SmartDataLake

DELIVERABLE D3.2

Initial version of the HIN mining engine

PROJECT NUMBER: 825041
START DATE OF PROJECT: 01/01/2019
DURATION: 36 months

Horizon 2020

SmartDataLake is a Research and Innovation action funded by the Horizon 2020 Framework Programme of the European Union.

Dissemination Level	Public
Due Date of Deliverable	Month 18 (30/06/2020)
Actual Submission Date	24/06/2020
Work Package	WP3 - Heterogeneous Information Network Mining
Tasks	Tasks 3.1: Similarity search and exploration & Task 3.2: Entity resolution and ranking
Type	Other
Lead beneficiary	ATHENA
Approval Status	Submitted for approval
Version	1.0
Number of Pages	81
Filename	SmartDataLake-D3.2-Initial_version_of_the_HIN_mining_engine.pdf

Abstract

This report presents the currently implemented components for exploring Heterogeneous Information Networks (HINs). These components provide functionalities for entity ranking, similarity search and entity resolution. For each component, we describe its functionalities, present its API, as well as installation and usage instructions, and conduct an experimental evaluation to assess its performance.

History

Version	Date	Reason	Revised by
0.1	04/05/2020	Table of Contents	Dimitris Skoutas
0.2	01/06/2020	First draft for internal review	Dimitris Skoutas
0.3	22/06/2020	Revised draft	Dimitris Skoutas
1.0	24/06/2020	Final version for submission	Dimitris Skoutas

Author List

Organization	Name	Contact information
ARC	Dimitris Skoutas	dskoutas@athenarc.gr
ARC	Thanasis Vergoulis	vergoulis@athenarc.gr
ARC	Kostas Patroumpas	kpatro@athenarc.gr
ARC	Serafeim Chatzopoulos	schatz@athenarc.gr
ARC	Alexandros Zeakis	azeakis@athenarc.gr
ARC	Georgios Chatzigeorgakidis	gchatzi@athenarc.gr
TUE	Larissa Shimomura	l.capobianco.shimomura@tue.nl
TUE	Nikolay Yakovets	N.Yakovets@tue.nl
TUE	George Fletcher	g.h.l.fletcher@tue.nl
TUE	Hamid Shahrivari	h.shahrivari.joghan@tue.nl
TUE	Odysseas Papapetrou	o.papapetrou@tue.nl
SDATI	Roberto Santoro	santoro@spaziodati.eu

Executive Summary

Heterogeneous Information Networks (HINs) are graphs comprising different types of nodes (*entities*) and edges (*relationships*). In this report, we present the components we have implemented for entity exploration in HINs. These components provide functionalities for: (a) entity ranking, (b) similarity search and join, and (c) entity resolution. For each developed component, we describe its supported functionalities and API, we list installation and usage instructions, and we present the results of an experimental evaluation.

In Section 1, we provide a summary of the developed components and the scope of this report in the context of the SmartDataLake project.

In Section 2, we present the components for entity ranking, which include HMiner and BRS. The former provides functionalities for ranking entities represented as nodes in a HIN. Each entity is assigned a ranking score, which is based on the centrality of the respective node in the network. In turn, the centrality score of a node is based on one or more user-defined metapaths in the HIN. On the other hand, the BRS component offers functionalities for identifying the top-k geospatial regions according to a user-defined scoring function. These regions are modelled as rectangles of user-defined width and height. In this case, ranking is based on the geolocation of entities rather than their position in the network.

In Section 3, we present the components for similarity search and join, which include SimJoin, SimSearch, and SimSearchTS. SimJoin focuses specifically on set similarity joins, targeting entities that can be described by sets of keywords, tags or, generally, any kind of tokens. This functionality is needed when dealing with entities having textual attributes. SimSearch offers functionalities for retrieving the top-k most similar entities to a given query entity. It supports entities with different types of attributes, including textual, numeric and spatial. Finally, SimSearchTS is designed for similarity search over time series. In particular, it operates on collections of time-aligned series, and can identify all pairs of time series having a similar subsequence, based on user-defined thresholds for distance and duration.

In Section 4, we present the component developed for entity resolution in HINs, called sHINER. This component identifies and links nodes in the HIN that may represent the same real-world entity. Its functionality is based on the novel concept of graph generating dependencies, which extend the idea of functional dependencies, conditional functional dependencies and differential dependencies from relational data to graph data.

Finally, Section 5 summarizes and concludes the report.

Abbreviations and Acronyms

API	Application Programming Interface
BRS	Best Region Search
CSV	Comma-Separated Values
DBMS	Database Management System
DCG	Discounted Cumulative Gain
DSV	Delimiter-Separated Values
GGD	Graph Generating Dependency
HIN	Heterogeneous Information Network
JDBC	Java Database Connectivity
JSON	JavaScript Object Notation
LRU	Least Recently Used
NDCG	Normalized Discounted Cumulative Gain
PAA	Piecewise Aggregate Approximation
RDBMS	Relational Database Management System
SQL	Structured Query Language

Table of Contents

1. Introduction

1.1. Overview

In this report, we present the components we have developed for exploring entities in Heterogeneous Information Networks (HINs). A HIN is a graph that comprises different types of nodes (entities) and edges (relationships) [1]. Figure 1 presents an illustrative example of a HIN containing news articles (a_1, a_2), persons (p_1, p_2, p_3, p_4), organizations (o_1, o_2, o_3) and locations (l_1, l_2, l_3). Due to their ability to represent different types of nodes and edges, HINs have rich semantics and are appropriate for representing heterogeneous information in data lakes, where entities come from different sources and have different attributes.

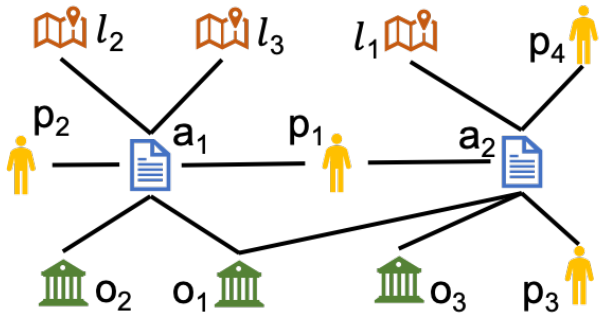


Figure 1: Illustrative example of a HIN containing news articles, persons, organizations and locations.

The components we have developed for entity exploration in HINs address three fundamental functionalities:

- *Entity ranking*. A data lake may contain millions of entities of various types. Entity ranking is essential for prioritizing attention. It provides guidance regarding which entities to focus on, especially when the user is not already familiar with the data. These entities could then be used as starting points for further exploration.
- *Similarity search*. Navigating the data lake often involves starting from some initial entity (e.g., a highly ranked one or one otherwise selected by the user) and asking the system to identify and retrieve the most similar entities to it. This is addressed by similarity search.
- *Entity resolution*. Data about the same real-world entity often comes from various sources. However, since the data sources in a data lake are typically independent from each other, there exist no common identifiers which could allow to easily identify these references to the same real-world entity. Thus, entity resolution aims at discovering and linking nodes in the network that represent the same entity.

Table 1 summarizes the developed components. Detailed descriptions are provided in Sections 2, 3 and 4, including a presentation of functionalities, explanation of their API, installation and usage, and experimental results.

Table 1: Summary of developed components.

