Scientific Lake

Deliverable D2.1: Initial version of the Scientific Lake service

Due Date of Deliverable	30/06/2024
Actual Submission Date	30/06/2024
Work Package	WP2
Tasks	T2.1, T2.2, T2.3, T2.4
Type	OTHER
Approval Status	Submitted
Version	v1.0
Number of Pages	39

The information in this document reflects only the author's views and the European Commission is not liable for any use that may be made of the information contained therein. The information in this document is provided "as is" without guarantee or warranty of any kind, express or implied, including but not limited to the fitness of the information for a particular purpose. The user thereof uses the information at his/ her sole risk and liability.

DRAFT

Abstract

This deliverable report provides an overview of the developments within Work Package 2 (WP2) of the SciLake project, detailing the progress on the initial version of the Scientific Lake service. The service aims to streamline knowledge acquisition, knowledge graph creation, and Scientific Lake navigation thus facilitating access to a vast and diverse array of scientific datasets. By aggregating and indexing large-scale datasets into a collection of Scientific Knowledge Graphs (SKGs) and providing efficient ways to access and reason over this data, the service enhances the discoverability and usability of research data and provides a robust data management foundation for downstream value-added services developed in other work packages of this project (WP3 and WP4).

This project has received funding from the European Union's Horizon Europe framework programme under grant agreement No. 101058573. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Executive Agency. Neither the European Union nor the European Research Executive Agency can be held responsible for them.

DRAFT

Revision history

Version	Date	Reason	Revised by
0.0	9/5/2024	Agreement on structure	N. Yakovets
0.1	7/6/2024	First Draft	N. Yakovets
0.2	10/6/2024	Ready for Peer review	N. Yakovets
0.3	28/6/2024	Peer review comments addressed	N. Yakovets
1.0	29/6/2024	Final Version after proofreading	N. Yakovets

Author List

Organisation	Name	Contact Information
TU Eindhoven	Daan de Graaf	d.j.a.d.graaf@tue.nl
TU Eindhoven	Yuanjin Wu	y.wu3@tue.nl
TU Eindhoven	Nikolay Yakovets	hush@tue.nl
Athena RC	Thanasis Vergoulis	vergoulis@athenarc.gr

Contributor List

Organisation	Name	Contact Information
CNR	Miriam Baglioni	miriam.baglioni@isti.cnr.it
Athena RC	Thanasis Vergoulis	vergoulis@athenarc.gr
ICM	Marek Horst	mhorst@icm.edu.pl

Table of Contents

1. Executive Summary 7
2. Introduction 8
3. Design 9
3.1. Initial requirements 9
3.2. Design approach 10
3.3. Integration into SciLake ecosystem 11
4. Implementation 11
4.1. Data acquisition and catalogue 11
4.1.1. Data acquisition 11
Description 11
Components of the Service 12
Deployment 13
4.1.2. Data catalogue 14
4.2. Knowledge Graph Creation Assistant 16
4.2.1. R2PG-DM 17
Limitations of Previous Mappings 17
Introduction of Direct Mapping 17
Properties of DM 18
Implementation 19
4.2.2. ProGGD-Graph Data Profiling with GGDs 19
Use cases of ProGGD 19
Introduction and Functionalities of ProGGD 19
4.2.3. ProGGD Pipeline 21
4.3. Data Interlinking 25
4.3.1. Introduction of Data Interlinking 25
4.3.2. Limitations of the state of the art 26
4.3.3. sHINER for Entity Resolution 26
G-Core Interpreter 27
GGDs Component 27
4.3.4. Use Case of Cancer Pilot 28
4.4. Data Lake search & navigation 29
4.4.1. Data management for SKGs 29
4.4.2. Integrating Analytics in Databases 30
4.4.3. GraphAlg: Algorithm Support Done Right 33
4.4.4. Use Cases for GraphAlg in Other Work Packages 33

DRAFT

4.4.5. Implementation of GraphAlg 34
GraphAlg Language Design 34
Integration With Regular Queries 35
Compiling GraphAlg 36
Simplification to MATLANG With Loops 36
Lowering to Extended Relational Algebra 36
4.4.6. Demonstration 37
5. Conclusions **38**
6. References **39**

List of Figures

Figure 1: Key components of data acquisition and catalogue services.

Figure 2: Workflows in the knowledge graph creation and interlinking.

Figure 3: The overview of the ProGGD functionalities.

Figure 4: ProGGD main panel.

Figure 5: ProGGD attributes panel.

Figure 6: ProGGD graph pattern panel.

Figure 7: ProGGD graph generating dependencies panel.

Figure 8: sHINer system architecture.

Figure 9: Using GGDs for entity resolution in the cancer pilot.

Figure 10: Architecture of the AvantGraph graph processing engine.

Figure 11: Different approaches to graph analytics in databases.

Figure 12: Single-source shortest paths in GraphAlg.

Figure 13: Definition of expressions in the GraphAlg core language.

Figure 14: An example sparse matrix converted into an equivalent set of tuples.

DRAFT

Abbreviation List

API: Application Programming Interface

DSL: Domain-Specific Language

ER: Entity Resolution

GGD: Graph Generating Dependency

GQL: Graph Query Language

IR: Intermediate Results

KG: Knowledge Graph

MATLANG: Matrix Language

ProGGD: Graph Data Profiling with GGDs

R2PG-DM: Relational to Property Graph Direct Mapping

RDB: Relational Database

REP: Regional Energy Planning

REST: Representational State Transfer

S3: Simple Storage Service

SKG: Scientific Knowledge Graph

SQL: Structured Query Language

URL: Uniform Resource Locator

DRAFT

1. Executive Summary

Scientific knowledge plays a pivotal role in deepening our understanding of the world, driving technological innovations, and improving our everyday lives. Additionally, past discoveries lay the groundwork for future research, building a cumulative foundation of understanding. Today, the volume of scientific knowledge is larger than ever, unlocking tremendous potential for future advancements. However, this knowledge is often fragmented and stored in heterogeneous formats, hindering discovery and the extraction of valuable insights necessary for informed decision-making. Addressing these challenges is crucial for maximising the impact of scientific research and fostering continued progress.

Knowledge Graph (KG) and graph database technologies offer powerful technologies for organising and analysing domain knowledge. By structuring data into interconnected nodes and relationships, they provide a more intuitive and flexible way to represent complex information compared to traditional databases. This interconnected structure allows for the seamless integration of heterogeneous data sources, making it easier to discover patterns and relationships that might otherwise remain hidden. Advanced query capabilities enable users to traverse these connections, uncovering valuable insights and facilitating informed decision-making.

Unfortunately, transforming domain knowledge into more structured formats is challenging because domain experts typically lack the specialised technical expertise required for this task. Conversely, knowledge management experts alone cannot effectively organise the information without domain-specific insights. To address this problem, SciLake is introducing the Scientific Lake service, a suite of customizable components designed to facilitate the organisation, querying, and analysis of scientific knowledge. This innovative approach bridges the gap between domain experts and knowledge management specialists, enabling a more efficient and accurate transformation of domain knowledge into structured formats.

In this report, we present the initial version of this Scientific Lake service. It provides a data management foundation for SciLake's Scientific Knowledge Graphs (SKGs) and streamlines data acquisition, catalogue management, KG creation and navigation across diverse scientific domains. Our goal is to address the challenges associated with managing and utilising large volumes of data in scientific research.

The rest of this document is structured as follows. Section 2 offers a short introduction to our motivation and the challenges we have identified. Section 3 outlines the design process followed. Section 4 discusses the implementation of major components of the service: data acquisition and catalogue (4.1), knowledge graph creation assistant (4.2), data enrichment and

DRAFT

interlinking (4.3), and data lake search and navigation (4.4). Section 5 discusses the future work and concludes.

2. Introduction

In the era of data-driven scientific research, the ability to efficiently manage, integrate, and analyse vast amounts of heterogeneous scientific data to produce knowledge has become crucial. The diversity of scientific domains and the lack of common practices in representing and sharing scientific knowledge further contribute to the aforementioned heterogeneity. This situation creates significant impediments to scientific discovery and progress, hindering the extraction of valuable insights necessary for informed decision-making by researchers and other professionals involved in research, such as research funding organisation officers.

The SciLake project aims to address this challenge by introducing the concept of a Scientific Lake service - a comprehensive data management and analysis framework designed to enhance the accessibility and usability of scientific knowledge across diverse domains.

This deliverable report outlines the initial version of the Scientific Lake service. The service aims to streamline the process of knowledge acquisition, knowledge graph creation, and navigation within Scientific Knowledge Graphs (SKGs) in a Scientific Lake. By aggregating and indexing large-scale datasets into a collection of SKGs, the service enhances the discoverability and usability of research data, providing a robust foundation for downstream value-added services.

The Scientific Lake service addresses several key challenges in scientific data management:

1. Efficient data acquisition and cataloguing from diverse sources
2. Semi-automated creation of Knowledge Graphs from heterogeneous data
3. Interlinking of data across different scientific domains
4. Advanced search and navigation capabilities within the Scientific Lake

By tackling these challenges, the Scientific Lake service aims to enable researchers with more efficient tools for data discovery, integration, and analysis. This, in turn, has the potential to accelerate scientific progress by enabling more comprehensive and insightful research across disciplines.

DRAFT

3. Design

In this section, we briefly discuss the main requirements of the service, the process followed during its design, and how it is integrated into the SciLake ecosystem. More details about the design process can be found in Deliverable D1.2 ("Initial integrated system").

3.1. Initial requirements

In our design, we considered the following initial requirements for the Scientific Lake service.

