same relationship with A. With this observation, the number of pairs can be halved, reaching 5778 ones to manually check. For this reason, after generating the pair combinations, we only need to double the number of relationships. After generating the ground truth, we evaluated KGLac on four different thresholds: 0.2, 0.4, 0.6, 0.8. In fact, there is a semantic similarity between two columns if their semantic similarity is equal or greater than the threshold. Then, based on the results, we retrieve an optimal threshold using which we compare KGLac with Aurum with its default parameters.

Figure 5.9 depicts the variation of KGLac recall, precision, and F1 score with respect to the thresholds. It shows that the recall has a decreasing trend starting at value 1 and dropping to 0.7. However, both the precision and the f1 score increase by starting at approximately 0 for the threshold 0.2 and going above 0.75, for the 0.8 threshold. In other words, the system recovered all the semantic

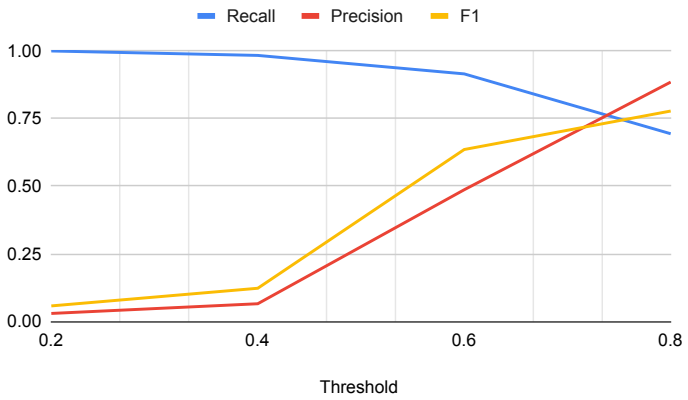


Figure 5.9: Performance of KGLac in determing semantic similarity relationships.

similarity relationships. For the same threshold, the precision and f1 score are very low, almost 0. These figures indicate that KGLac has no false negatives at the cost of having many false positives. In other words, the low threshold enabled KGLac to establish relationships between columns that are not semantically related. The graph can be divided into two intervals [0.2, 0.4] and [0.4, 0.8] based on the trend evolution of the graphs. Between 0.2 and 0.4, the recall slightly decreases and drops from 1 to 0.98. However, for the precision and f1 score, moderately increase to reach almost 0.1. This observation shows that the threshold is still loose as KGLac reports many false positives. Between 0.4 and 0.8: on one hand, the recall starts to decrease at a higher rate, dropping from 0.98 to 0.7. On the other hand, the precision and f1 score increase at a higher rate to go from roughly 0.1 to 0.9 and 0.76, respectively. In fact, increasing the threshold leads to pruning a big portion of false positives at the cost of having a few false negatives. In other words, KGLac did not report columns with a low affinity. Hence, the f1 measure shows that with 0.8 threshold, KGLac still has a promising performance as it balances between discovering a lot of true positives without having a lot of true negatives. Besides, based on the f1 measure, KGLac is most performant when the threshold is 0.8.

Next, let us compare the performance of KGLac with Aurum's. We set the threshold to be equal to 0.8 for the former and keep the default parameters for Aurum. Figure 5.10 shows chart showing the recall, precision, and f1 measure for both systems. Fo the three metrics, KGLac outperforms Aurum. Based on the recall, Aurum misses more than two-thirds of the total number of pairs, while KGLac misses less than one-third. As for the precision, both systems have comparable results.

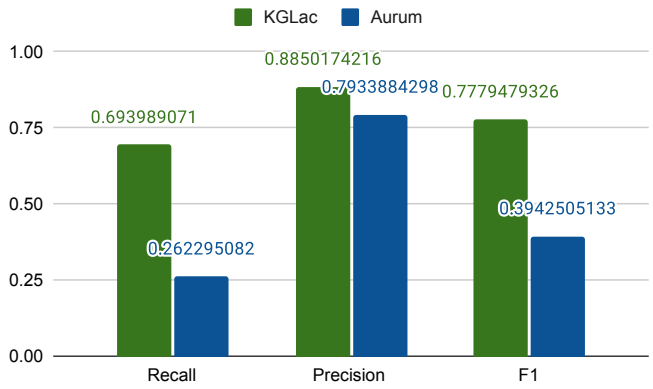


Figure 5.10: Evaluation comparision between KGLac (threshold =0.8) and Aurum (default parameters)

Yet, KGLac is slightly better as it reports fewer false positives than Aurum. As for the F1 score, a harmonic mean of the call and precision, shows that KGLac performs better than Aurum since it has a relatively higher recall. In fact, the recall is low for Aurum because it calculates the similarity between two columns by computing the cosine distance between TF-IDF [4] vectors and considering each column name to be a document. This approach does not take into consideration the context of the word and merely calculates its frequency. For example, it does not capture a semantic similarity between (total earnings, total compensation). In addition, Aurum does not clean the data as it misses capturing (JobTitle, Job Title). It merely feeds the documents as extracted from the documents to the model without parsing. However, KGLac normalizes the column names by creating labels of the column names. A label is a cleaner version of names where both 'JobTitle' and 'Job Title' correspond to the single label 'job table'. In addition, it takes into consideration the context of each token of the label by relying on word embeddings. Hence, KGLac outperforms Aurum in determining semantic relationships between columns.