Component	Short Description
Entity Ranking	
HMiner[1]	This component ranks entities represented as nodes in a HIN. The ranking of an entity depends on its centrality in the network. In turn, the centrality is based on one or more metapaths defined by the user, thus allowing to capture and combine different ranking criteria. [2]
BRS[2]	This component ranks geospatial regions represented as rectangles with user-defined width and height. The ranking is based on a user-specified scoring function computed over the contents of each candidate region. Diversification is applied over the candidate regions to avoid overlapping results in the top-k answers. [10]
Similarity Search	
SimJoin[3]	This component discovers pairs of similar entities based on set-valued attributes. It can identify all pairs with similarity score above a threshold, as well as the k-nearest neighbours of each entity or the top-k most similar pairs in the entire collection. It also supports fuzzy matching between set elements.
SimSearch[4]	This component retrieves the k most similar entities to a given query entity. It supports different types of attributes, including set-valued, numeric and spatial. The overall similarity score is computed as an aggregate of the individual similarity scores, after being appropriately scaled and weighted. [2][3]
SimSearchTS[5]	This component supports similarity search and join over time series data. Given a collection of time series, it discovers all pairs of time-aligned subsequences that have distance below a user-defined threshold and duration above a user-defined threshold. [4]
Entity Resolution	
sHINER[6]	This component provides functionalities for Entity Resolution in HINs. Specifically, it relies on the concept of Graph Generating Dependencies to generate *same-as* links between network nodes that represent the same real-world entity according to specified conditions. [5]

[1] https://github.com/smartdatalake/HMiner

[2] https://github.com/smartdatalake/best_region_search

[3] https://github.com/smartdatalake/simjoin

[4] https://github.com/smartdatalake/simsearch

[5] https://github.com/smartdatalake/time_series_explorer

[6] https://github.com/smartdatalake/gcore-spark-ggd

1.2. Role in the project

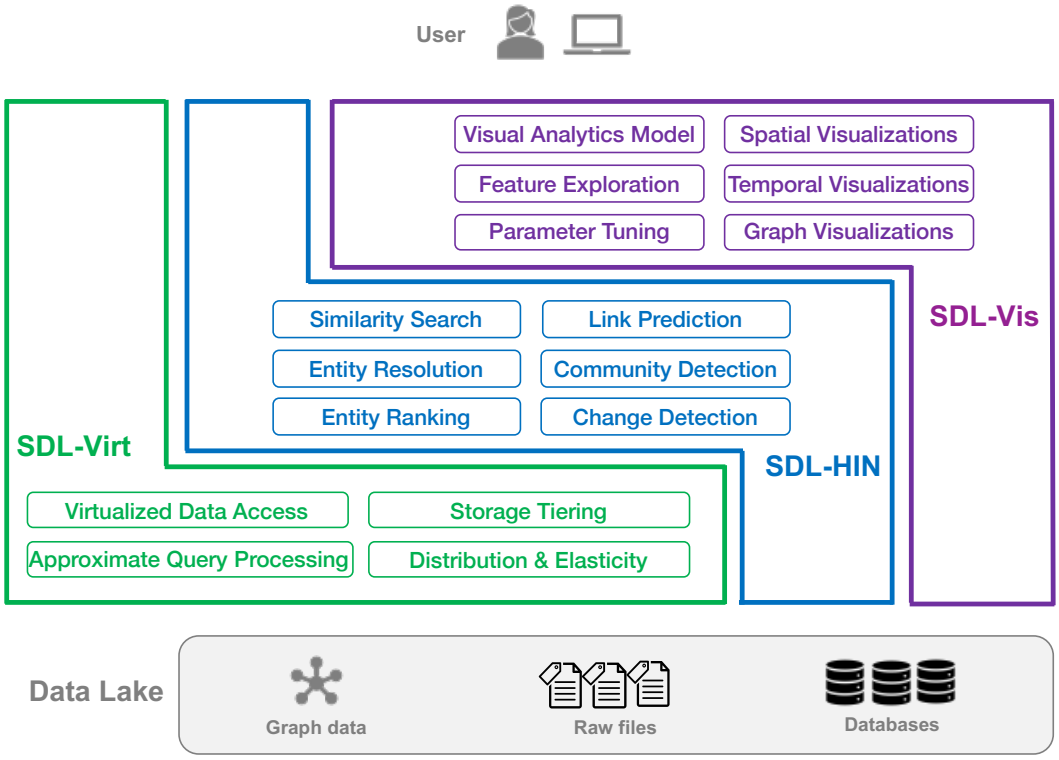


Figure 2: The three layers of SDL: SDL-Virt, SDL-HIN, and SDL-Vis. This deliverable covers SDL-HIN, and in particular Entity Ranking, Similarity Search and Entity Resolution.

The work presented in this report covers the functionalities for entity ranking, similarity search and entity resolution that are part of the SDL-HIN layer (see Figure 2 and Deliverables D1.2: "System architecture" and D1.3: "Similarity search, entity resolution and ranking"). These components emphasize on *exploration* and have been our main focus during the first period of the project. They also lay the foundation for the rest of the components of SDL-HIN, i.e., link prediction, community detection and change detection, which emphasize on *mining*, and will be the main focus during the second period.

This deliverable builds upon what has been presented in D3.1. Specifically, D3.1 has introduced the main concepts and our approaches for addressing the tasks of entity ranking, similarity search and entity resolution. Here, in D3.2, we describe the implemented components, listing their functionalities and interfaces, as well as presenting an experimental evaluation of their performance. The connections between these components and those of the data virtualization layer (SDL-Virt) as well as the visual analytics engine (SDL-Vis) will be described in Deliverable D1.4: "Initial integrated system".

2. Entity Ranking

In this section, we describe the two components we have implemented for entity ranking, namely HMiner and BRS. The former ranks entities represented by nodes in a HIN, while the latter computes the top-k spatial regions according to the entities enclosed in them.

2.1. Entity ranking in HINs

2.1.1. Implemented functionalities

The HMiner component provides functionalities for ranking entities represented as nodes in a HIN. In its current implementation, it operates in two stages. First, given a metapath, it creates a view of the HIN as a homogeneous graph. Then, it performs ranking of entities in this homogeneous graph. For the latter, traditional ranking algorithms can be applied to identify the most important (i.e., central) nodes in the graph; in our implementation, we use PageRank [6]. Notice that more than one metapath can be provided as input, which is useful when the user wants to rank entities according to multiple criteria, represented by paths with different semantics. In that case, the individual rankings for each metapath are first computed, and then the results are aggregated using a rank aggregation algorithm [15].

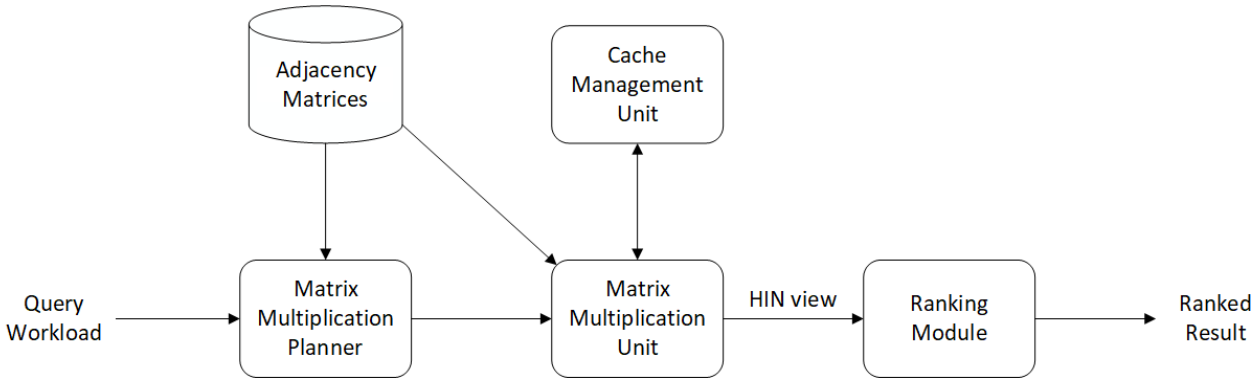


Figure 3: Overview of HMiner operation.

Producing the view of a large-scale HIN based on a particular metapath is computationally expensive. Specifically, the metapath-based view can be computed by matrix multiplication between the adjacency matrices of the relationships involved in the metapath, which is a computationally expensive process. To make matters worse, in a data exploration scenario, a data scientist typically needs to perform a series of such operations for multiple metapaths.

To address this challenge, HMiner is designed to support such metapath query workloads. In particular, our approach is based on the observation that metapaths in a data exploration workload are typically expected to have overlaps. HMiner identifies and exploits such overlaps among the

metapath queries in the workload, avoiding redundant computations. Moreover, its performance is further enhanced using sparse matrix representations.

An overview of HMiner's operation and main components is shown in Figure 3. The *Matrix Multiplication Planner* receives each metapath query of the workload and translates it to the corresponding sequence of adjacency matrix multiplications of the relationships it involves. As matrix multiplication is an associative operation, there are multiple orderings of these matrix multiplications that produce the same results but have different computational cost. So, the goal is to re-organize matrix multiplications to minimize the cost. The best multiplication plan of the multiplication chain $A_1 A_2 \dots A_n$ can be found following a dynamic programming approach (similar to the one presented in [7]). The basic idea is to construct the optimal parenthesization based on the optimal sub-parenthesizations, minimizing a cost function F_{cost} with respect to a split point k:

$$C(i,j) = min_{i \le k < j}\{C(i,k) + C(k+1,j) + F_{cost}(A_{i,k}, A_{k+1,j})\}$$

where $C(x,y)$ is the optimal multiplication cost to compute the sub-chains from index x to y and $F_{cost}(A_{i,k}, A_{k+1,j})$ is the cost function for multiplying the two matrices that result from sub-chains $A_{i,k}$ and $A_{k+1,j}$. As most adjacency matrices are sparse, i.e. contain a few non-zero elements compared to their size, we exploit sparse matrix representations to speedup multiplication and reduce the overall memory footprint. For this reason, we incorporate the sparse cost approximation model proposed in [8] into the dynamic programming formula. Our sparsity-aware multiplication planning requires an additional $n \; x \; n$ matrix to hold the densities of the intermediate multiplication results.