First, for **Data acquisition and catalogue**, we aimed to implement mechanisms for acquiring full-text publications from open access sources and develop crawlers and scrapers for retrieving scholarly content from web resources. We also planned to create a toolset to facilitate community-driven content acquisition, implement a data catalogue to keep track of all acquired and generated data assets, and ensure proper metadata management for all acquired content.

For the **Knowledge Graph Creation Assistant**, our requirements included developing tools for creating Scientific Knowledge Graphs (SKGs) from structured and unstructured data sources, and supporting semantic modelling and maintenance of primary and auxiliary ontologies for SKGs. We also aimed to implement information extraction techniques for entities, relations, and properties from unstructured raw data, provide schema mapping tools for transforming (semi-)structured data to property graphs, and develop a graph profiler to showcase graph information using Graph Generating Dependencies.

In terms of **Data Interlinking**, we focused on developing tools and methods for interlinking data across different SKGs and implementing entity resolution techniques to identify matching entities across heterogeneous graph data. We also aimed to support ontology alignment to facilitate integration of different knowledge domains and provide mechanisms for federation of multiple SKGs.

For **Data Lake search and navigation**, our requirements included implementing advanced search functionalities across heterogeneous scholarly content and developing data virtualization techniques to provide a unified view of diverse data sources. We also planned to create powerful navigation tools to explore the contents

of the Scientific Lake, support complex query capabilities across structured and unstructured data, and implement efficient indexing and retrieval mechanisms for large-scale data.

Finally, we established a number of **cross-cutting requirements**. These included ensuring scalability to handle large volumes of heterogeneous scholarly content, implementing data preservation practices, and ensuring FAIRness of all managed data. We also aimed to support customization and integration with existing workflows of researchers across different disciplines, implement community governance mechanisms to democratise scholarly content management, ensure openness and transparency in all aspects of the service, and comply with relevant standards and best practices in data management and interoperability.

3.2. Design approach

To address these requirements, we followed a user-centric design process with the following key principles. First, we **co-designed with pilot partners.** We closely involved the pilot partners representing different research domains (neuroscience, cancer research, transportation research, energy research) from the early stages of the design process. This helped ensure the service addresses real needs across diverse scientific communities. We employed **agile prototyping**. We adopted an iterative approach with short development cycles, allowing for rapid prototyping and continuous refinement based on feedback. This enabled us to quickly test ideas and adjust the design as needed. We designed a **modular system architecture** to enable flexibility, extensibility, and customization for different use cases. This allows components to be developed and updated independently. Further, we paid special attention to **scalability**. The architecture was designed to be scalable from the start, able to handle large volumes of heterogeneous data. We emphasised **standards compliance**. We aligned our design with relevant standards and best practices in data management, preservation, and interoperability. Finally, we always aim to use **open APIs**. Towards this, we prioritise the development of comprehensive, well-documented APIs to facilitate integration with external systems and workflows.

DRAFT

3.3. Integration into SciLake ecosystem

The Scientific Lake service is designed as a foundational component of the overall SciLake ecosystem. It provides the core data management infrastructure on which other SciLake services are built. The Scientific Knowledge Graphs created and managed by this service serve as key data sources for the smart impact analysis (WP3) and reproducibility assistance (WP4) services. Its APIs enable seamless integration with other SciLake components and external systems. The service implements the common data models and standards defined for the SciLake project to ensure interoperability across services. By serving as the central data hub, the Scientific Lake enables the creation of a cohesive ecosystem of services to support open, data-driven scientific research.

4. Implementation

4.1. Data acquisition and catalogue

4.1.1. Data acquisition

DESCRIPTION

The PDF Aggregation Service handles the acquisition of the Open Access publications (i.e. the PDF file) in the OpenAIRE ecosystem.

It leverages publication URLs obtained from the publication's metadata available in the OpenAIRE Graph and employs cutting-edge algorithms to crawl the Web effectively. Its primary function is to locate and download full texts of open-access publications, prioritising recently published ones in the download attempts. Throughout this process, it remains respectful of server capacities at repositories and publishers.

To efficiently manage the processing and download of millions of publications, the PDF Aggregation Service orchestrates a distributed execution system on the Cloud. This system relies on multiple microservices running in parallel for optimal performance. All the retrieved publications are securely stored within an S3 Object Store. Furthermore,

DRAFT

the service utilises an advanced pattern-matching algorithm to analyse the structural elements of web pages, like tags, attributes and links, to pinpoint full-text URLs. Additionally, during its execution, it monitors various performance indicators to optimise the crawling speed and ensure maximum efficiency.

The PDF Aggregation Service also allows to bulk-import full texts from compatible data sources, significantly accelerating the overall collection process. This allows the service to efficiently gather a vast amount of full-text publications, ultimately enriching the OpenAIRE ecosystem with valuable scientific content to text-mine.

Components of the Service

The PDF Aggregation Service operates as a distributed system, efficiently collecting full texts of open access publications. This system consists of several key components working in concert:

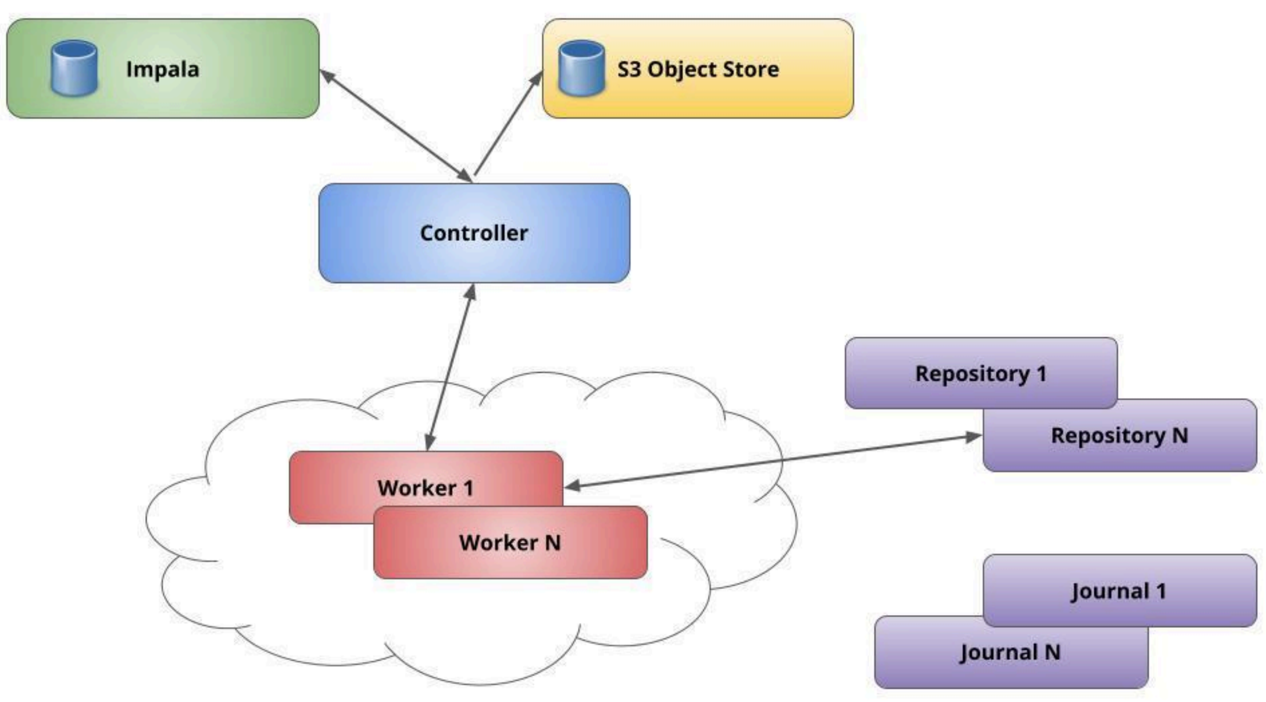

Figure 1: *Key components of data acquisition and catalogue services*

Controller: The brain of the operation, the Controller receives requests from worker applications deployed on the Cloud. It utilises a database (Apache Impala[1]) to construct a list of assignments for these workers and then transmits this list back for processing. Once complete, the Controller retrieves "Worker Reports" and requests full-text files in batches. These retrieved full texts are then uploaded to the S3 Object Store for secure storage. Finally, the Controller updates the database with relevant reports and the newly acquired file locations. It can also handle bulk import requests from compatible data sources, bypassing the crawling stage and directly receiving full-text files for immediate storage. In the bulk import process, external services deposit research full texts in a designated directory in the machine hosting the Controller. The Controller, via an external call, is prompted to load the full texts. The Controller then generates the OpenAIRE identifier for each file and uploads them in the S3 Object Store. Lastly, it generates a “payload” record with the basic information for the file and its OpenAIRE identifier, thus connecting the pdf to the metadata of the publication existing in the OpenAIR Graph.

Worker: These worker applications, deployed in multiple instances on the Cloud, request assignments from the Controller. Each worker utilises the Publications Retriever software to process the assigned tasks and download available full texts. Once processing is complete, the results are reported back to the Controller. The Controller then requests any missing full texts from the worker in batches. These requested files are compressed using Facebook's Zstandard[2] compression algorithm before transmission.

Impala: This database serves as the central repository for storing the state of the download process. Information such as the download time for a PDF, the number of attempts made in case of failures, and other relevant data are tracked within Impala. Additionally, it prepares assignment lists for the Controller and stores information derived from worker reports.

S3 Object Store: This secure storage solution houses all retrieved full texts.

Deployment

The Controller instance, accessible through source code, is deployed within a Docker container. This container connects to six worker instances (source code). These worker instances operate on virtual machines hosted on Google Cloud. They utilise the Publication Retriever software (source code). Finally, the code relies on an Impala database and an S3 Object Store hosted on the same cluster.