5.2.2 Content Similarity

Similar to semantic similarity, before evaluating KGLac, we need to come up with ground truth. For these relationships, we also limit the experiment to the same subset of datasets. Determining the ground truth for content similarity happens in two phases after generating all the pairs. First,

[4] https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html

we manually check if the variables stem from the same variable. For instance, the 'FirstName' of the employee_info.csv and the 'Name' in the name_gender.csv is a valid tuple. However, 'BasePay' and 'TotalPay' in Employee_slaries.csv is not. Second, we compute the overlap between the kept ones. Determining the overlap depends on the type of the columns. For textual, each couple is associated with the exact Jaccard similarity. As for the numerical, the overlap between the ranges of the columns reflects the content similarity. For both types, we filter out the pairs having an overlap strictly less than 0.5. After the two-step approach to determining the ground truth, the final result consists of 507 pairs, 25 textual and 482 numerical. To evaluate KGLac and compare it to Aurum, we set 5 thresholds α, st. $\alpha \in$ (0.5, 0.6, 0.7, 0.8, 0.9) for the overlap related to the ground truth. For each, we set 4 thresholds β s.t $\beta \in$ (0.2. 0.4. 0.6, 0.8) associated with the overlap for Aurum and KGLac to create a content similarity relationship. Hence, the evaluation consists of 20 experiments summarized in Figure 5.11.

The different graphs represent a comparative evaluation between KGLac and Aurum for the different thresholds for the ground truth (top to bottom) and the different metrics: Recall, Precision, and F1-measure (Left to right). For the recall, both systems have the same performance regardless of the values of α and β. All the recall graphs show that they start at a recall equal to 1 for β = 0.2 and reach to 0.5 for β = 0.8. As for the precision, the systems have comparable performances. For all values of α, KGlac outperforms Aurum with roughly 0.025 difference on average. The majority of the graphs have the same allure where the performances increase as β increases. They all start at values belonging to the interval [0.04, 0.06] for Aurum and [0.06., 0.09] for KGLac at β = 0.2. The values reach [0.1, 0.0.85] and [0.11, 0.125] respectively for β = 0.8. Last, for the F1 score, KGLac also slightly outperforms Aurum, with a difference equal to 0.3 on average. Similarly, the graphs have the same allure regardless of the value of α. However, the graphs increase as β but reach a peak at β=0.6 and then decrease. For Aurum, in all the graphs, the F1 score starts at 0.1 except for α = 0.9, with a value is equal to 0.8. Then keep increasing to reach the peak. The highest values range from 0.13 and 0.18 with highest for α = 0.6 and the lowest for α = 0.9. The F1 score drops to 0.15 for most of the graphs. For KGLac, the F1 score starts between 0.1 and 0.15. The score keeps increasing until attaining the peak that belongs to the [0.14, 0.2] range. The highest peak is for α = 0.6 and the lowest corresponds to α = 0.9. Then the values drop to 0.8 for