The best plan is forwarded to the *Matrix Multiplication Unit* for execution. Before proceeding with the involved matrix multiplications, this unit communicates with the *Cache Management Unit* to check if any part of the required computations is already available in a special cache memory that is dedicated to store frequently occurring intermediate results. In case there is a relevant cached item, the corresponding entry is loaded from the cache and is directly exploited in the current computation. Furthermore, the intermediate results that emerge during the multiplication are sent to the Cache Management Unit that decides if they need to be stored. The Cache Management Unit exploits a dedicated data structure to track the metapath query overlaps and their frequencies. This is used to cache intermediate results that correspond to frequent query overlaps. The intuition is that frequent query overlaps are more likely to continue to occur in future queries. Hence, we choose to cache those results. In addition, we also cache the final result of each query, so it can be used in the case that the exact same query appears again in the future. Based on the described technique, our algorithm chooses to cache only the most important parts of the metapath queries avoiding inserting into the cache all intermediate results that would lead to numerous cache evictions. The result of this step is a sparse matrix that represents all metapath instances and constitutes the metapath-based HIN view.

The HIN view is then passed to the *Ranking Module* that performs the actual entity ranking algorithm. In this step, any ranking algorithm for homogeneous networks can be applied on the HIN view. As mentioned earlier, in the current implementation we employ PageRank.

2.1.2. API

As explained above, HMiner first transforms the HIN into a homogeneous graph and then performs entity ranking on the derived network. Since, besides ranking, the first step can be used in other HIN analyses as well, it is exposed to the user through the API as a stand-alone operation too. All required input parameters are provided in a JSON configuration file. Its schema along with short descriptions for each of its fields are given below:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "indir": {
      "type": "string",
      "description": "The directory that contains data files for node
attributes"
    },
    "irdir": {
      "type": "string",
      "description": "The directory that contains data files for relations"
    },
    "algorithm": {
      "type": "string",
      "description": "The algorithm to be used, one of { HRankSeq, HRankDynP,
CBS1, CBS2, OTree }"
    },
    "hin_out": {
      "type": "string",
      "description": "The output directory to store the transformed network"
    },
    "qf": {
          "type": "string",
          "description": "A file with the query workload to be executed"
    },
    "cache_size": {
          "type": "number",
          "description": "The size of the cache in MB"
    },
    "cache_policy": {
          "type": "string",
          "description": "The cache policy to be used; one of { LRU, PGDS }"
    },
```

```
    "dynamic_optimizer": {
          "type": "string",
          "description": "The heuristic to be used for dynamic programming
planning; one of { Sparse, Dense }"
    },
    "ranking": {
      "type": "object",
      "description": "Ranking specific parameters (optional); if present
entities of the HIN view will also be ranked based on these parameters"
      "properties": {
        "ranking_out": {
          "type": "string",
          "description": "The output directory to store the final ranking list"
        },
        "pr_alpha": {
          "type": "number",
          "description": "The restart probability (or alpha) of the Pagerank
method"
        },
        "pr_tol": {
          "type": "number",
          "description": "The error tolerance of the Pagerank method"
        },
        "threshold": {
          "type": "integer",
          "description": "A cut-off threshold to exclude paths that occur less
times than this parameter (optional)"
        }
      }
    }
}
```

As shown above, the JSON configuration file consists of two basic blocks, as described next.

Basic configuration block. This contains information regarding the input and output directories as well as the algorithm and cache configuration. In particular, HMiner requires two input directories containing files with node attributes ("indir" field) and relations ("irdir" field) respectively.

File containing node attributes. These are tab-separated files containing all node attributes. The first line is the header. It contains attribute names with "_n" or "_s" suffixes denoting numeric or alphanumeric attributes, respectively. The first column contains an incremental integer identifier starting from 1, with the name "id_n". By convention, these files should be named according to the first letter of the entity type they represent. For example, the file that contains the attributes for

node type “Author” should be named A.csv. An example of a file containing node attributes is the following:

```
id_n  name_s    surname_s
0     Makoto    Satoh
1     Ryo       Muramatsu
...
```

File containing relations. These are tab-separated files needed to construct the relations among nodes. These files contain two columns, the source and target identifiers respectively and should be sorted based on the first column. They do not contain a header and they should be named according to the source and target node types. For example, the file with the relations between node types “Author” and “Paper” is named AP.csv. An example of a file containing node relations is the following:

```
0     1
0     2
0     3
...
```

The “qf” parameter indicates the query workload to be executed. The query workload file adopts the jsonl format as follows:

```
{ "metapath": "APPA", "constraints": { "P": "year >= 2000" } }
{ "metapath": "APA", "constraints": { "P": "year > 2010" },
"src_field": "name" }
```

Each line corresponds to a specific metapath query and should contain at least the “metapath” and “constraint” fields. Additionally, if the “src_field” is given, the specified column is appended in the ranking output file.

Ranking block. This contains ranking specific parameters such as the restart probability of the random walk (also known as alpha parameter or damping factor), the tolerance of the error in the results and a cut-off threshold in case we want to exclude paths occurring a few times. If this block is absent from the configuration file, only the HIN views are saved in the output folder without performing ranking.

2.1.3. Installation and usage

Next, we present the steps for installing and using HMiner. HMiner is implemented in C++ and is publicly available on GitHub[7] under Apache Licence 2.0. The code has been developed and tested using the g++ 7.5.0 compiler.

HMiner has the following dependencies to external libraries that are downloaded automatically:

- Eigen[8] is a C++ template library for efficient Linear Algebra data structures and algorithms. It is used to compute sparse matrix multiplications.
- CSV/DSV Filter[9] is an efficient and fast in-memory library that enables evaluation of complex expressions over a CSV or DSV file. HMiner uses this library to load from the data files only the nodes that satisfy the user-defined constraints.
- JSON Parser[10] is a simple C++ parser for JSON objects that is used to parse the configuration file provided by the user.

To install and use HMiner, the following steps should be performed:

Step 1. Clone the repository including its submodules:

```
git clone --recursive https://github.com/smartdatalake/HMiner.git
```

Step 2. Open a terminal inside the root directory and update the submodules:

```
git submodule update --remote
```

Step 3. Build the project using the makefile:

```
make
```

Step 4. Edit the JSON file including the configuration parameters with a text editor. An example configuration file is the following:

```
{
    "indir": "./data/DBLP_subset/nodes/",
    "irdir": "./data/DBLP_subset/relations/",
    "algorithm": "OTree",
    "hin_out": "./data/out/",
    "qf": "./data/DBLP_subset/queries.txt",
    "cache_size": 4096,
    "cache_policy": "PGDS",
    "dynamic_optimizer": "Sparse",
    "ranking": {
        "ranking_out": "./data/out/",
        "pr_alpha": 0.5,
        "pr_tol": 0.00000000001,
        "threshold": 2
```

[7] https://github.com/smartdatalake/HMiner
[8] http://eigen.tuxfamily.org/index.php
[9] http://www.partow.net/programming/dsvfilter/index.html
[10] https://nlohmann.github.io/json/

```
    }
}
```

This JSON configuration file ranks the authors of a subset of the DBLP dataset based on the metapath APPA and considering papers published after 2000.

Step 5. Invoke HMiner with the configuration file created in the previous step:

```
./run –c config.json
```

Finally, the result is stored in TSV format in the file specified in the "ranking_out" field in the configuration file. A sample output file for the configuration provided above is the following:

```
7412            Geoffrey E. Hinton              0.00231373693338
7025            Andrew Zisserman                0.00228635903207
2107            Ian T. Foster                   0.00169658948736
4231            Wil M. P. van der Aalst         0.00134563441064
8636            Erhard Rahm                     0.00133491710831
```

It contains the author names and their respective ranking scores. All ranking scores add up to 1 and authors are sorted based on their ranking scores.

2.1.4. Experimental evaluation

We focus our experimental evaluation on HIN transformation as this step is crucial for many HIN analyses and raises challenges in the scenario where a data scientist exploring the HIN poses multiple metapath queries to reveal latent knowledge related to different entities of interest. First, we present the datasets and methods used in our experimental evaluation. Then, we compare HMiner against the state-of-the-art single query methods. Finally, we showcase the performance of our approach against three baseline methods adjusted to support query workloads.

For our experiments, we utilised two real-world HINs from publicly available data, namely DBLP and GDELT. They are briefly described below:

- DBLP is a HIN constructed from AMiner's DBLP Citation Dataset[11] enriched with research topics from the CSO Ontology[12]. The CSO Classifier[13] was used to retrieve article topics based on their title and abstract. The final HIN contains the following entity types: Papers (3,079,008), Authors (1,766,548), Venues (5,079) and Topics (3,294). It also contains the following types of relationships: PP (25,166,994), AP/PA (9,476,165), VP/PV (2,572,308) and TP/PT (26,044,698).
- GDELT is a HIN constructed from the news articles from the GDELT project[14] published during 15-21 January 2018. Apart from Articles (664,066), the GDELT HIN also contains Organisations (190,669), Persons (266,516), Sources (17,001), Themes (5,386) and

[11] https://www.aminer.org/citation
[12] https://cso.kmi.open.ac.uk/home
[13] https://cso.kmi.open.ac.uk/cso-classifier
[14] https://www.gdeltproject.org/

Locations (44,698). Furthermore, it includes the following relationship types: OA/AO (1,124,614), PA/AP (1,223,093), SA/SA (664,049), TA/AT (1,745,539) and LA/AL (723,006).

To the best of our knowledge, there are no available query workloads for these datasets, so we constructed synthetic query workloads for our evaluation. These simulate multiple data scientists that explore different aspects of particular entities. Each of them poses a series of metapath queries for a given entity that can be specified with the appropriate constraint in the metapath query. Therefore, multiple *query sessions* are generated, each one consisting of a series of metapath queries that share the same constraint. We generate such query sessions to imitate the aforementioned behavior of the data scientists as follows: at each step, we randomly select a new metapath and either we pick a new constraint (and therefore start a new session) with *session restart probability p* or we continue using the same constraint with probability 1 – p. Intuitively, small value of the session restart probability result in fewer but larger sessions in a query workload, while larger values generate more sessions of fewer queries. The generated query workloads contain 200 metapath queries with randomly selected metapaths of length 3 to 5.

For our experimental evaluation we consider the following methods:

- HRank: the method presented in [7] with its proposed configuration.
- Neo4j[15]: one of the most popular graph databases. Metapath queries are converted to Neo4j's native query language Cypher using the parameter's syntax[16] to express given constraints.
- Baseline (BS): a variant of HRank adapted to use sparse matrix representations.
- Cache Baseline 1 (CBS1): an adaptation of BS that stores the final result of each metapath query in an LRU cache, benefitting from repetitive queries in a query workload.
- Cache Baseline 2 (CBS2): a more advanced baseline method that apart from caching the final result, also caches intermediate results of matrix multiplications.
- HMiner: our approach that exploits frequent query overlaps in a query workload.

All experiments ran on an Intel Xeon E5-2609 CPU machine with 1.7GHz clock frequency and 64GB of RAM on Ubuntu 18.04 LTS operating system. The Neo4j 4.0.4 Community Edition was used and was further configured to utilize 40GB of main memory.

First, we examine the performance of our approach against the two state-of-the-art approaches for single metapath query evaluation, HRank and Neo4j. For this experiment, we consider a DBLP subset of ~30k nodes and ~115k edges and a GDELT subset of ~35k nodes and 167k edges as HRank and Neo4j face memory issues for the full datasets. Figure 4 summarizes the results in terms of execution time per query in logarithmic scale. We observe the same behavior in both datasets; HMiner clearly outperforms its competitors as it is one order of magnitude faster than Neo4j. Meanwhile, HRank is the slowest method of the three being nearly 3 orders of magnitude slower than Neo4j. For this reason, in the following experiments, we only consider the three enhanced baselines BS, CB1 and CB2 which utilize sparse matrix representations and improve upon HRank.

[15] https://neo4j.com/

[16] https://neo4j.com/docs/cypher-manual/current/syntax/parameters/

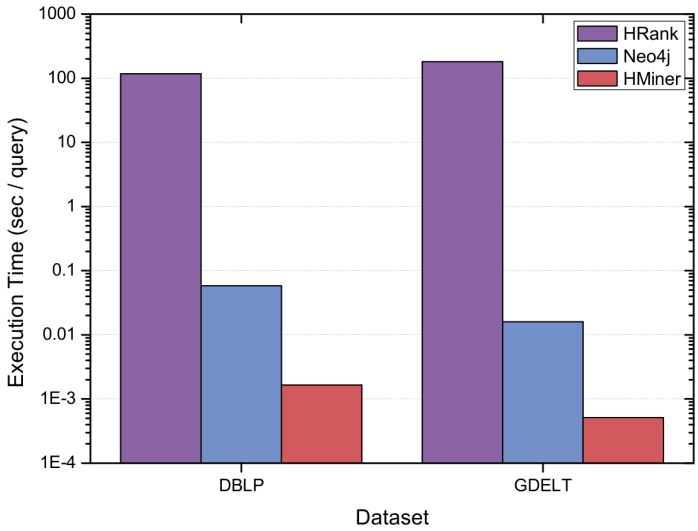
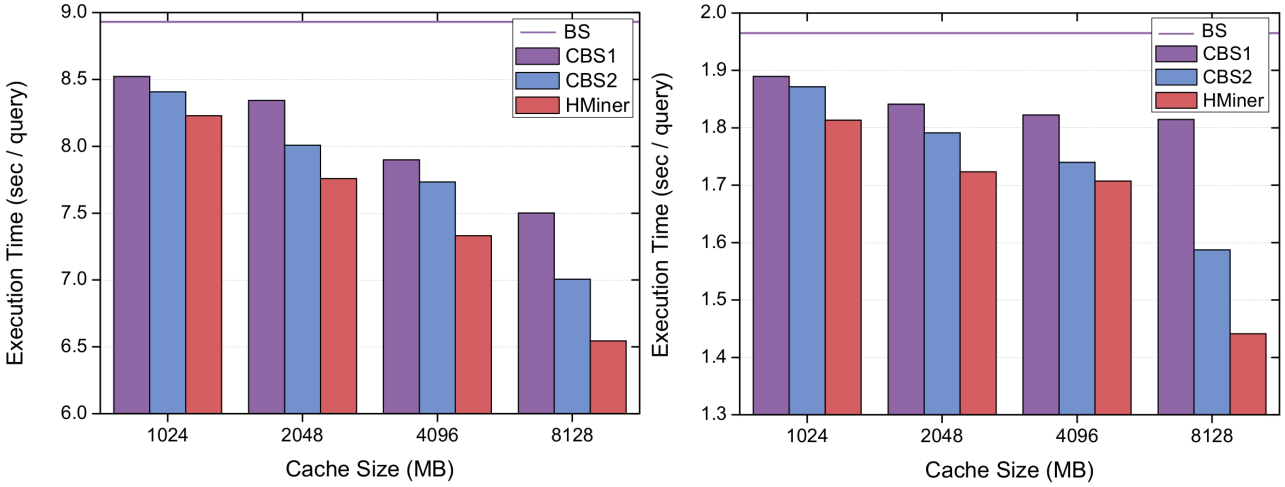


Figure 4: Evaluation against HRank and Neo4j for single metapath queries.

In the next experiment, we examine the performance of HMiner against the enhanced baselines in terms of execution time per query while varying the available cache size. Figure 5 presents our findings. The same behavior is observed for both datasets. BS does not take advantage of a cache memory therefore its execution time is independent from the available cache size. For this reason, we visualize BS with a straight line in this experiment. It is evident that BS is slower than all cache-based approaches, observing a growing performance difference while the cache size increases.



(a) DBLP (b) GDELT

Figure 5: Evaluation against competitors with varying cache size.

CBS1 achieves faster execution times than BS by exploiting repetitive metapath queries. Its execution time improves slightly as cache size increases. In the GDELT dataset, CBS1 time nearly falls for cache sizes larger than 2048MB meaning that there are no repetitive metapath queries that can be further exploited in the query workload. However, CBS2 and HMiner manage to

improve upon CBS1, both in DBLP and GDELT, by additionally caching intermediate matrix multiplication results.

Overall, HMiner outperforms its competitors in all examined configuration settings. Contrary to CBS2, that caches all intermediate results of matrix multiplications, HMiner caches intermediate results that correspond to frequent query overlaps. That way, it selectively chooses to cache the most important intermediate results avoiding inserting all intermediate results. Its advantage is more prominent for cache size of 4086MB in the DBLP dataset and 8128MB in both datasets.

2.2. Ranking of spatial regions

2.2.1. Implemented functionalities

The amount of geospatial data generated from social networks, sensors, smartphone applications, tracking devices and so on is constantly increasing. Hence, analysing big geospatial data at scale is of paramount importance for numerous applications in various areas such as geomarketing, mobile advertisement, urban planning, tourism and logistics. In many cases, the analysis involves identifying areas where the intensity of a studied phenomenon is maximized (e.g., hot spots of user check-ins or commercial activities). Similar requirements arise in data lakes, where spatial data is frequently combined with other types of data. A particular example arising in SmartDataLake includes combining addresses of companies with their balance sheets, number of employees, and number of clients, to identify hot spots in terms of industrial development. The problem of finding the top-k hotspots (represented as rectangular regions) in the map that maximize a scoring function is known as the k best regions search problem (k-BRS) [9], and can be formally stated as follows.