[1] Apache Impala: https://impala.apache.org/
[2] Zstandard: https://facebook.github.io/zstd/

4.1.2. Data catalogue

A crucial component of the SciLake project is the creation of a comprehensive resource catalogue. This catalogue serves as a central registry, outlining the various tools and datasets (e.g., SKGs) which are part of the SciLake ecosystem and available for use in the SciLake use cases. These resources are developed, maintained, and extended throughout the project's lifecycle.

The aim of the catalogue is to facilitate the exploration of the SciLake ecosystem, facilitating the discovery of resources, their contents, functionalities, and specifications. By presenting a clear and concise overview of each resource, the catalogue empowers researchers (or other end-users) to identify the specific tools/datasets that align with their needs and workflows. This fosters efficient adoption and utilisation of the SciLake offerings, ultimately accelerating scientific progress within the interconnected research environment envisioned by the project.

The catalogue itself delves deeper, detailing the functionalities and target audience for each product. This includes tools for creating, managing, and interlinking Scientific Knowledge Graphs (SKGs) across various disciplines. Additionally, the catalogue showcases smart services designed to aid researchers in discovering emerging research trends, identifying valuable research outputs, and enhancing research reproducibility. For ease of access, the catalogue provides clear descriptions, user guides, and links to relevant technical specifications for each product.

The catalogue leverages a two-part architecture: a back-end and a front-end.

- **Back-end:** The back-end infrastructure builds upon the one developed by ARC for the Intelcomp Project (https://code-repo.d4science.org/D-Net/scilake-catalogue). This foundation ensures a robust and proven technical base.
- **Front-end:** The front-end interface is specifically tailored to match the needs of SciLake users. The source code can be found on GitHub (https://code-repo.d4science.org/D-Net/scilake-catalogue-ui).

This combined approach fosters a user-friendly and efficient information resource aligned with the SciLake information space.

DRAFT

The SciLake product catalogue leverages Docker containers for a robust and scalable deployment. Two dedicated containers manage the back-end and front-end functionalities. External volumes, independent of the Docker containers, house the database and search engine critical for back-end operations. The front-end interacts solely with the back-end, which in turn handles communication with other components. This deployment resides on a Docker cluster hosted on CNR premises. The cluster ensures high availability through a configuration of 3 master nodes and 8 worker nodes distributed across separate physical machines.

Code repo(s):
https://code-repo.d4science.org/lsmyrnaios/UrlsController
https://code-repo.d4science.org/lsmyrnaios/UrlsWorker
https://github.com/LSmyrnaios/PublicationsRetriever

DRAFT

4.2. Knowledge Graph Creation Assistant

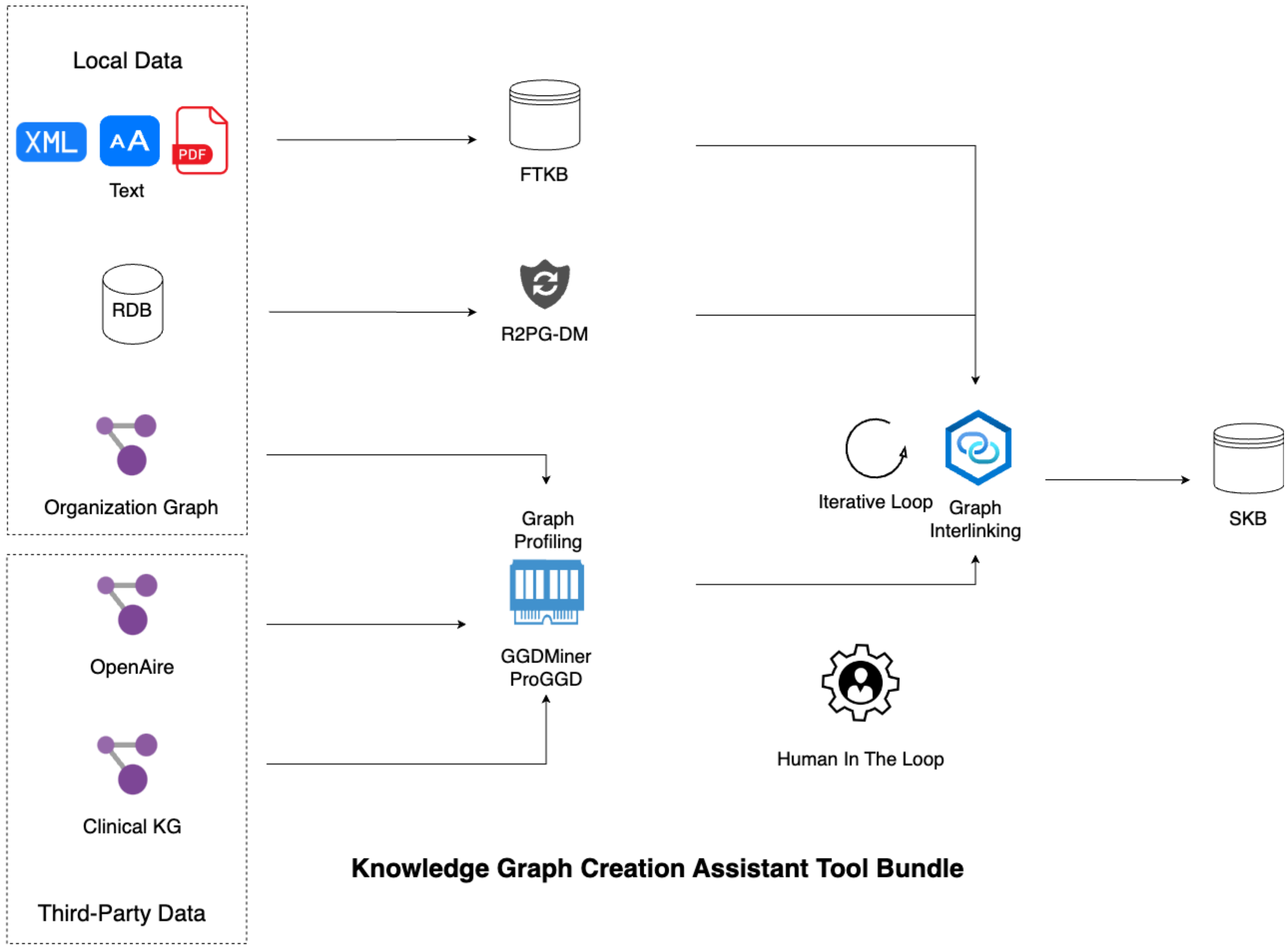

Figure 2: *Workflows in the knowledge graph creation and interlinking*

Within the **Knowledge Graph Creation Assistant Tool Bundle**, our aim is to deliver a set of automated and semi-automated tools to facilitate the creation of a collection of SKGs, following the task of data acquisition. Use-case partners have different data sources which could be structured as well as unstructured and would utilise the tools we provide to create their own pipeline with respect to the creation of a knowledge graph. Unstructured data would be transformed to (semi-)structured data by information (entity, relation, property) extraction. Schema mapping tools are developed to transform (semi-)structured data to a property graph which captures both topological and data artefacts in a flexible-schema data structure. For example, **R2PG-DM** is a direct mapping from relational databases to property graphs. Moreover,

ProGGD, a graph profiler, is developed to employ Graph Generating Dependencies to showcase graph's information, such as interaction between graph entities, relationships and graph patterns.

4.2.1. R2PG-DM

It is well known that large amounts of enterprise data is stored in relational databases (RDB) [6]. Since our goal is to assist in creation of Knowledge Graphs from available enterprise data sources, it is essential to handle the translation of relational data into a graph.

The **R2PG-DM** tool developed as a part of our Knowledge Graph creation assistant bundle aims to provide an automated translation of data stored in the relational databases (*relational data model* [4]) into a property graph (*property graph data model* [5]).

Limitations of Previous Mappings

Previous mappings focus on the idea of query efficiency, which try to optimise the query execution over the generated graphs. As a result, the generated graphs are either too complex, lossy or obfuscate the input RDB schema. Moreover, proving the correctness of these mappings is difficult, since it is infeasible to reason over properties such as information preservation, query preservation or semantics preservation, because of being procedural and by aggregating data. Different from the previous mappings, R2PG-DM is a direct mapping which follows a natural, logical translation of relation databases to property graphs by preserving the schema and by reasoning over basic properties of a direct mapping.

Introduction of Direct Mapping

A direct mapping M is a total function from RC to G, where G is the set of all property graphs and RC is the set of all triples of the form (R,Σ,I) where R is a relational schema, I is its corresponding instance and Σ is a set of PKs and FKs over R. A direct mapping translates relational databases into property graphs without any input from the user. The input of a direct mapping M is a relational schema R, its corresponding instance I and a set Σ of PKs and FKs over R. The output is a property graph. Direct mapping is defined as a set of Datalog rules, which are divided in two parts: translate relational instances and translate relational schemas.

To translate a relational database to a property graph, we applied Datalog rules [7] for the generation of nodes, edges and properties. Several Datalog rules are also

DRAFT

represented for encoding a relation schema into a property graph. Corresponding details could be seen in [1].

Properties of DM

The properties of R2PG-DM include two fundamental properties: information preservation and query preservation; and two desirable properties: monotonicity and semantics preservation. The fundamental properties are important because they assure that the data is unaltered after the direct mapping is applied, which is meaningful for our SciLake pilots. It can contribute to the reusability and reproducibility of research to support energy planning towards eco-transition for regional energy planning (REP) pilot. Also, the query preservation guarantees that every relational algebra query over the relational database has a G-CORE query equivalent over the resulting property graph. This fundamental property is very powerful as it assures that G-CORE queries can be translated over other more popular query languages used in real-life such as SQL. In addition, we have monotonicity as a desirable property. This would ensure that any update to the relational database will have a minimal impact on the mapping. Finally, the semantics preservation property ensures that any violated integrity constraints of relational databases are also reflected as inconsistency over the resulting property graph.