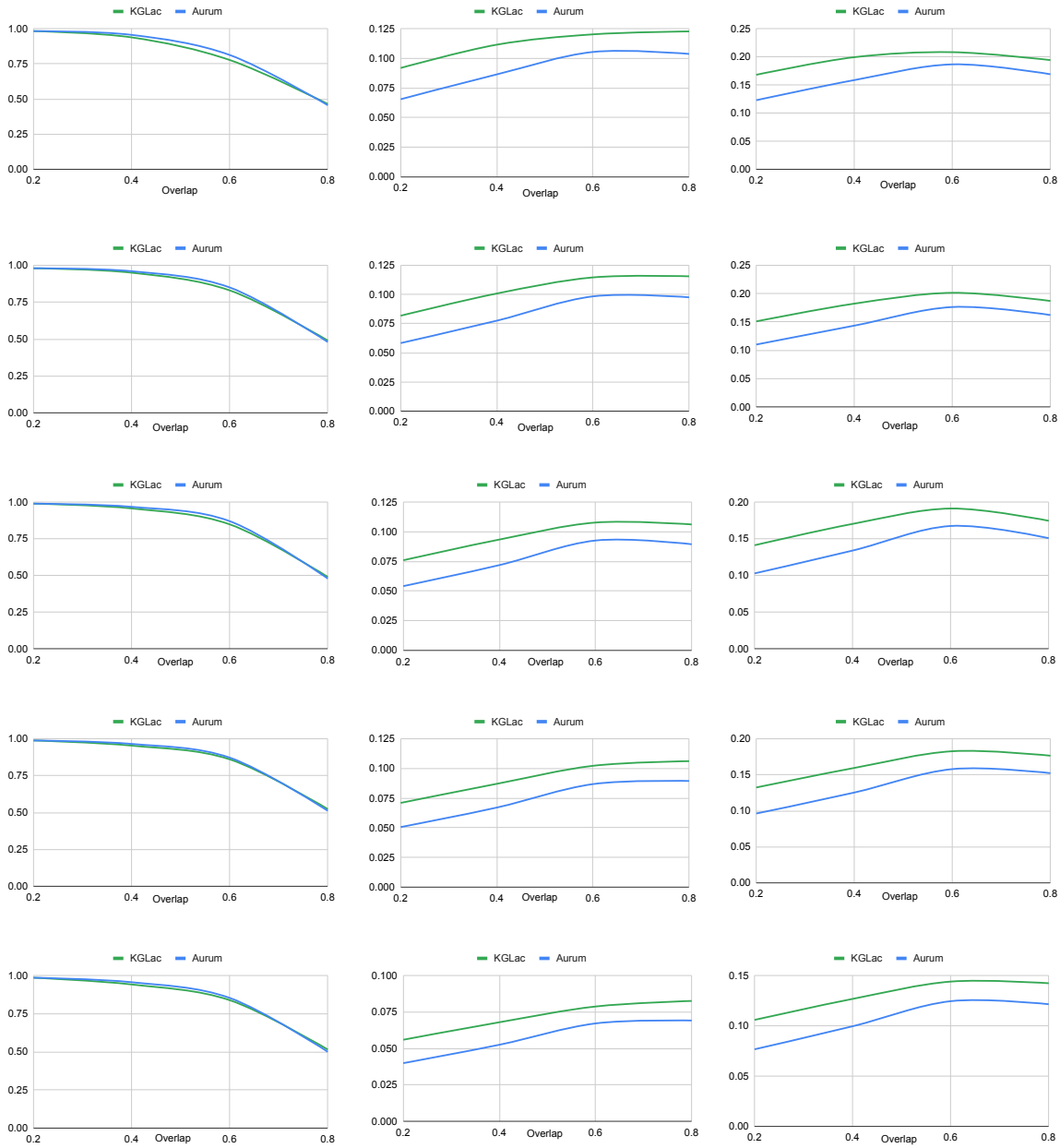


Figure 5.11: Each row corresponds to α value starting from 0.5 and ending with 0.9 (top to bottom). Each column corresponds to an evaluation metric: Recall, Precision, F1-score (left to right)

most of the graphs. In fact, despite adopting the same approach to establish the content similarity for both types, there is a slight difference between the systems. This difference lies within the profiler. For numerical values, Aurum approximately calculates the quartiles with a relative error equal to 0.15, using the QDigest class [5] with 128 as compression ratio. Yet, KGLac limits the relative error to 0.03 using Spark [6]. As for the textual columns, both systems perform similarly in terms of capturing and missing relationships. Hence, both systems compromise exactitude with a performance by approximately calculating the quartiles. So the different relative errors explain the difference in precision, impacting the difference in F1 measure. In conclusion, regardless of α both KGLac and Aurum are performance for β = 0.6.

5.2.3 PKFKs

To evaluate the KGLac performance in detecting PKFKs and compare it with Aurum, we need the ground truth. Similar to the other relationships, we generated it using the same datasets in the use case. To determine where there is a PKFK between two columns, they need to meet two conditions. First, all the records in the columns are unique. In other words, the ratio of the number of distinct values and the total number of values is equal to 1. Second, all the records of the foreign key constitute a subset of those in the primary one. First, for each column type, we create pairs p=(potential primary column, potential foreign column) using all the columns in the dataset. Then we filter them by keeping only those having the $uniqueness\ ratio_{p[1]} = 1$. Hence we obtain a list of pairs where each is $p\prime$=(primary column, potential foreign column). Next, we verify if the content of the $p\prime[2]$ is a subset of $p\prime[1]$. For the textual ones, if the ratio of the $\frac{size(p'[1] \cap p'[2])}{size(p'[2])} \geq 1-10^{-4}$, the pair is valid, where size return the size of the set. As for the numerical ones, it is sufficient to check if the range of the $p\prime[2]$ is a subset $p\prime[1]$ and the overlap is also larger or equal to $1 - 10^{-4}$. After performing these computations, the ground truth shows only two results summarized in the Table 5.9. For these two results, both KGLac and Aurum recovered them. Hence, this dataset does not give enough insight about the performance of KGLac. We use another dataset, ChEMBL [34] (22nd version), that provides the schema with a ground truth [7] for PKFKs.

[5] https://www.javadoc.io/doc/com.clearspring.analytics/stream/2.5.2/com/clearspring/analytics/stream/quantile/QDigest.html
[6] https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.DataFrame.approxQuantile.html
[7] https://ftp.ebi.ac.uk/pub/databases/chembl/ChEMBLdb/releases/chembl_22/archived/chembl_22_schema.png

Table 5.9: Columns having PKFK relationships in the ground truth.

PK		FK	
Table	**Column**	**Table**	**Column**
employee_info.csv	EmployeeID	employee_salaries.csv	EmployeeID
employee_salaries.csv	EmployeeID	employee_info.csv	EmployeeID

The ChEMBL [8] dataset is a manually curated database of bioactive molecules. It consists of 68 tables. However, based on the provided schema, some tables are missing. We generated GLac [9] for those tables and hosted it on Blazegraph. So for the evaluation, only the tables in the schema are kept. After cleaning, the dataset contains 46 tables and 289 columns. Between the different tables and based on the provided schema, we retrieved 32 PKFKs. To evaluate the systems, we need to set γ, the threshold for the cardinality (uniqueness ratio), and δ, a threshold for the domain overlap to determine inclusion dependency. For the experiments $\gamma_{KGLac} = \gamma_{Aurum} = 0.7$ for the same γ, we ran several experiments with a varying $\delta \in (0.2, 0.4, 0.6, 0.8)$. For textual columns, both systems consider a column to have an inclusion dependency with another one, if and only if it has a content similarity. And since the approach is the same for KGLac and Aurum, δ will only be used to determine the existence of an inclusion dependency between two numerical columns. Table 5.10 and Figure 5.12 summarise the results of the experiments.