Formalization of k-BRS. Given a two-dimensional point dataset D, a monotone scoring function f, width and height parameters w, h (*query window*), and an integer k, the goal is to compute a ranked list of k axis-aligned rectangles R_i with dimensions $w \times h$, such that for each i, j, $1 \leq i < j \leq k$, it holds that:

- $f(R_i) \geq f(R_j)$, i.e., the rectangles are ranked in decreasing order of their score,
- $R_i \cap R_j = \emptyset$, i.e., R_j does not overlap with any higher ranked result R_i, and
- any other rectangle R' either has score $f(R') \leq f(R_k)$, or there exists another rectangle R_i in the top-k list such that $f(R_i) \geq f(R')$ and R' overlaps with R_i

A scalable implementation for k-BRS. We have designed and implemented a solution to the k-BRS problem that can scale out by distributing and parallelizing the computation over multiple workers (computing nodes) in a cluster. To address this problem, we have proposed and evaluated different methods for parallel and distributed top-k best region search while taking into account the criterion of overlap among the returned results. In particular, we consider a setting where the input dataset is partitioned across several workers that perform local computations in parallel, while a coordinator node is assigned the task to coordinate the execution and merge local results

to produce the global top-k result list. Notice that this is the default setting employed in all Big Data platforms, e.g., Hadoop MapReduce, Apache Spark, and Apache Flink.

Our implementation partitions data points spatially by a uniform grid with partition width $w_p >> w$ and height $h_p >> h$. Each point with coordinates (x, y) is mapped to partition $P_{i,j}$, with $i = \lceil x / w_p \rceil$ and $j = \lceil y / h_p \rceil$. In Spark, this entails a simple map followed by a groupByKey, that parses the original data and generates a new pairRDD with key being the partition *id*, and value the list of points belonging to this partition. Partitions are held by N nodes (in our case, the Spark workers). Typically, the number of nodes is smaller than the number of partitions. For example, consider Figure 6, where the input data is partitioned into 25 parts distributed among 4 nodes (N_1 to N_4).

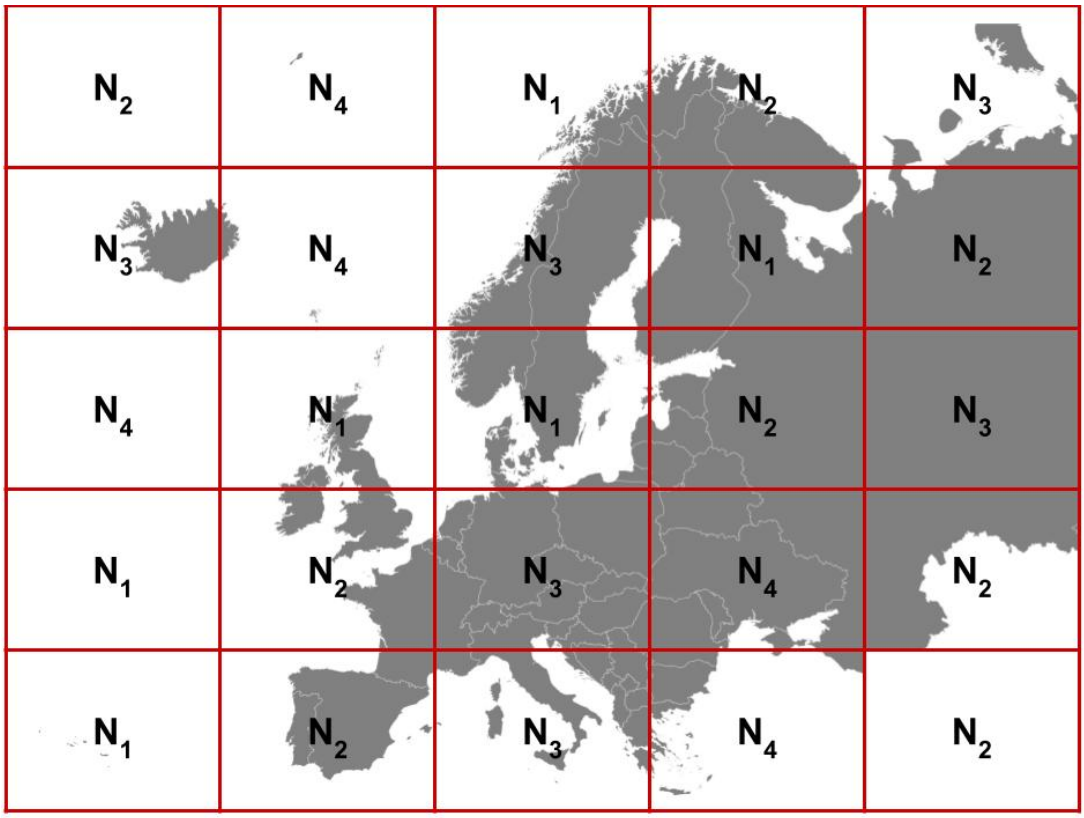


Figure 6: Partitioning of the input data to 25 partitions, which are uniformly assigned to the 4 available nodes – Spark workers.

The node holding each partition is responsible for processing the partition and identifying top-*k* regions having their top-left corner within the partition. Notice however that the local processing at the workers leads to a set of *partial results* corresponding to each individual partition. It is important to note that the final results – the global top-k best regions – are not necessarily included in these partial results, as the merging of the results of neighbouring partitions may lead to invalidation of some of the results due to overlaps. Hence, the next step is the aggregation of these partial results for generating the final top-k list. We have implemented two alternative

strategies for aggregating local results: a) the multi-round algorithm (MR) that progressively collects partial top-k results from each node in the cluster, while a coordinator handles the aggregation of the global top-k list, b) the single-round algorithm (SR), where the algorithm executed by each node is enhanced with additional conditions that anticipate potential overlapping solutions from neighbouring nodes. Moreover, we have implemented a hybrid algorithm (HY), which covers the space between the single-round and multi-round algorithms, balancing the number of rounds and the number of results expected from each round. Detailed descriptions of these algorithms can be found in Deliverable D3.1, and in [10].

2.2.2. API

The Best Region Search component is implemented over Apache Spark and includes all three algorithms described above. It is distributed as a stand-alone jar file. Here, we describe the required parameters, and the expected input and output format. The installation steps and commands for submitting the jar file for execution on a Spark cluster are detailed in the next section.

The input data for this component should be in a tabular format. For example, input data may represent a collection of companies, with each company being characterized by its coordinates and a list of keywords, e.g., "industrial", "research", "NGO", "not-for-profit". Additional attributes are also possible, e.g., revenue or number of employees, and can be used for defining the score function.

Score function. The algorithm works with any monotone function. For ease of use, functions count and sum are already supported out-of-the-box. The user needs to define the target column (parameter *targetColumn*), which corresponds to the attribute that will be used as a parameter for the function.

Input parameters. There are two sets of parameters. The first corresponds to the parameters for defining the dataset and query, whereas the second is responsible for configuring the algorithms.

- Query parameters:
  - k : number of top regions to be found
  - eps : the size of region to be found (measured in degrees)
  - targetColumn : indicates a column name with non-negative value from input data on which the scoring function is applied. If null, the number of enclosed points in each region is used as the score.
  - keywordColumn : column name for keywords. Set to null if no keywords are used.
  - keywords : list of keywords separated by ','. It is used to filter a record if its keywordColumn contains one of keywords. Set to null if no keywords are used.
- System parameters:
  - partitionCNT : number of data partitions. This parameter controls the following trade-off. On the one hand, larger partitions reduce the area covered by border cells, thereby reducing the probability that overlapping regions of two partitions require more rounds. On the other hand, they also lead to scalability problems of

the local algorithm (the workers that hold dense partitions run out of memory, swap aggressively, and eventually crash). The default value for partionCNT is 100.

- algorithmType : 0 for multi-round, 1 for single-round, 2 for hybrid.
- k' : number of collected local results per round for the hybrid algorithm. This parameter controls the following trade-off. On the one hand, smaller values of k' lead to lower execution time for each round of computation. On the other hand, they may also lead to a higher number of rounds required. The default value for k' is 5.

For example, assume an input table with the following columns: companyID; lon; lat; revenue; tags, and we set *targetColumn* = revenue, *keywordColumn* = tags, and *keywords* = Food, Restaurant. At first, we filter records *r* for which *r*.tags contains Food or Restaurant, then we find top regions where the sum of companies' revenue is maximized. Otherwise, if we set *targetColumn* = null, *keywordColumn* = null, and *keywords* = null, then we find top regions where the number of companies is maximized regardless of companies properties.

Input data format. The input data can be read from a CSV file with a header line, used for defining the column names. Columns are separated by a semicolon ';'. Furthermore, the file includes the coordinates – two columns named: "lon" and "lat". If the user asks for a specific column as scoring function or defines keywords, these columns should also exist in the schema of the input data. The keywords are separated with ','.

Output format. The output is provided in GeoJSON format[17], with each result being represented by the center of the region, its position in the list of ranked top-k regions, and its score. For example:

```
{
  {
    "center" : [1.2,3.4],
    "position" : "1",
    "score" : "10.0"
  },
  {
    "center" : [5.6,7.8],
    "position" : "2",
    "score" : "9.0"
  }
}
```

[17] https://geojson.org/

2.2.3. Installation and usage

The Best Region Search component is packaged as a jar file, including all external dependencies. The input dataset should be stored on HDFS, to which all the Spark workers have access. The requirements are as follows:

1. The stand-alone jar file named SpatialProject.jar.

2. Running Spark cluster, with a configured HDFS for storing the input data.

3. The input dataset stored in HDFS.

A query can be submitted for execution to the cluster as follows:

```
spark-submit --class SDL.main.Run --packages org.locationtech.jts:jts-core:1.16.0 SpatialProject.jar k eps partitionCNT algorithmType k' targetColumn keywordColumn keywords pathToDataOnHDFS
```

Ignore k' for MR and SR algorithms.

For example:

```
spark-submit --class SDL.main.Run --packages org.locationtech.jts:jts-core:1.16.0 SpatialProject.jar 30 0.01 100 2 10 employees keywords Restaurant, Cooperative /data.csv
```

The above command will return the top 30 regions of size 0.01*0.01 (in degrees) maximizing the number of employees, using the hybrid algorithm. Furthermore, the command instructs the algorithm to consider only the companies that contain "Restaurant" or "Cooperative" in their keywords. The data file is saved in /data.csv (in this case, the root point is assigned to point to HDFS root). A snippet of the output is shown in Figure 7.

2.2.4. Experimental evaluation

We have conducted an experimental evaluation to assess the performance of the Best Region Search component, investigating its scalability and exploring the impact of the various parameters. For our experiments, we use a real-world dataset comprising 64 million records representing Points of Interest from OpenStreetMap[18] and geo-located photos from Flickr[19] worldwide. We have mapped these to 26 million distinct locations (points). We have assigned a weight (score) to each point denoting the number of records (POIs or photos) mapped to it. The experiments are executed on a cluster of 11 nodes, each with 30 GB of RAM. Ten of the nodes are configured as workers, while the eleventh is indicated as the master/coordinator. The cluster runs Spark 2.4.3, and Hadoop HDFS.

[18] https://www.openstreetmap.org

[19] https://www.flickr.com