1. **Information preservation**: A direct mapping is information preserving if no information about the relational instance being translated is lost during the mapping process.
2. **Query preservation**: A direct mapping is query preserving if every query over a relational database can be translated into an equivalent query over the resulting property graph.
3. **Monotonicity**: A direct mapping monotone if for any pair of instances (I_1,I_2), where I_1 is a subset of I_2, the property graph generated from the instance I_1 is a subset of the property graph generated from the instance I_2.
4. **Semantics preservation**: A direct mapping is semantics preserving if the mapping reflects the condition of a set of PKs and FKs defined over a relational schema R. It generates a consistent property graph from corresponding instance when the set Σ of PKs and FKs over the relational schema R is consistent.

Implementation

R2PG-DM has been implemented as a Java application using Maven[3] as the build automation tool. The project is open-source and can be found in the following Github repository: https://github.com/avantlab/R2PG-DM. The JDBC library has been used for obtaining the schema of the database in order to construct the mapping. We wanted the Java application to be as generic as possible. R2PG-DM takes as input a database instance from which the metadata is extracted such as relation names and foreign keys.The output is stored into a relational instance as well into three different tables: node, property and edge. The two connections are retrieved at run-time from a configuration file. It is important to mention that three classes are used for the implementation, each representing a property graph object as follows: (1) a Node class is used for storing the node properties (id, ,label), (2) a Property class is used for storing the property properties (id, key, value), and (3) an Edge class is used for storing the edge properties (id, sourceId, targetId, label). The implementation starts by retrieving all the relational names from the input database. Then the algorithm is divided into two steps: (1) first, we create nodes and properties and (2) second, we create the edges.

4.2.2. ProGGD-Graph Data Profiling with GGDs

Use cases of ProGGD

In Knowledge Graphs, especially in big graphs from pilots such as the Clinical Knowledge Graph used by the Cancer Research pilot, the interplay among different graph entities, properties, and more complex patterns that emerge in the graph is crucial to understanding its content. Although many methods for profiling Knowledge Graphs have been proposed, few systems employ graph data dependencies to represent information about the data, which conveys detailed information about the Knowledge Graph to the user, and in turn, offer insights into the quality of the data.

Introduction and Functionalities of ProGGD

ProGGD is a system that employs Graph Generating Dependencies to showcase information about the graph's content. We identify GGDs from the data based on the frequency of a graph pattern's appearance and the prevalence of similar or correlated attributes of nodes/edges in the graph. In ProGGD, our goal is to not only display interesting information about the data to the user but also make it easy to understand

[3] Apache Maven: https://maven.apache.org/

DRAFT

the discovered GGDs so that the user can also use it in their downstream tasks, for example, graph interlinking between different graphs.

Given a graph G, we consider interesting GGDs for data profiling i.e., GGDs with graph patterns and constraints according to the similarity of the attributes (differential constraints) that: (1) occur frequently on G, (2) validated according to a user-defined rate (called confidence in our system) and, (3) can maximise the total number of matched nodes and edges in the graph G (called coverage in our system).

ProGGD has three main components: GGDMiner, sHINER and Interface.

To discover GGDs from G, we use our GGDMiner discovery algorithm. The main parameters of GGDMiner that need to be set by the user through ProGGD are frequency, confidence, and the maximum size of the result set of GGDs. More details and the source code for GGDMiner are available at https://github.com/avantlab/ggdminer.

sHINER runs the validation of the input GGDs and "fix" it by generating new nodes/edges in the graph, which can also solve the entity resolution problem and perform graph interlinking between different graphs such as the OpenAIRE Graph and the Clinical Knowledge Graph. Source code of sHINER system is available at https://github.com/avantlab/gcore-spark-ggd.

The interface component is to give overall information about the graph such as statistics about the attributes and graph pattern query results. We use the Spark framework as the primary backend of ProGGD, to retrieve information about the attributes and also query graph patterns by using the G-Core language. For the user's ease of navigation, we divide functionalities of ProGGD into four main panels: (1) Metadata information and initialization, (2) Attribute information, (3) Topological Information and, (4) GGDs. Each one of these panels displays information about the graph data according to a different aspect of the graph. More details could refer to [2]. The following figure from [3] shows an overview of the ProGGD functionalities.

DRAFT

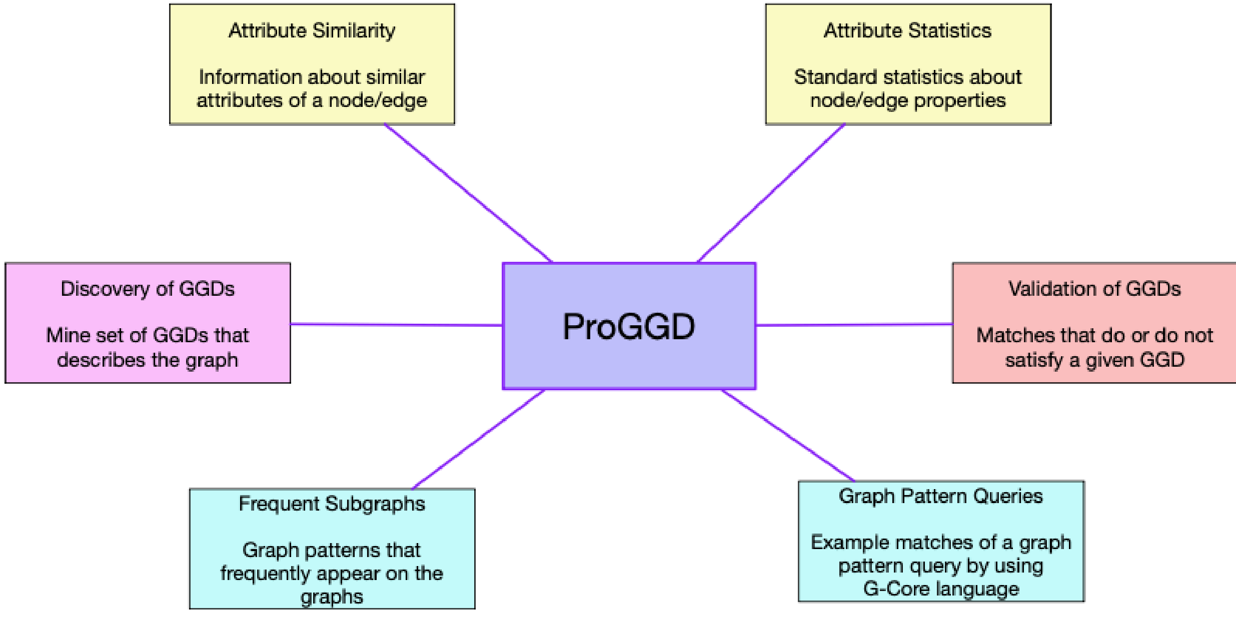

Figure 3: *The overview of the ProGGD functionalities*

4.2.3. ProGGD Pipeline

As explained above, ProGGD is a system that employs Graph Generating Dependencies to showcase information about the graph's content. We now present a demo to show how ProGGD works within the knowledge graph creation assistant tool bundle.

Firstly, we need to build and run sHINER's REST API mode. And then run the ggd-backend and ggd-interface components. Please refer to ProGGD's Github for more details: https://github.com/avantlab/proggd.

The following figures show the interactive interface of ProGGD. From the left-hand side, we can choose to showcase information of Dataset, Attributes, Graph Pattern and GGDs. The first figure is about the information of Dataset. We could choose one specific graph dataset, for example Graph Amazon. It shows the number of nodes and edges, labels and configuration parameters of differential constraints. The second figure shows the information of Attributes, and the third one shows the information of Graph Pattern. The last panel shows the overview of GGDs in the selected dataset. We visualise GGDs discovered from the knowledge graph along with the total number of source matches, target matches and the rate of validated sources.

With the information provided by ProGGD, we are able to understand the content of knowledge graph more comprehensively, which is significantly useful for downstream tasks, for example, graph interlinking.