Table 5.10: Number of correctly detected and the total number of detected PKFKS by KGLac and Aurum

Threshold	**0.2**		**0.4**		**0.6**		**0.8**	
System	***KGLac***	***Aurum***	***KGLac***	***Aurum***	***KGLac***	***Aurum***	***KGLac***	***Aurum***
Number of correclty detected PKFKs	23	22	22	21	22	21	17	15
Number of detected PKFKs	662	736	408	472	304	342	158	190

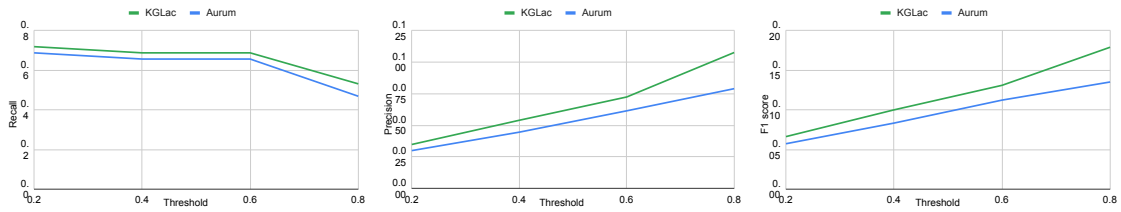


Figure 5.12: Evaluation of KGLac and Aurum using the ChEMBL dataset

[8] https://ftp.ebi.ac.uk/pub/databases/chembl/ChEMBLdb/releases/chembl_22/archived/
[9] https://github.com/CoDS-GCS/KGLac_materials

Table 5.10 reports the number of detected PKFKs and the number of the correctly detected ones for each system and each δ. Figure 5.12 shows three graphs containing the evaluation of both systems for three different metrics: Recall, Precision, F1 score (left to right). The results show that KGLac and Aurum have comparable performances. The recall graph is decreasing while those of the precision and f1 score are monotonously increasing. The allure for both systems for each metric is the same, with KGLac being slightly better. For a $\delta = 0.2$, they are capable of recovering the highest number of PKFKs. The recall is around 0.7. However as δ increases, the recall decreases to go below 0.6 at $\delta = 0.8$. Moreover, the precision keeps increasing to go above 0.1 for KGLac and 0.075 for Aurum for $\delta = 0.8$. For the same δ value, the F1 scores are almost triple for both systems. Despite adopting the same approach to establish PKFKs, the evaluation of both systems show a slight difference. In fact, similar to the content similarity relationships, the difference lies within how each system computes the quartiles. As mentioned in section 4.1.4 These statistics are important to determine the domain of each numerical column and compute the overlap for inclusion dependency. Hence, the difference stems from how each system profiles. Furthermore, for a low δ, they could detect the highest number of valid PKFKs at the expense of reporting several false positives. As a result, the recall is high while the precision is low. In contrast, for a higher δ the systems become more restrictive. In other words, they prune out previously detected false positives, simultaneously leading to increasing the number of false negatives. As a consequence, the recall decreases while the precision increases. In conclusion, both systems have similar performance, and based on the f1 score, the best value for δ is 0.8.

This chapter demonstrated a use case of how KGLac would help AI practitioners operate on data lakes to discover and enrich data. The use case consists of compiling a dataset to help a data scientist investigate the compensation gap between gender in the Information systems domain. In fact, the user could discover several tables containing pertinent information to accomplish the task. The use case shows how the system can be seamlessly integrated into a data science pipeline thanks to its interface services. In addition, the chapter evaluated the POC performance in detecting the various relationships in GLac. It also compared it with Aurum, a state-of-the-art system. The comparative analysis shows that for detecting content similarity and PKFKs, they both have a similar

performance. However, for the semantic similarity, KGLac provides better results because it pre-processes the raw data and leverages word embedding. The evaluation shows that there is room for improving the performance of KGLac. Hence, the next chapter concludes and lists future work.

Chapter 6

Conclusion & Future Work

In this thesis, we propose an approach to leverage the content of data lakes. It consists of two steps: First, it generates GLac, a highly interconnected RDF knowledge graph modeling the schema. Second, it equips the user with data discovery and data enrichment operations on the GLac. The work presents three contributions: A proof of concept system that processes the tabular-like content of data lakes to produce data profiles in a scalable manner. Then it leverages these profiles to build GLac using the methods based on statistics and deep learning. With the POC, the user considerably reduces the time spent manually examining the data to extract its metadata. LAC, a data lake ontology that conceptualizes relationships and entities in data lakes. LAC enriches GLac by provides predicates and constraints to reflect the semantic of data entities in data lakes [24]. A set of interface services to perform data discovery and data enrichment on top of GLac. The user interacts with GLac by means of user-friendly pre-defined APIs. The discovery operations range from a basic keyword search to finding paths between data entities along a given relationship. As for the semantic similarity ones, the user can find data based on column content or name embeddings.