```
{
{
"rank":1,
"center":[8.04837,45.551500000000004],
"score":564.0
}
{
"rank":2,
"center":[11.36347,44.50967],
"score":403.0
}
{
"rank":3,
"center":[14.19131,42.47954000000001],
"score":237.0
}
{
"rank":4,
"center":[12.59074,41.85324],
"score":224.0
}
{
"rank":5,
"center":[9.17138,45.4266],
"score":217.0
}
{
"rank":6,
"center":[7.31013,45.73395],
"score":201.0
}
{
"rank":7,
"center":[16.786953849999996,41.08619932999999],
"score":188.0
}
```


Figure 7: Example of BRS output.

We start by comparing the performance of the multi-round (MR) and the single-round (SR) algorithms. We vary the number of requested top-k regions from 50 to 500. Figure 8 shows the results – wall clock time for both algorithms on the left Y axis and number of rounds for MR on the right Y axis. We see that the value of k has only a minor influence on the performance of SR. Conversely, the execution time of MR increases as k increases, eventually becoming around 40 times higher than that of SR. This stark difference is attributed to the number of rounds required by MR. Of course, one round of execution for SR is more time-consuming than for MR due to the extra time spent on creating and maintaining the dependency graph and continuing the computation of results until k safe ones are found. Yet, this difference is very small to compensate the extra overhead incurred by the high number of rounds required by MR.

Figure 9 depicts the influence of the respective parameter k' for HY. In this case, k' denotes the number of safe regions requested per partition. We see that a very low value of k' raises the need for additional rounds, since the appearance of non-admissible results prevents the coordinator from obtaining a valid top-k at the first place. However, with k' equal to 6, HY already completes in a single round and achieves the optimal performance, which is around one third of the baseline performance of SR (this would correspond to the worst performance of HY).

Figure 10 presents the execution time of SR and HY algorithms, for different values of k, and for two hybrid executions with k' = 5 and k' = 10, noted as HY(5) and HY(10), respectively. The plot also includes the number of rounds for HY(5) (right Y axis). The number of rounds for HY(10) (and for SR) is always 1, and therefore it is omitted. As expected, the value of k brings a noticeable increase

on the cost of SR, since it leads to larger dependency graphs. For HY(5), a higher k leads to more rounds, which also causes an increase in execution time. Nevertheless, HY(5) still outperforms SR, because due to the low value of k' the additional rounds are much faster compared to one round of SR. Also, HY(10) exhibits a very mild increase of the execution time, since HY(10) always takes 1 round.

Our last set of experiments focuses on investigating the scalability of the three algorithms, when varying the size of the dataset. Given the original dataset, containing 26 million distinct points, we derive datasets of size from 5 to 20 million points by applying uniform sampling. Figure 11 plots the execution time of the three algorithms. We observe that MR is the slowest algorithm in all cases, while HY performs better than SR. Moreover, MR demonstrates poor scalability compared to the other two, which exhibit similar performance, with HY scaling slightly better.

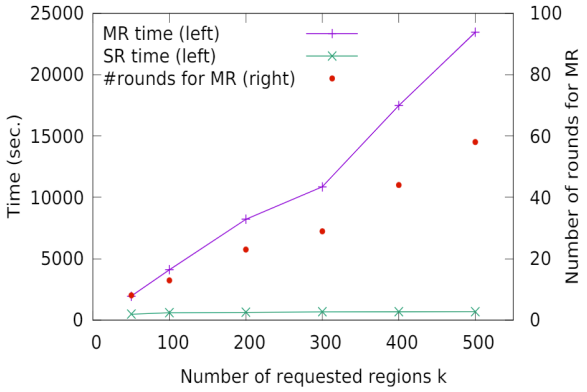


Figure 8: Performance of MR and SR for varying k

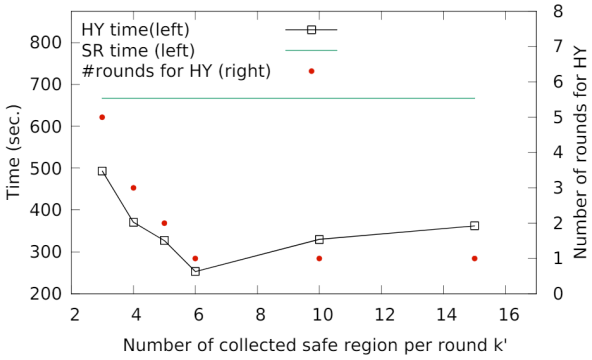


Figure 9: Execution time of HY for varying k (with SR included for reference).

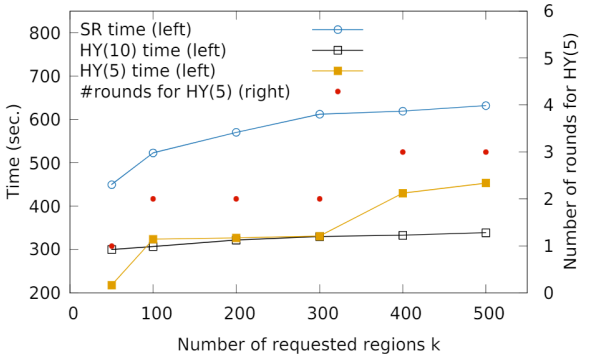


Figure 10: Comparison of SR and HY for varying k. HY(5) and HY(10) correspond to HY with k' = 5 and k' = 10, respectively. HY(10) always completes in a single round.

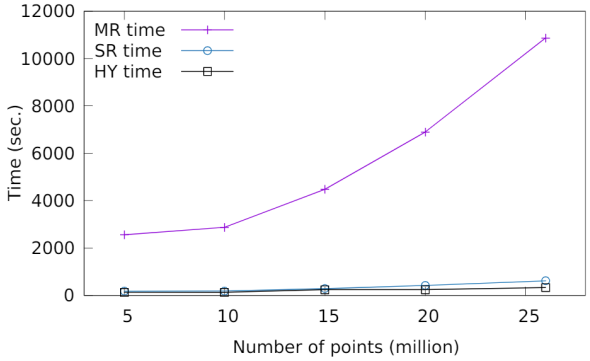


Figure 11: Execution time of MR, SR, HY for varying the number of points.

The experiments show that SR substantially outperforms MR in all cases, indicating that the extra cost incurred by multiple rounds dominates that for retrieving a sufficiently larger number of local results to ensure the construction of the global top-k in a single round. Overall, HY is the most efficient algorithm, indicating that in practice it suffices to compute just a few more than k local

results to ensure that the coordinator can assemble the correct top-k list, despite any inadmissible local results.

More detailed discussions of our experimental insights and further experiments (e.g., varying the size of query window and the number of executing nodes) are presented in [10].

3. Similarity Search and Join

In this section, we describe the components we have implemented for similarity search and join. Specifically, SimJoin represents entities as sets of elements, and discovers all pairs of similar sets based on set overlap, supporting also fuzzy matching between elements. SimSearch retrieves the top-k most similar entities to a given query entity. It supports different types of attributes, including textual, numeric and spatial. Finally, SimSearchTS is designed for similarity search and join over time series data.

3.1. Set similarity joins

3.1.1. Implemented functionalities

The SimJoin component allows to discover similar entities within the data lake based on overlap similarity in set-valued attributes. To achieve efficiency and scalability, it relies on filtering algorithms, as described in Deliverable D3.1. In a nutshell, this exploits an appropriate index and filtering conditions to quickly identify a relatively small set of candidates, thus restricting the number of required comparisons only to those candidates.

SimJoin supports two different modes of operation:

- *Standard similarity join*. Here, each set -from now on called *element* in this case- contains one or more tokens. These tokens can be keywords, tags, q-grams extracted from a string, etc. For instance, an element may represent a product of a company, characterized by a set of keywords. In this mode, matching between tokens results in a boolean value: two tokens have similarity score 1 if they are identical, or 0 otherwise. Then, the similarity between two elements is computed using some typical set similarity measure (in our implementation, we currently support Jaccard similarity).
- *Fuzzy similarity join*. Here, each set -from now on called *record* in this case- contains one or more *elements*. An example of a record can be a company, which is described by its products, with each product being in turn characterized by a set of keywords. In this case, the similarity between two elements of a record can take any value between 0 and 1 (hence, the term *fuzzy*). Then, the similarity between two records is defined as the maximum matching score in the respective bipartite graph representing their elements.

In each mode, SimJoin supports the following operations: (a) ThresholdJoin, which retrieves all pairs of entities with similarity score above a threshold parameter θ; (b) TopKJoin, which retrieves the k most similar pairs of entities; and (c) kNNJoin, which retrieves, for each entity, its k-nearest neighbors.

In addition, SimJoin supports assigning different weights to each set., as described in Deliverable D3.1 (see Section 3.2.4).

The implemented algorithms in SimJoin are based on techniques described in [11][12][13]. In particular, for fuzzy set similarity joins, we follow the Silkmoth algorithm presented in [13] but with some adaptations to reduce its memory footprint. More importantly, Silkmoth is designed only for threshold-based join. Therefore, we have designed and implemented variations of this algorithm to also support the top-k and kNN operations required in our case.

3.1.2. API

In the following, we present the Java API of the SimJoin component[20].

The input can be read from a CSV file. Two functions are provided for this purpose, depending on whether the mode is standard or fuzzy.

```
GroupCollection<String> fromCSV(String file, int colSetId, int colSetTokens, int colWeights, String colDelimiter, String tokDelimiter, int maxLines, boolean header, String tokenizer, int qgram)
```

```
GroupCollection<ArrayList<String>> fromCSV(String file, int colSetId, int colElemId, int colSetTokens, int colWeights, String columnDelimiter, String tokenDelimiter, int maxLines, boolean header, String tokenizer, int qgram)
```

After reading the data into the collection, the user can invoke the corresponding function depending on the chosen operation, as listed below.

```
void thresholdJoin(GroupCollection<T> collection, double threshold, ConcurrentLinkedQueue<MatchingPair> results);
```

```
void thresholdJoin(GroupCollection<T> collection1, GroupCollection<T> collection2, double threshold, ConcurrentLinkedQueue<MatchingPair> results);
```

[20] https://github.com/smartdatalake/simjoin

```
void knnJoin(GroupCollection<T> collection, int k, double limitThreshold,
ConcurrentLinkedQueue<MatchingPair> results);
```

```
void knnJoin(GroupCollection<T> collection1, GroupCollection<T> collection2,
int k, double limitThreshold, ConcurrentLinkedQueue<MatchingPair> results);
```

```
void topkJoin(GroupCollection<T> collection, int k,
ConcurrentLinkedQueue<MatchingPair> results);
```

```
void topkJoin(GroupCollection<T> collection1, GroupCollection<T> collection2,
int k, ConcurrentLinkedQueue<MatchingPair> results);
```

Notice that the API is the same for both standard and fuzzy similarity joins; the implementation differs according to the type of input and mode of operation.

The similar pairs of entities are returned within the ConcurrentLinkedQueue<MatchingPair> results and can be consumed for further analysis.

The format of a typical input file for standard similarity join is as shown below, where semicolon (;) is used as column delimiter. The first column contains the Set ID, whereas the second column contains a list of comma-separated tokens.

```
id;title
1;Preliminary Design of a Network Protocol Learning Tool Based on the Comprehension of High School Students: Design by an Empirical Study Using a Simple Mind Map
2;A methodology for the physically accurate visualisation of roman polychrome statuary
3;Comparison of GARCH, Neural Network and Support Vector Machine in Financial Time Series Prediction
4;Development of Remote Monitoring and Control Device for 50KW PV System Based on the Wireless Network
```

In the case of fuzzy similarity join, the input file format is shown below. Again, the first column contains the set id. The third column contains the Element ID and the last column contains the list of tokens.

```
author_id;author_name;paper_id;paper_title
0;Makoto Satoh;713dc931-1846-486f-b9ec-287c2e81864d;fir filter design on flexible enginegeneric alu array and its dedicated synthesis algorithm
0;Makoto Satoh;dd7c908a-2d97-4220-ac03-fce737bcf0e7;algorithmic thinking learning support system with eassessment function
0;Makoto Satoh;1d31eadf-7bd3-449c-8ce0-19f5292d08f7;analysis of twolevel data mapping in an hpf compiler for distributedmemory machines
0;Makoto Satoh;ac10ade0-e463-4d20-b401-6c8825fde92b;a high speed database machine hdm and its performance evaluation
```

The output is a CSV file with three columns as shown below. Each record denotes a matching pair of sets. The first two columns contain the IDs of the sets. The third column contains the similarity score.