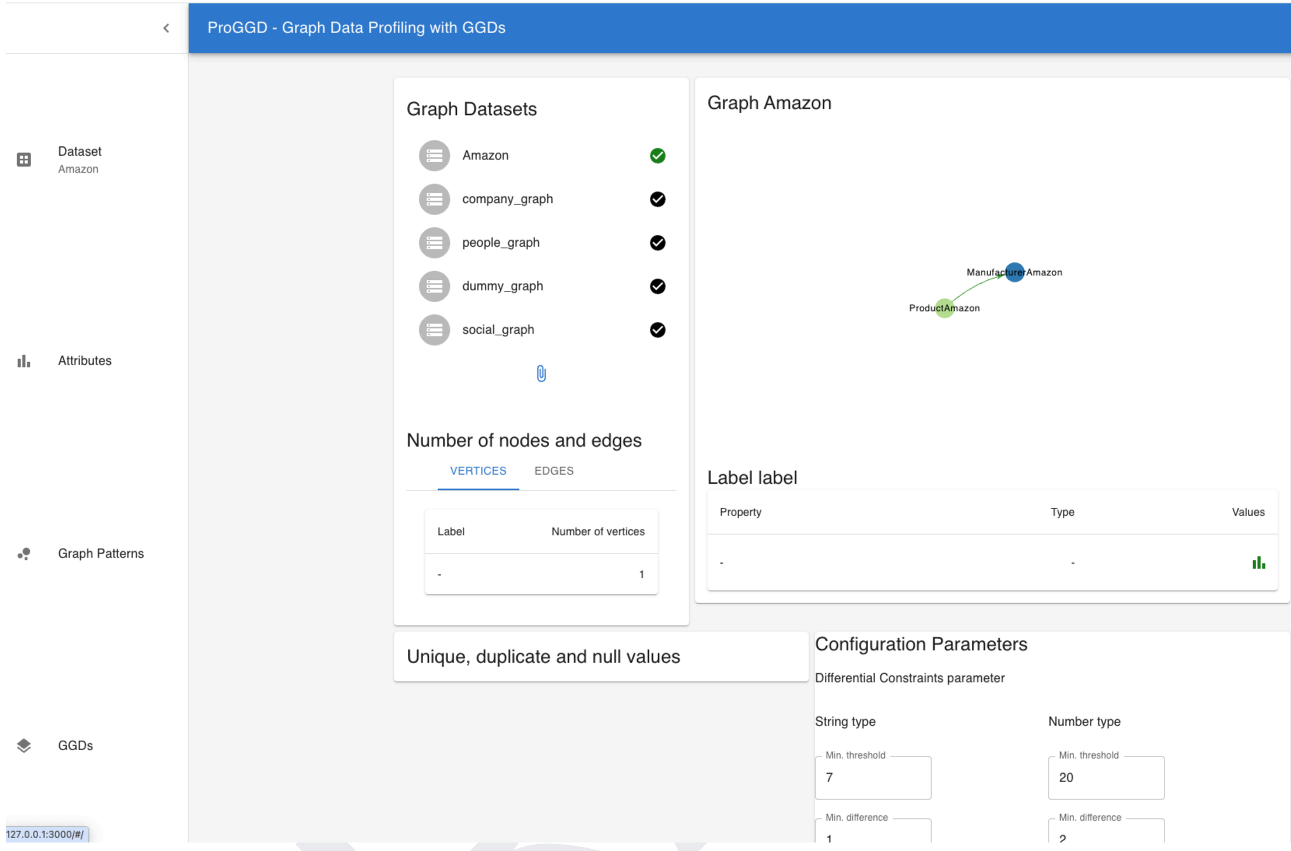

Figure 4: *ProGGD main panel*

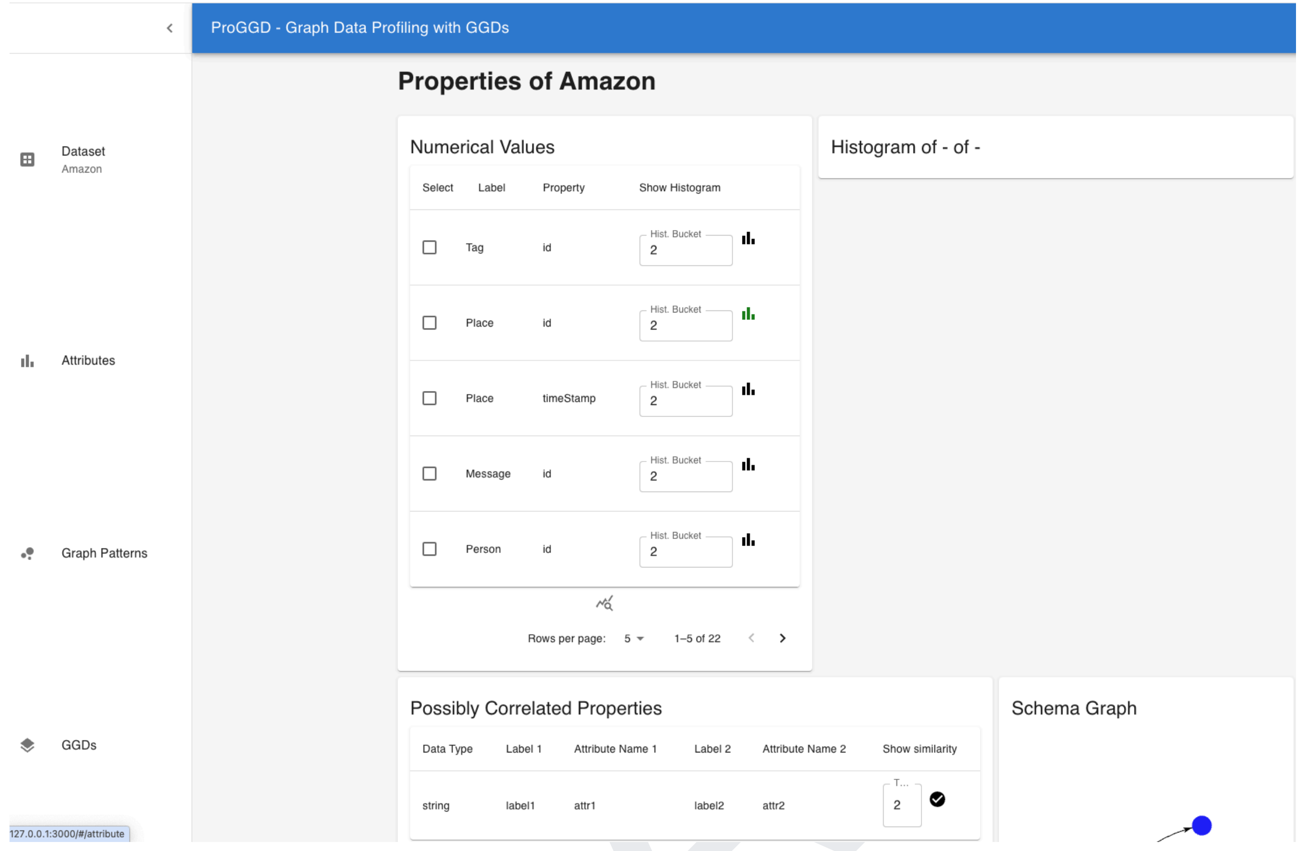

Figure 5: *ProGGD attributes panel*

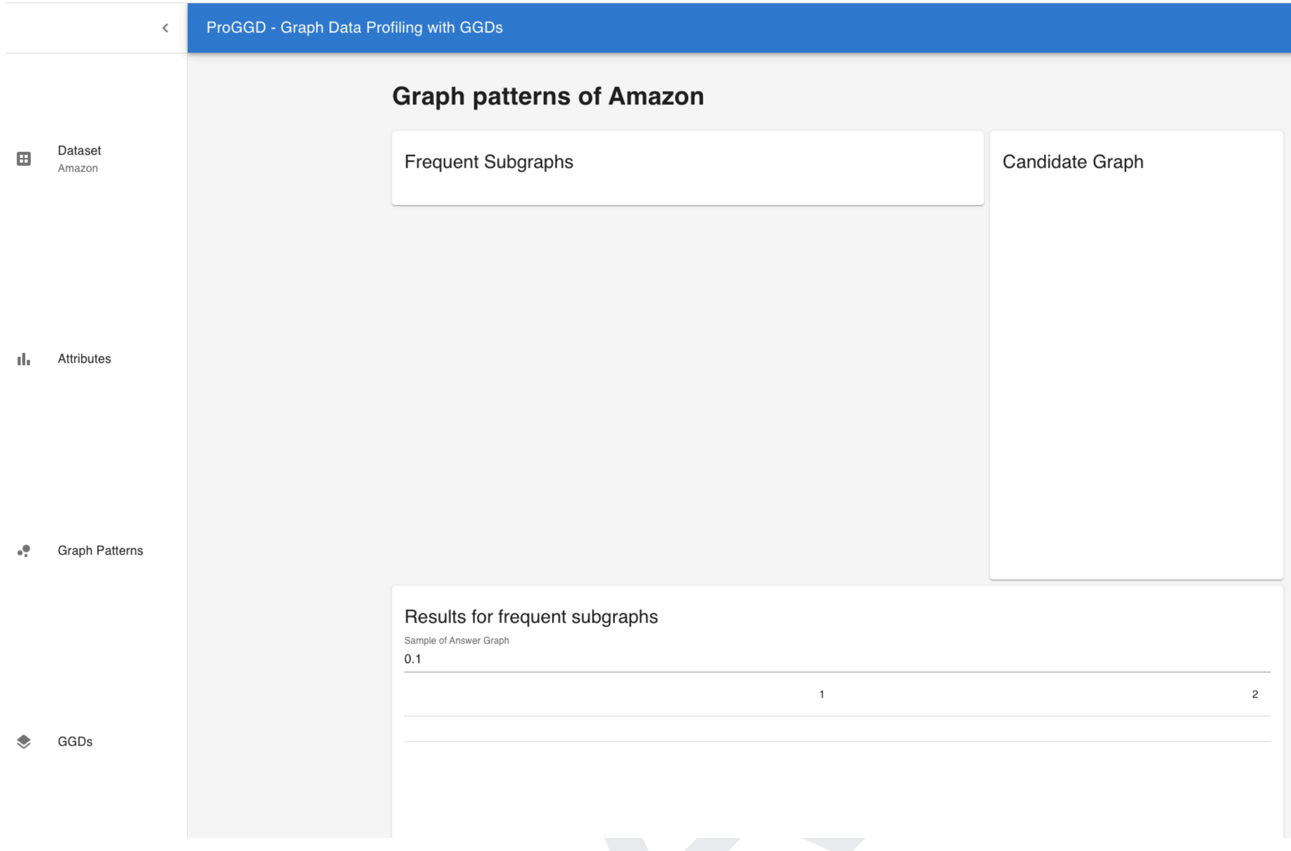

Figure 6: *ProGGD graph pattern panel*

DRAFT

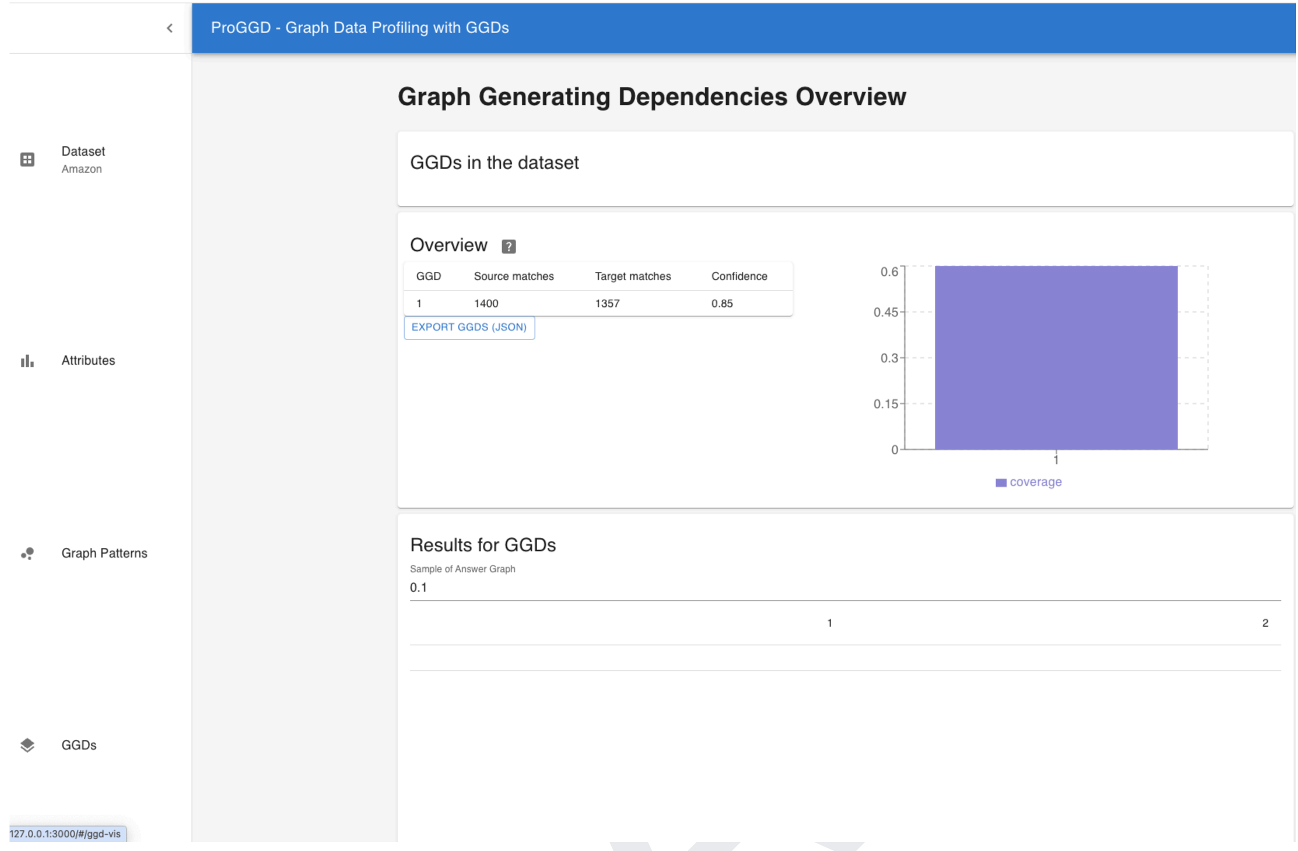

Figure 7: ProGGD graph generating dependencies panel

4.3. Data Interlinking

4.3.1. Introduction of Data Interlinking

This task is to deliver (semi-)supervised tools and methods for data (ontology, entity, relationship, property) interlinking in SciLake's SKGs. Graph Generating Dependencies, which can capture both differential and topological information, will be used to perform entity resolution and generate sameAs edge in case a GGD is violated. GGDMiner is developed to mine a set of Graph Generating Dependencies which are used for the entity resolution problem.

4.3.2. Limitations of the state of the art

Entity resolution is the task of identifying the same real-world entity from instances of data. Entity resolution can be used not only to deduplicate data but also to interlink different data sources which have the same real-world entities. The entity resolution problem in our case is to identify matching entities of heterogeneous graph data such as OpenAIRE Graph and Clinical Knowledge Graph, from which we can integrate different sources of graph data into one. Specifically, we work on a property graph and perform entity linkage to two matching nodes or relationships or graph patterns of two graph data which is significant for graph data interlinking.

Current entity resolution methods such as entity embedding based techniques and ML methods are inherently lacking in explainability and interpretability, which prevents explicit encoding of domain knowledge and interactive debugging of results. There are some logic-rule based methods, but most of them utilise attribute information to develop distance- or similarity-based measures, without consideration of topology or structure information.

GGDs can solve all the above problems, focusing on graph generation using source and target graph patterns. Source graph pattern is depicted as two disjoint interesting patterns which have potential matching entities from two different sources graph data, while sameAs edges or new patterns are generated in target graph pattern. GGDs can also express the similarity between the properties of the entities to be matched with the differential constraints. In case GGDs are violated, the target graph pattern is completed for the validation of GGDs and to link two matching entities or patterns.

4.3.3. sHINER for Entity Resolution