Even though it is not part of the contribution of this thesis, the proof of the concept system generates promising results and GLac quality. When compared to Aurum, one of the state of the art systems in data discovery, the POC has comparable or better performance in detecting relationships between the entities in GLac. However, the results also show that there is room for improvement. For this reason, in the future, several approaches will be explored to not only boost the quality of GLac but also add more functionalities to the system. One of the venues to investigate is to support

more types of columns. Currently, all non-numerical types go under the umbrella of the textual one. Hence, KGLac can break it down into sub-types. [36] proposes a method to detect whether a column is a numerical, string (postal code or any domain-specific strings), or language (names or other standard text fields). This work is a starting point to support more data types and attain a richer GLac. To further enrich it, the system processes unstructured data like text documents and images. this will require amending *LAC* to provide adequate terms to model these entities and reflect the relationships that potentially govern them. In addition, the system can support more functionalities. First, a linker, which is a tool to find possible links between two KGs [24]. In fact, the linker serves two tasks: when presented with newly added files in the data lake, the user does not need to generate GLac from scratch, they can separately generate the KG for the new data, and then the linker will merge it with the existing one. In addition, it can help to link GLac with other existing KGs such Wikidata [1] or DBpedia [2]. To do so, the linker will leverage techniques like Graph Neural Networks [55, 61, 62] and reasoning[11]. As a result, this component will incrementally build GLac. In addition, Access Control is helpful in terms of managing whether a user is authorized to query the graph, use the embedding, or access the actual data. Lastly, the query manager can provide insight about the usage of the dataset by keeping track of how the user queries GLac [25].

[1] https://www.wikidata.org/wiki/Wikidata:Main_Page
[2] https://www.dbpedia.org/

Appendix A

LAC schema

Classes

Class: lac:Dataset

used with: lac:Table, lac:used_in, lac:insights
lac:dataset is a class to model a dataset in data lakes. A dataset contains a set of data that belong to the same domain.

Class: lac:Table

used with: lac:Dataset, lac:Column, lac:used_in, lac:insights
lac:table is a class to model tabular-like data files such as CSV and JSON.

Class: lac:Column

used with: lac:Table, lac:semanticSimilarity, lac:contentSimilarity, lac:inclusionDependency, lac:pkfk, lac:path, lac:distinctVCount, lac:totalVCount, lac:missingVCount
A class to represent a column in a table class.

Properties

Property: lac:semanticSimilarity

domain: lac:Column
range: lac:Column
lac:semanticSimilarity indicates that the two columns are semantically similar based on their names.

Property: lac:contentSimilarity

domain: lac:Column
range: lac:Column
lac:contentSimilarity indicates that the two columns have similar content based on their instances

Property: lac:inclusionDependency

domain: lac:Column
range: lac:Column
lac:inclusionDependency indicates that the content of a column is contained in other's.

Property: lac:pkfk

domain: lac:Column
range: lac:Column
lac:pkfk indicates that a column is foreign key to primary key column.

Property: lac:used_in

domain: lac:Column, lac:Table, lac:Dataset
range: rdf:Bag
lac:used_in indicates the different usages of the data entity resource.

Property: lac:insights

domain: lac:Column, lac:Table, lac:Dataset
range: rdf:Bag
lac:insights indicates the different insights provided by the user on the data entity resources.

Property: lac:path

domain: lac:Table, lac:Dataset
range: rdfs:Literal
lac:path provides the path of the dataset or table in the file system.

Property: lac:totalVCount

domain: lac:Column
range: rdfs:Literal
lac:totalVCount povides the total number of instances in a column.

Property: lac:distinctVCount

domain: lac:Column
range: rdfs:Literal
lac:distinctVCount provides the number of distinct instances in the column..

Property: lac:missingVCount

domain: lac:Column
range: rdfs:Literal
lac:missingVCount provides the number of missing instances in the column..

Property: lac:certainty

domain: Triple
range: rdfs:Literal
lac:certainty annotates a triple to reflect how certain of a similarity, inclusion dependency, or pkfk edges to be established.

Appendix B

LAC Vocabulary

```
<?xml version="1.0"?>
<rdf:RDF xmlns="http://www.example.com/lac#"
     xml:base="http://www.example.com/lac#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:owl="http://www.w3.org/2002/07/owl#"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
     xmlns:lac="http://www.example.com/lac#"
     xmlns:dc="http://purl.org/dc/elements/1.1/">
    <owl:Ontology rdf:about="http://www.example.com/lac#">
        <dc:description>LAC RDF vocabulary, described using W3C RDF
        Schema and the Web Ontology Language.</dc:description>
        <dc:title>LAC vocabulary</dc:title>
    </owl:Ontology>
```

```
<!--///////////////////////////////////////////////////////////
//
// Classes
//
///////////////////////////////////////////////////////////-->

<!-- http://www.example.com/lac#Dataset -->
<rdf:Description rdf:about="http://www.example.com/lac#Dataset">
    <rdfs:label>Dataset</rdfs:label>
    <rdfs:comment>A class to model a dataset in data lakes. A dataset
    contains a set of data that belong to the same domain.
    </rdfs:comment>
</rdf:Description>

<!-- http://www.example.com/lac#Table -->
<rdf:Description rdf:about="http://www.example.com/lac#Table">
    <rdfs:label>Table</rdfs:label>
    <rdfs:comment>A class to model tabular-like data files
    such as CSV and JSON.</rdfs:comment>
</rdf:Description>

<!-- http://www.example.com/lac#Column -->
<rdf:Description rdf:about="http://www.example.com/lac#Column">
    <rdfs:label>Column</rdfs:label>
    <rdfs:comment>A class to represent a column in a table class.
    </rdfs:comment>
</rdf:Description>
```

```
<!--///////////////////////////////////////////////////////////////
//
// Properties
//
///////////////////////////////////////////////////////////////-->

<!-- http://www.example.com/lac#semanticSimilarity -->
<owl:DatatypeProperty
rdf:about="http://www.example.com/lac#semanticSimilarity">
    <rdfs:comment>Indicates that the two columns are semantically
    similar based on their names.</rdfs:comment>
    <rdfs:domain rdf:resource="http://www.example.com/lac#Column"/>
    <rdfs:range rdf:resource="http://www.example.com/lac#Column"/>
    <rdfs:isDefinedBy rdf:resource="http://www.example.com/lac#"/>
    <rdfs:label>Semantic Similarity</rdfs:label>
</owl:DatatypeProperty>

<!-- http://www.example.com/lac#contentSimilarity -->
<owl:DatatypeProperty
rdf:about="http://www.example.com/lac#contentSimilarity">
    <rdfs:comment>Indicates that the two columns have similar content
    based on their instances.</rdfs:comment>
    <rdfs:range rdf:resource="http://www.example.com/lac#Column"/>
    <rdfs:isDefinedBy rdf:resource="http://www.example.com/lac#"/>
    <rdfs:label>Content Similarity</rdfs:label>
</owl:DatatypeProperty>
```

```
<!-- http://www.example.com/lac#inclusionDependency -->
<owl:DatatypeProperty
rdf:about="http://www.example.com/lac#inclusionDependency">
    <rdfs:comment>Indicates that the content of a column is contained
    in other's.</rdfs:comment>
    <rdfs:range rdf:resource="http://www.example.com/lac#Column"/>
    <rdfs:isDefinedBy rdf:resource="http://www.example.com/lac#"/>
    <rdfs:label>Inclusion Dependency</rdfs:label>
</owl:DatatypeProperty>

<!-- http://www.example.com/lac#pkfk -->
<owl:DatatypeProperty
rdf:about="http://www.example.com/lac#pkfk">
    <rdfs:comment>Indicates that a column is foreign key
    to primary key column.</rdfs:comment>
    <rdfs:range rdf:resource="http://www.example.com/lac#Column"/>
    <rdfs:isDefinedBy rdf:resource="http://www.example.com/lac#"/>
    <rdfs:label>PKFK</rdfs:label>
</owl:DatatypeProperty>

<!-- http://www.example.com/lac#used_in -->
<owl:DatatypeProperty rdf:about="http://www.example.com/lac#used_in">
    <rdfs:comment>Indicates the different usages of the data
    entity resource.</rdfs:comment>
    <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#
    <rdfs:isDefinedBy rdf:resource="http://www.example.com/lac#"/>
    <rdfs:label>Used in</rdfs:label>
</owl:DatatypeProperty>
```

```
<!-- http://www.example.com/lac#insights -->
<owl:DatatypeProperty rdf:about="http://www.example.com/lac#insights">
    <rdfs:comment>Indicates the different insights provided by the user
    on the data entity resources.</rdfs:comment>
    <rdfs:range
    rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Bag"/>
    <rdfs:isDefinedBy rdf:resource="http://www.example.com/lac#"/>
    <rdfs:label>Insights</rdfs:label>
</owl:DatatypeProperty>

<!-- http://www.example.com/lac#path -->
<owl:DatatypeProperty rdf:about="http://www.example.com/lac#path">
    <rdfs:comment>Provides the path of the dataset or table in the file
    system.</rdfs:comment>
    <rdfs:range
    rdf:resource="http://www.w3.org/2000/01/rdf-schema#Literal"/>
    <rdfs:isDefinedBy rdf:resource="http://www.example.com/lac#"/>
    <rdfs:label>Path</rdfs:label>
</owl:DatatypeProperty>
```

```
<!-- http://www.example.com/lac#totalVCount -->
<owl:DatatypeProperty
rdf:about="http://www.example.com/lac#totalVCount">
    <rdfs:comment>Povides the total number of instances
    in a column.</rdfs:comment>
    <rdfs:range
    rdf:resource="http://www.w3.org/2000/01/rdf-schema#Literal"/>
    <rdfs:isDefinedBy rdf:resource="http://www.example.com/lac#"/>
    <rdfs:label>Total Value Count</rdfs:label>
</owl:DatatypeProperty>

<!-- http://www.example.com/lac#distinctVCount -->
<owl:DatatypeProperty
rdf:about="http://www.example.com/lac#distinctVCount">
    <rdfs:comment>Povides the number of different instances in a column.
    </rdfs:comment>
    <rdfs:range
    rdf:resource="http://www.w3.org/2000/01/rdf-schema#Literal"/>
    <rdfs:isDefinedBy rdf:resource="http://www.example.com/lac#"/>
    <rdfs:label>Distinct Value Count</rdfs:label>
</owl:DatatypeProperty>
```

```
    <!-- http://www.example.com/lac#missingtVCount -->
    <owl:DatatypeProperty
    rdf:about="http://www.example.com/lac#missingVCount">
        <rdfs:comment>Povides the number of missing instances in a column.
        </rdfs:comment>
        <rdfs:range
        rdf:resource="http://www.w3.org/2000/01/rdf-schema#Literal"/>
        <rdfs:isDefinedBy rdf:resource="http://www.example.com/lac#"/>
        <rdfs:label>Missing Value Count</rdfs:label>
    </owl:DatatypeProperty>

    <!-- http://www.example.com/lac#certainty -->
    <owl:DatatypeProperty rdf:about="http://www.example.com/lac#certainty">
        <rdfs:comment>Annotates a triple to reflect how certain of a
        similarity,inclusion dependency, or pkfkedges to be established.
        </rdfs:comment>
        <rdfs:range
        rdf:resource="http://www.w3.org/2000/01/rdf-schema#Literal"/>
        <rdfs:isDefinedBy rdf:resource="http://www.example.com/lac#"/>
        <rdfs:label>Certainty</rdfs:label>
    </owl:DatatypeProperty>

</rdf:RDF>
```

Bibliography

[1] M. Arenas, A. Bertails, E. Prud'hommeaux, and J. Sequeda. A direct mapping of relational data to rdf, Sep 2012.