sHINER system is implemented for entity resolution in graph data using GGDs. The following figure shows the sHINER system architecture. sHINER has mainly two components, the G-Core language interpreter and the GGDs component. Source code of sHINER system is available at https://github.com/avantlab/gcore-spark-ggd.

DRAFT

G-Core Interpreter

The G-Core language interpreter is an interpreter of the G-Core graph query language built over Apache Spark[4]. Its implementation is open source and can be accessed in the Linked Data Benchmark Council repository. We chose to use G-Core language and its language interpreter for mainly two reasons: (1) the possibility of returning graphs as results of the queries and, (2) the G-Core interpreter is built over Apache Spark framework which gives the possibility of querying/analysing big data. One of the main characteristics of the G-Core language is that the result of a graph query is also a graph.

GGDs Component

The following figure from [3] shows the main logical component of sHINER, responsible for the interpretation and execution of the GGDs validation and graph generation algorithms used in the entity resolution.

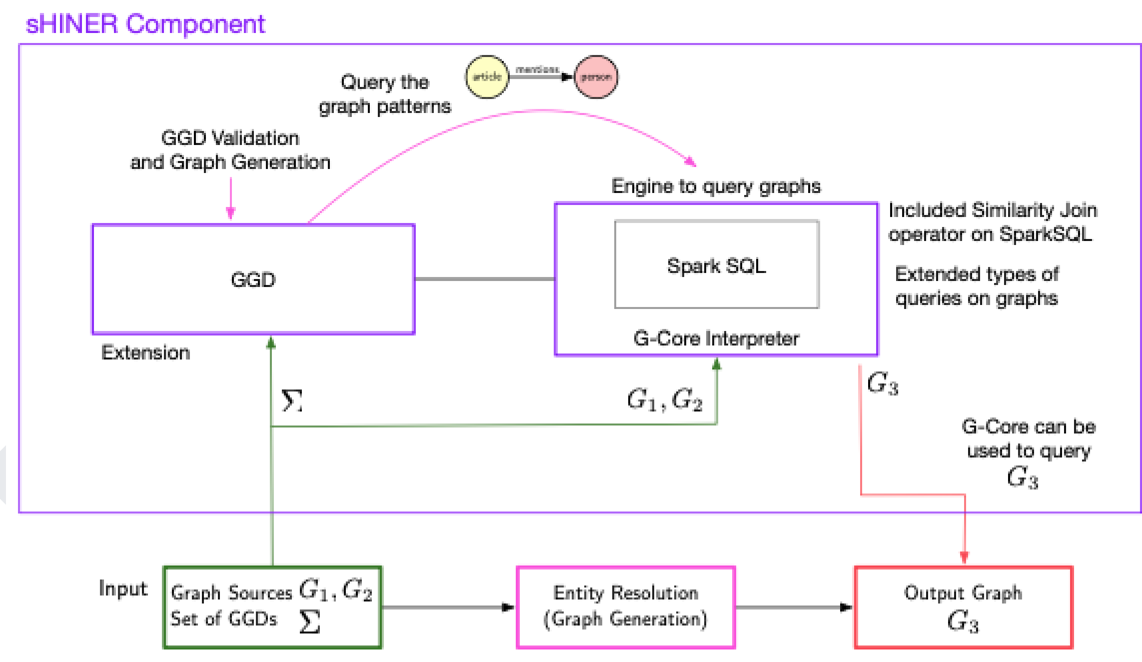

Figure 8: *sHINer system architecture*

[4] https://spark.apache.org/

4.3.4. Use Case of Cancer Pilot

The following example shows one possible way of using GGDs to express matching rules for clinical knowledge graph and cancer specific OpenAire graph.

The source graph pattern (left-hand side) includes the entity of Publication from the clinical knowledge graph and the entity of Result from the OpenAIRE Graph. The variables depicted for them are x and y which could be any character. There are two differential constraints for the source graph pattern. The first one is about the type of entity of Result, since Publication is not the only one type of Result in the OpenAIRE Graph. This constraint expresses that the attribute of type of Result should be Publication. The second constraint expresses that the edit distance between DOI of Publication from clinical knowledge graph and DOI of Publication from OpenAIRE Graph should be no more than 1. Moreover, the entity of Publication and entity of Result cannot be the same one.

The target graph pattern (right-hand side) generates the *sameAs* edge to match the entity of Publication from clinical knowledge graph and entity of Result from OpenAIRE Graph, if there is no existing *sameAs* edge between the two entities. The differential constraint of target graph pattern is empty, stating that the matching of target graph pattern is without the consideration of attributes of entities, which is different from the matching of source graph pattern.