[2] D. Arndt, J. Broekstra, B. DuCharme, O. Lassila, P. F. Patel-Schneider, E. Prud'hommeaux, T. Thibodeau, Jr, and B. Thompson, Jul 2021.

[3] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives. Dbpedia: A nucleus for a web of open data. In *Proceedings of the 6th International The Semantic Web and 2nd Asian Conference on Asian Semantic Web Conference*, ISWC'07/ASWC'07, page 722–735, Berlin, Heidelberg, 2007. Springer-Verlag.

[4] J. Bao, D. Calvanese, B. C. Grau, M. Dzbor, A. Fokoue, C. Golbreich, S. Hawke, I. Herman, R. Hoekstra, I. Horrocks, E. Kendall, M. Krötzsch, C. Lutz, D. L. McGuinness, B. Motik, J. Pan, B. Parsia, P. F. Patel-Schneider, S. Rudolph, A. Ruttenberg, U. Sattler, M. Schneider, M. Smith, E. Wallace, Z. Wu, A. Zimmermann, J. Carroll, J. Hendler, and V. Kashyap. Owl 2 web ontology language, Dec 2012.

[5] T. Berners-Lee. Relational databases on the semantic web, Sep 1998.

[6] S. Bonner, I. P. Barrett, C. Ye, R. Swiers, O. Engkvist, C. T. Hoyt, and W. L. Hamilton. Understanding the performance of knowledge graph embeddings in drug discovery, 2021.

[7] E. P. Bontas, M. Mochól, and R. Tolksdorf. Case studies on ontology reuse. 2005.

[8] D. Brickley, M. Burgess, and N. Noy. Google dataset search: Building a search engine for datasets in an open web ecosystem. In *The World Wide Web Conference*, WWW '19, page 1365–1375, New York, NY, USA, 2019. Association for Computing Machinery.

[9] D. Brickley and L. Miller. Foaf vocabulary specification 0.99, Jan 2014.

[10] A. Broder. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No.97TB100171)*, pages 21–29, 1997.

[11] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. Jena: Implementing the semantic web recommendations. WWW Alt. '04, page 74–83, New York, NY, USA, 2004. Association for Computing Machinery.

[12] R. Castro Fernandez, Z. Abedjan, F. Koko, G. Yuan, S. Madden, and M. Stonebraker. Aurum: A data discovery system. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1001–1012, 2018.

[13] S. Das, S. Sundara, and R. Cyganiak, Sep 2012.

[14] J. Dixon. Pentaho, hadoop, and data lakes, Oct 2014.

[15] F. Dürsch, A. Stebner, F. Windheuser, M. Fischer, T. Friedrich, N. Strelow, T. Bleifuß, H. Harmouch, L. Jiang, T. Papenbrock, and F. Naumann. Inclusion dependency discovery: An experimental evaluation of thirteen algorithms. CIKM '19, page 219–228, New York, NY, USA, 2019. Association for Computing Machinery.

[16] L. Ehrlinger and W. Wöß. Towards a definition of knowledge graphs. 09 2016.

[17] Elasticsearch. Scalability and resilience: clusters, nodes, and shards.

[18] M. Färber, F. Bartscherer, C. Menne, and A. Rettinger. Linked data quality of dbpedia, freebase, opencyc, wikidata, and yago. *Semantic Web*, 9:77–129, 2018.

[19] K. D. Foote. https://www.dataversity.net/enterprise-data-unification-and-knowledge-graphs-making-complexity-simple/, Jul 2019.

[20] T. R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.

[21] R. V. Guha and B. McBride. Rdf schema 1.1, Feb 2014.

[22] A. Halevy, F. Korn, N. F. Noy, C. Olston, N. Polyzotis, S. Roy, and S. E. Whang. Goods: Organizing google's datasets. *SIGMOD*, 2016.

[23] O. Hartig. Rdf* and sparql*: An alternative approach to annotate statements in rdf. In *International Semantic Web Conference*, 2017.

[24] A. Helal. Data lakes empowered by knowledge graph technologies. SIGMOD/PODS '21, page 2884–2886, New York, NY, USA, 2021. Association for Computing Machinery.

[25] A. Helal, M. Helali, K. Ammar, and E. Mansour. A demonstration of kglac: A data discovery and enrichment platform for data science. *Proc. VLDB Endow.*, 14(12), 2021.

[26] A. Hogan, E. Blomqvist, M. Cochez, C. d'Amato, G. de Melo, C. Gutierrez, J. E. L. Gayo, S. Kirrane, S. Neumaier, A. Polleres, R. Navigli, A.-C. N. Ngomo, S. M. Rashid, A. Rula, L. Schmelzeisen, J. Sequeda, S. Staab, and A. Zimmermann. Knowledge graphs, 2020.

[27] A. Holst. Total data volume worldwide 2010-2025, Jun 2021.

[28] T. Johnson. *Data Profiling*, pages 604–608. Springer US, Boston, MA, 2009.

[29] J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of Massive Datasets*. Cambridge University Press, USA, 2nd edition, 2014.

[30] M. Lock. Angling for insight in today's data lake, Oct 2017.

[31] G. W. M. Kroetsch. Special issue on knowledge graphs. Aug 2016.

[32] E. Mansour, D. Deng, R. C. Fernandez, A. A. Qahtan, W. Tao, Z. Abedjan, A. Elmagarmid, I. F. Ilyas, S. Madden, M. Ouzzani, M. Stonebraker, and N. Tang. Building data civilizer pipelines with an advanced workflow engine. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1593–1596, 2018.

[33] D. L. McGuinness and F. van Harmelen. Owl 2 web ontology language, 02 2004.

[34] D. Mendez, A. Gaulton, A. P. Bento, J. Chambers, M. De Veij, E. Félix, M. Magariños, J. Mosquera, P. Mutowo, M. Nowotka, M. Gordillo-Marañón, F. Hunter, L. Junco, G. Mugumbate, M. Rodriguez-Lopez, F. Atkinson, N. Bosc, C. Radoux, A. Segura-Cabrera, A. Hersey, and A. Leach. ChEMBL: towards direct deposition of bioassay data. *Nucleic Acids Research*, 47(D1):D930–D940, 11 2018.

[35] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space, 2013.

[36] J. Mueller and A. Smola. Recognizing variables from their data via deep embeddings of distributions. *2019 IEEE International Conference on Data Mining (ICDM)*, pages 1264–1269, 2019.

[37] F. Nargesian, K. Q. Pu, E. Zhu, B. Ghadiri Bashardoost, and R. J. Miller. Organizing data lakes for navigation. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1939–1950, New York, NY, USA, 2020. Association for Computing Machinery.

[38] F. Nargesian, E. Zhu, R. J. Miller, K. Q. Pu, and P. C. Arocena. Data lake management: Challenges and opportunities. *Proc. VLDB Endow.*, 12(12):1986–1989, Aug. 2019.

[39] F. Naumann. Data profiling revisited. *SIGMOD Rec.*, 42(4):40–49, Feb. 2014.

[40] N. Noy, Y. Gao, A. Jain, A. Narayanan, A. Patterson, and J. Taylor. Industry-scale knowledge graphs: Lessons and challenges. *Commun. ACM*, 62(8):36–43, July 2019.

[41] M. Ota, H. Müller, J. Freire, and D. Srivastava. Data-driven domain discovery for structured datasets. *Proc. VLDB Endow.*, 13(7):953–967, Mar. 2020.

[42] F. Özcan, C. Lei, A. Quamar, and V. Efthymiou. Semantic enrichment of data for ai applications. In *Proceedings of the Fifth Workshop on Data Management for End-To-End Machine Learning*, DEEM '21, New York, NY, USA, 2021. Association for Computing Machinery.

[43] H. Paulheim. Knowledge graph refinement: A survey of approaches and evaluation methods. *Semantic Web*, 8:489–508, 2017.

[44] J. Pennington, R. Socher, and C. D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

[45] F. Pérez and B. E. Granger. IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, May 2007.

[46] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, Dec. 2001.

[47] S. Sahoo, W. Halb, S. Hellmann, K. Idehen, T. Jr, S. Auer, J. Sequeda, and A. Ezzat. A survey of current approaches for mapping of relational databases to rdf. *W3C*, 01 2009.

[48] J. F. Sequeda, W. J. Briggs, D. P. Miranker, and W. P. Heideman. A pay-as-you-go methodology to design and build enterprise knowledge graphs from relational databases. In C. Ghidini, O. Hartig, M. Maleshkova, V. Svátek, I. Cruz, A. Hogan, J. Song, M. Lefrançois, and F. Gandon, editors, *The Semantic Web – ISWC 2019*, pages 526–545, Cham, 2019. Springer International Publishing.

[49] S. Stardog Union. Bring actionable meaning to your data.

[50] M. Stonebraker, L. Rowe, and M. Hirohama. The implementation of postgres. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):125–142, 1990.

[51] Systap. bigdata. Technical report, May 2013.

[52] J. Vanschoren, J. N. van Rijn, B. Bischl, and L. Torgo. Openml: Networked science in machine learning. *SIGKDD Explorations*, 15(2):49–60, 2013.

[53] P. Wang, Y. He, R. Shea, J. Wang, and E. Wu. Deeper: A data enrichment system powered by deep web. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 1801–1804, New York, NY, USA, 2018. Association for Computing Machinery.

[54] J. Weiner and N. Bronson. Facebook's top open data problems, Oct 2014.

[55] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, pages 1–21, 2020.

[56] S. Xia, N. Anzum, S. Salihoglu, and J. Zhao. Ktabulator: Interactive ad hoc table creation using knowledge graphs. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI '21, New York, NY, USA, 2021. Association for Computing Machinery.

[57] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, page 2, USA, 2012. USENIX Association.

[58] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, page 10, USA, 2010. USENIX Association.

[59] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, Oct. 2016.

[60] Y. Zhang and Z. G. Ives. Finding related tables in data lakes for interactive data science. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1951–1966, New York, NY, USA, 2020. Association for Computing Machinery.

[61] Z. Zhang, P. Cui, and W. Zhu. Deep learning on graphs: A survey. *IEEE Transactions on Knowledge and Data Engineering*, pages 1–1, 2020.

[62] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, and M. Sun. Graph neural networks: A review of methods and applications. *CoRR*, abs/1812.08434, 2018.

[63] E. Zhu, D. Deng, F. Nargesian, and R. J. Miller. Josie: Overlap set similarity search for finding joinable tables in data lakes. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 847–864, New York, NY, USA, 2019. Association for Computing Machinery.