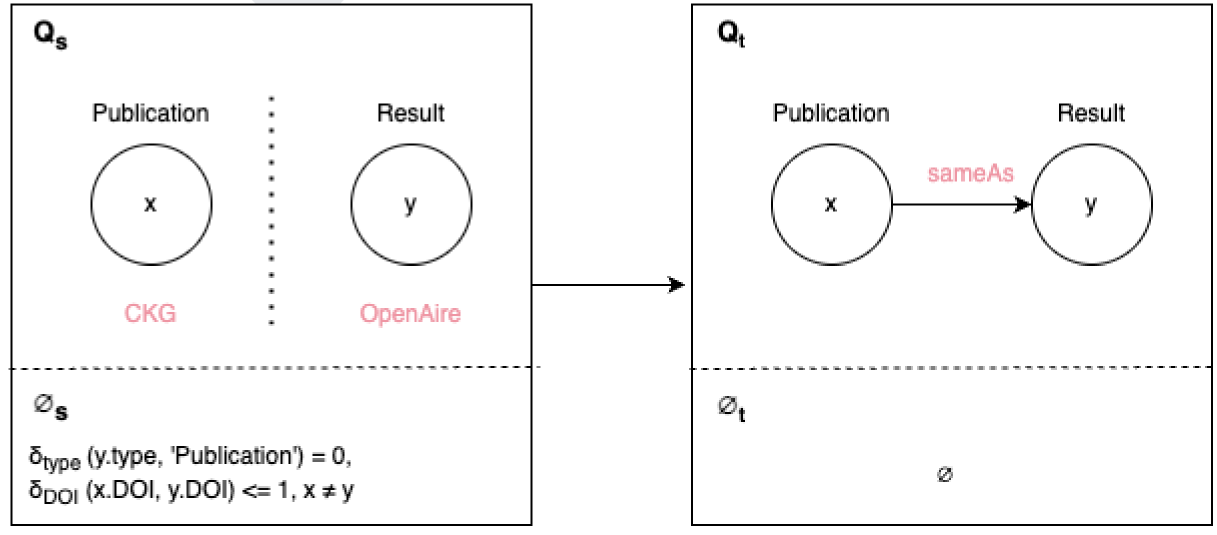

Figure 9: *Using GGDs for entity resolution in the cancer pilot*

The above GGD encodes the rules to perform entity resolution over clinical knowledge graph and cancer specific OpenAIRE Graph. To perform ER, we can add links of type *sameAs* between the matched entities in the target graph pattern. These links will be generated to validate the defined GGD.

Paper(s):

- Shimomura, L. C., Fletcher, G., & Yakovets, N. (2023). ProGGD-Data Profiling on Knowledge Graphs using Graph Generating Dependencies. ISWC 2023 Posters and Demos: 22nd International Semantic Web Conference, 2023, Athens, Greece
- Shimomura, L. C. (2024). On Graph Generating Dependencies and their Applications in Data Profiling. PhD Dissertation. Eindhoven University of Technology.
- Larissa C. Shimomura, Nikolay Yakovets, George Fletcher, Reasoning on property graphs with graph generating dependencies, Information Sciences, Volume 672, 2024, 120675, ISSN 0020-0255, https://doi.org/10.1016/j.ins.2024.120675.
- Another paper under review

Code repo(s):
https://github.com/avantlab/R2PG-DM
https://github.com/avantlab/proggd
https://github.com/avantlab/gcore-spark-ggd
https://github.com/avantlab/ggdminer

DRAFT

4.4. Data Lake search & navigation

4.4.1. Data management for SKGs

To provide a robust and efficient analytics and storage engine for scientific knowledge graphs (SKGs), we employ AvantGraph, a next-generation graph processing engine. Figure 10 showcases its architecture. AvantGraph is designed from the ground up to efficiently process complex analytics, utilising a number of novel techniques and optimizations. Specifically, AvantGraph uses advanced graph storage and indexing techniques to enable the execution of novel worst-case optimal join algorithms. Special care is taken in handling large intermediate results (IR), which often occur during complex analytical query processing. These results are significantly reduced through IR factorization. Additionally, AvantGraph features an advanced query planner and

optimizer tailored to handle graph queries involving recursion. In the context of the SciLake project, we are enhancing AvantGraph with native support for user-defined algorithms, as discussed in the following sections.

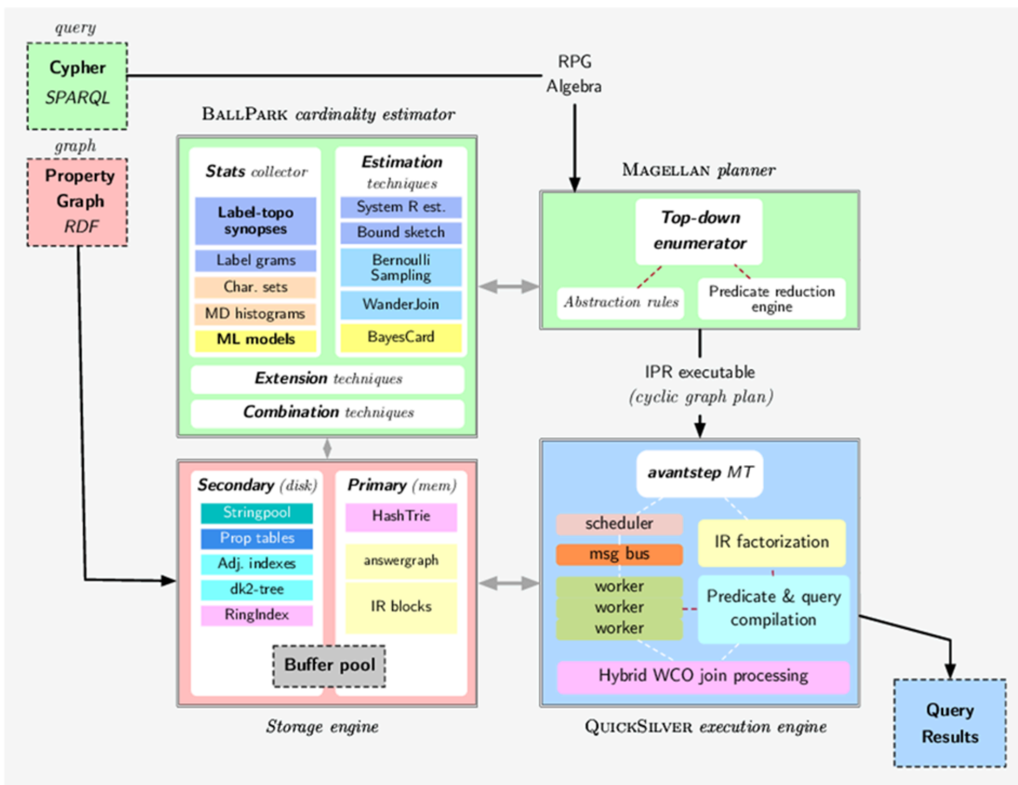

Figure 10: *Architecture of the AvantGraph graph processing engine*

4.4.2. Integrating Analytics in Databases

With the growing deployment of graph databases, more and more users are finding the data that they need to analyse stored inside a graph database. To access the data, the user sends queries in a graph query language such as Cypher or GQL. These languages excel at expressing graph patterns, yet they lack the expressive power to encode arbitrary algorithms.

For users wanting to perform complex graph analytics, a common approach is to export the contents of the graph to a text file and process it in Python instead. However, involving external tools brings with it multiple issues:

- **Data wrangling**: An interchange format must be found that the database can write and the external tool can read. Widely-used formats such as CSV (comma-separated values) have ambiguity in their definition, which leads to import failures or data loss in the conversion process.
- **Data duplication**: Exporting often requires a full copy of the graph.
- **Overhead of import/export**: For highly selective analyses, the time spent exporting and importing data can dominate the execution time of the analysis.

Database vendors have also identified the need for algorithm support in their systems. Multiple approaches to integrating algorithm support have been explored in both industry and academic systems. These invariably suffer from at least one of three key problems:

- **Flexibility**: Widely useful algorithms are difficult or impossible to express. In some cases the set of algorithms is completely fixed, so even minor variations of common algorithms require an external tool.
- **Performance**: Analytics tasks are an order of magnitude slower than they would be in a competent external tool. Often due to lack of optimization or mismatch in execution paradigm.
- **Lack of Integration**: Queries and algorithms are strictly separated and are not interoperable. In some cases algorithms even run on a separate representation of the graph, which brings back the data duplication issue from the external tool scenario.

DRAFT

Approach	Key Problems	Used by
Built-in Algorithms Library	- Fixed set of Algorithms	neo4j
Pregel API	- Performance issues - Not analysable	ArangoDB
User-defined operators	- Unsafe - Not analysable	UMBRA
Recursive CTE	- Difficult to write - Performance issues	
Procedural SQL	- Overhead - Limited analysis	ORACLE
Algorithm DSL	- Proprietary - No integration with queries	Oracle Labs PGX

Figure 11: *Different approaches to graph analytics in databases*

4.4.3. GraphAlg: Algorithm Support Done Right

GraphAlg is a language for writing graph algorithms that is designed to be tightly integrated into graph databases. GraphAlg is fully integrated with the AvantGraph database management system, allowing users to embed algorithms directly into cypher queries. The language is specifically designed to easily express graph algorithms. It has a theoretically sound foundation rooted in linear algebra, enabling aggressive optimization for maximum performance.

```
func SSSP(graph: Matrix<s, s, trop_real>,
          source: Vector<s, bool>)
          -> Vector<s, trop_real> {
  v = cast<trop_real>(source);
  for i in graph.nrows {
    v += v * graph;
  }
  return v;
}
```

Figure 12: *Single-source shortest paths in GraphAlg*

4.4.4. Use Cases for GraphAlg in Other Work Packages

Algorithm support in AvantGraph is a key building block for other work packages and pilot partners. GraphAlg will be used to power the smart impact analysis service, computing impact analyses on the OpenAIRE graph and domain-specific SKGs for pilot partners.

For example, the BIP! Toolbox today computes PageRank scores in the OpenAIRE graph using a heavily-contested cluster of more than a thousand of cores over dozens of nodes. With the OpenAIRE Graph already stored in AvantGraph, GraphAlg is ideally suited to take over this responsibility: Doing so will eliminate the current data

wrangling, reduce a complex distributed Apache Spark workflow to a simple 100 line algorithm, and allow PageRank computation of the full OpenAIRE graph on a single server.

The neuroscience and transportation research pilot partners both indicate a desire to use the smart analysis service, either through the OpenAIRE Graph or in custom SKGs. GraphAlg will therefore provide value to these partners indirectly by supporting impact analysis. Additionally, GraphAlg is available to all AvantGraph users through the regular Cypher interface, ready to use for any custom analytics workloads that pilot partners may wish to experiment with.

4.4.5. Implementation of GraphAlg

GraphAlg Language Design

We categorise GraphAlg as a domain-specific language (DSL) for writing graph algorithms in the language of linear algebra. All of the fundamental operations in GraphAlg are linear algebra operations such as matrix multiplication and element-wise product. These familiar operations not only make it easier for new users to learn the language, but they also provide a solid mathematical foundation. In contrast to other DSLs in the same space, GraphAlg has a formally defined grammar, type system and operational semantics that fully capture the language.

The formal properties of GraphAlg are achieved by reducing complex operations to a small *core* language with simple, well-defined operations. Our core language is equivalent to the MATLANG language when extended with a limited form of iteration. MATLANG is a formal language for matrix manipulation whose expressive power has been well-studied. Of particular note is that MATLANG has a predefined translation into relational algebra, which makes it an ideal starting point for integrating algorithm support into a database system.

In the design of GraphAlg we strike a careful balance between expressive power and analyzability: On the one hand the language is powerful enough to express a wide variety of algorithms, but it is also limited enough to remain amenable to aggressive optimization. We provide high-level operations that are necessary for high-performance implementations of multiple algorithms, while prohibiting operations that are known to exhibit poor performance on large graphs. Our proposed core language has only a small number of high-level operations with well-defined semantics, allowing for extensive compiler analysis and optimization.

Expression	
$e ::= (e)$	(parentheses)
$\mid M$	(matrix variable)
$\mid$ onev(e)	(one vector)
$\mid$ e.T	(transpose)
$\mid$ diag(e)	(diagonalize vector)
$\mid$ apply(lam, e_1, e_2)	(function application)
$\mid$ e_1 * e_2	(matrix multiplication)
$\mid$ $sring(l)$	(scalar literal)
$\mid$ zero($sring$)	(additive identity)
$\mid$ one($sring$)	(multiplicative identity)
$\mid$ e_1 {+,-,/,==,<} e_2	(scalar binary operator)
$\mid$ cast<$sring$>(e)	(scalar cast)

Figure 13: *Definition of expressions in the GraphAlg core language*

Integration With Regular Queries

AvantGraph allows embedding algorithms directly inside of Cypher queries. A simple example is shown below.

```
WITH ALGORITHM "
func TriangleCount(graph:Matrix<s, s, bool>) -> int {
  C = Matrix<int>(graph.nrows, graph.ncols);
  C<graph> = cast<int>(graph) * cast<int>(graph);
  return reduce(C) / 6;
}"
CALL TriangleCount()
RETURN count
```

Algorithms can be invoked directly from Cypher queries. It is also possible to provide inputs to the algorithm as subqueries, and to process query results as part of a query. This deep integration is made possible by our unified intermediate representation (IR). Queries and algorithms are both converted into the same IR, after which there is no more explicit distinction between the two. Our approach allows for maximum research of existing infrastructure for query processing: The existing optimization and execution pipeline is reused for algorithms.

Another benefit of the unified IR is that it enables *cross-optimization*: Optimizations that cross the border between query and algorithm. Because the unified IR removes all separation between queries and embedded algorithms, we can perform additional optimization that would be impossible to perform exclusively on the query or algorithm side.

Compiling GraphAlg

The compilation of GraphAlg is done in two stages: A simplification and optimization step into the core language, followed by a lowering into an extended relational algebra.

Simplification to MATLANG With Loops

Complex GraphAlg operations are replaced with simpler ones from the core language (based on MATLANG). The majority of the optimization is done after this simplification step since the remaining operations are simple and thus easier to reason about.

Lowering to Extended Relational Algebra

This lowering step converts the core language operations into the unified IR used internally by much of the AvantGraph system. The lowering step bridges any differences between the operations available in GraphAlg and those available in AvantGraph. As a result, AvantGraph can optimise and execute algorithms in the same pipelines used for queries. An example of this is the conversion of data types: GraphAlg operates on matrices, but AvantGraph operations process tuples, so the lowering step decomposes matrices into tuples.

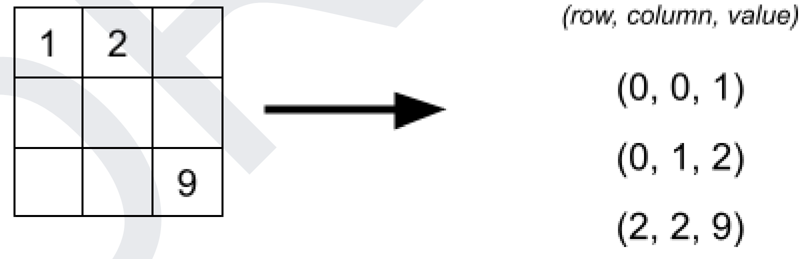

Figure 14: *An example sparse matrix converted into an equivalent set of tuples.*

Most of the GraphAlg core operations can be readily converted into standard relational algebra operators. A notable exception here is the conversion of loops. Loops are not part of the standard relation algebra definition, and most database systems do not implement them efficiently, if at all. To support this, we have extended AvantGraph with a custom loop operator that supports all GraphAlg use cases. It is part of the unified IR and is supported throughout the pipeline, including in the query optimizer.

4.4.6. Demonstration

The functionalities that were described above are demonstrated in our AvantGraph for SciLake repository. This repository serves as a practical guide for users to leverage AvantGraph for managing and analysing scientific data within the SciLake framework. The repository demonstrates the following functionalities:

- **Graph Database Integration**: It integrates AvantGraph, a graph database, to manage and query scientific data efficiently.
- **Data Ingestion**: The demo includes scripts and tools for ingesting scientific datasets into the graph database.
- **Querying Capabilities**: We provide examples of how to perform complex queries on the ingested data using AvantGraph's query language.
- **Documentation and Examples**: Comprehensive documentation and example scripts are provided to help users understand how to use the functionalities of AvantGraph within the context of SciLake.

Paper(s):
- Paper in preparation

Code repo(s):
https://github.com/avantlab/avantgraph

Demo(s):
https://github.com/avantlab/scilake-demo

DRAFT

5. Conclusions

We presented our progress in creating the first version of the Scientific Lake service as part of the SciLake project. We have set up a foundation for the service, making it easier for researchers to find and use a wide range of scientific data through our Scientific Knowledge Graphs (SKGs). Our work has focused on putting together different parts like gathering data, creating knowledge graphs, linking data together, and improving how users can search through the data lake. For each task, we provide the lists of scientific publications published under SciLake project and links to corresponding code repositories. We provide links to demonstration code, where appropriate.

Looking ahead, we are looking forward to keeping improving the service. Notable future work areas include the following. First, tailoring and scaling the data acquisition service to meet the demands of the pilots. Second, improving/tailoring the knowledge graph construction for the pilots will provide a structured representation of the data, facilitating better analysis and insights. Profiling the constructed knowledge graphs is another key area, as it will help in understanding the data's characteristics and quality. Additionally, enriching and interlinking the knowledge graphs (especially with third-party data, like the OpenAIRE Graph) will enhance their utility by integrating diverse data sources. Deeper embedding and interoperability of the AvantGraph into the SciLake data management service is also a priority, as it will streamline data management and improve data accessibility. Finally, supporting non-trivial analytical tasks for downstream SciLake services will enable more complex and insightful analyses while, at the same time, reducing the computational requirements.

DRAFT

6. References

[1] Stoica, R. A. (2019). R2PG-DM: a direct mapping from relational databases to property graphs (Doctoral dissertation, Master's thesis, Eindhoven University of Technology).

[2] Shimomura, L. C., Fletcher, G., & Yakovets, N. (2023). ProGGD-Data Profiling on Knowledge Graphs using Graph Generating Dependencies.

[3] Capobianco Shimomura, L. (in press). On Graph Generating Dependencies and their Applications in Data Profiling. [Phd Thesis 1 (Research TU/e / Graduation TU/e), Mathematics and Computer Science]. Eindhoven University of Technology.

[4] Codd EF. A relational model of data for large shared data banks. Commun ACM. 1970;13(6):377-387.

[5] Angles, R. (2018). The property graph database model. In CEUR Workshop Proceedings (Vol. 2100).

[6] Daniel Bauer, Florian Froese, Luis Garcés-Erice, Chris Giblin, Abdel Labbi, Zoltán A. Nagy, Niels Pardon, Sean Rooney, Peter Urbanetz, Pascal Vetsch, Andreas Wespi, Building and Operating a Large-Scale Enterprise Data Analytics Platform, Big Data Research, Volume 23, 2021, 100181, ISSN 2214-5796

[7] Abiteboul, S., Hull, R., & Vianu, V. (1995). Foundations of Databases. Addison-Wesley.

DRAFT