```
Craig McAnulla,Sarah Hunter,0.9
Pilar Lafont Morgado,Andrés Díaz Lantada,0.8333333333333334
Yoon-Hee Hwang,Un-Sook Choi,0.9090909090909091
Philippe Poure,Shahrokh Saadate,1.0
Tor P. Schultz,Harry P. Schultz,0.9090909090909091
Diego Cirio,Andrea Pitto,0.8181818181818182
```

The configuration file is in json format. The parameters are shown below, with the required fields indicated in bold:

- query_file: File for the left collection of the join. If omitted, then self-join is implied.
- **input_file**: File for the right collection of the join or only collection for self-join.
- max_lines: Lines to read from both files. If -1, read all lines.
- **set_column**: Column containing the Set ID. Numbering starts from 1.
- elements_column: Column containing the Element ID. Numbering starts from 1. Required only in fuzzy mode.
- **tokens_column**: Column containing the tokens. Numbering starts from 1.
- **column_delimiter**: Delimiter that separates columns.
- token_delimiter: Delimiter that separates tokens in tokens_column. If omitted, default is white space.
- header: Boolean indicating the existence of a header. Default is false.
- output_file: Name of the file to output results. If omitted, then the results are not stored, but only the size of the join (i.e., number of pairs) is provided.
- log_file: Name of the file to output logs.
- **mode**: Values allowed: standard or fuzzy.
- **join_type**: Values allowed: threshold, knn or topk.
- **threshold**: Similarity threshold (double).
- **k**: Number of results to return (integer).
- tokenizer: How to tokenize tokens. Values allowed: word or qgram. Default is word.
- qgram: The qgrams length.

3.1.3. Installation and usage

SimJoin is implemented in Java and the source code is publicly available under the Apache License 2.0[21]. It is packaged as a single Maven project and has been built and tested using Java JDK v.1.8.

External dependencies include the following libraries:

- JGraphT[22], a Java library for graph theory data structures and algorithms, to create the graph and calculate the maximum weight bipartite matching for the fuzzy mode.
- Trove[23], a Java library that provides high speed object and primitive collections for efficiency.

To install and use the SimJoin component, the following steps should be performed:

[21] https://github.com/smartdatalake/simjoin

[22] https://jgrapht.org/

[23] http://trove4j.sourceforge.net/html/overview.html

Step 1. Download or clone the project:

```
$ git clone https://github.com/smartdatalake/simjoin.git
```

Step 2. Open terminal inside root folder and install by running:

```
$ mvn install
```

Step 3. Edit the parameters in the config.json file.

Step 4. Execute by running:

```
$ java -jar target/simjoin-0.0.1-SNAPSHOT-jar-with-dependencies.jar [Config File]
```

3.1.4. Experimental evaluation

In the following, we present an experimental evaluation of the SimJoin component. The experiments were conducted on a server with Intel Xeon E5-2420 v2 CPU with 2.20GHz processor and 64GB RAM running Ubuntu.

For standard similarity join, we tested our component on two datasets: (a) Atoka – a collection of companies provided by our partner SpazioDati, where each company is represent by a set of categories, and (b) DBLP – a collection of scientific publications from DBLP, where each publication is represented by a set of terms extracted from its title. The characteristics of these datasets are listed in Table 2.

Table 2: Datasets used for standard similarity join.

Dataset	Number of records	Average tokens per record
Atoka	4.5 M	5.3
DBLP	3 M	9.6

We use two scenarios for testing. For the Atoka dataset, we randomly select 10,000 companies as query set, and we perform a join operation between this query set and the entire collection to identify similar companies to the selected ones. For the DBLP dataset, we perform a self-join operation to identify all pairs of publications with similar titles. In both cases, we also consider subsets of the entire collection to test scalability.

In these tests, we use as competitor Magellan[24], in particular its Python library py_stringsimjoin, which is an open source project developed by the University of Wisconsin. Magellan is a state-of-the-art library using filtering techniques for set similarity joins [14]. During its execution, we use the Cython version, which is much faster, and we also enable its support for multiple cores.

We measure the execution time for different collection sizes (N) and for different similarity thresholds (θ). The results are shown in Figure 12 and Figure 13. As we can see, the execution time

[24] https://sites.google.com/site/anhaidgroup/projects/magellan/py_stringsimjoin

increases, both for SimJoin and Magellan, as the dataset size increases, as well as when the similarity threshold decreases. This is due to the increase in the number of matching pairs, as also shown in the plots. SimJoin has a lower execution time than Magellan in all cases. The difference becomes significantly higher in the case of the DBLP dataset, which is attributed to two reasons. First, as we can see, the number of pairs is significantly smaller in this dataset, i.e., the selectivity of the join operator in this case is much higher. This suggests that in SimJoin the execution of filters to prune candidate pairs is more efficient. Moreover, Magellan does not directly support a self-join operation; instead, a self-join is treated as a join between two collections that happen to be the same. Instead, SimJoin explicitly supports self-join operations, which allows for certain additional optimizations (essentially, building the inverted index on tokens incrementally).

A drawback of threshold-based join is that it may be difficult for the user to set an appropriate value for the threshold parameter. To overcome this, in SimJoin we have implemented algorithms for kNN and top-k join operations. These enable the user to directly choose the desired number of results, and also to retrieve the top results progressively, which is very useful in data exploration scenarios. In addition, our implementation of the kNN and top-k functionalities in SimJoin allows to also set a similarity threshold θ that is used in conjunction with the parameter k, in which case the k most similar results are retrieved as long as their similarity score is higher than θ. In this way, we avoid cases where the algorithm takes too long to execute because it searches over candidates with very low similarity score. In the following, we examine the performance of SimJoin in these operations, varying the value of k, and using $\theta = 0.7$ as a cut-off threshold for similarity. We only consider SimJoin in these experiments, since Magellan does not provide functionalities for kNN or top-k joins. The results are shown in Figure 14 and Figure 15, for kNN join, and in Figure 16 and Figure 17, for top-k join. Again, we can observe that overall the execution time increases accordingly to the number of pairs.

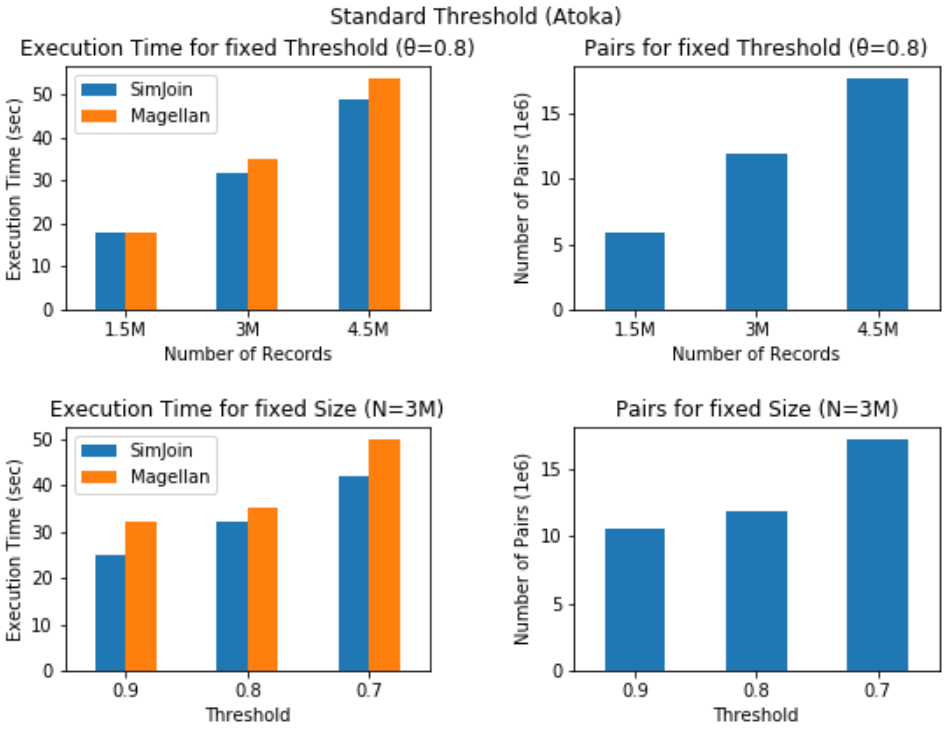


Figure 12: Results for threshold join on Atoka.

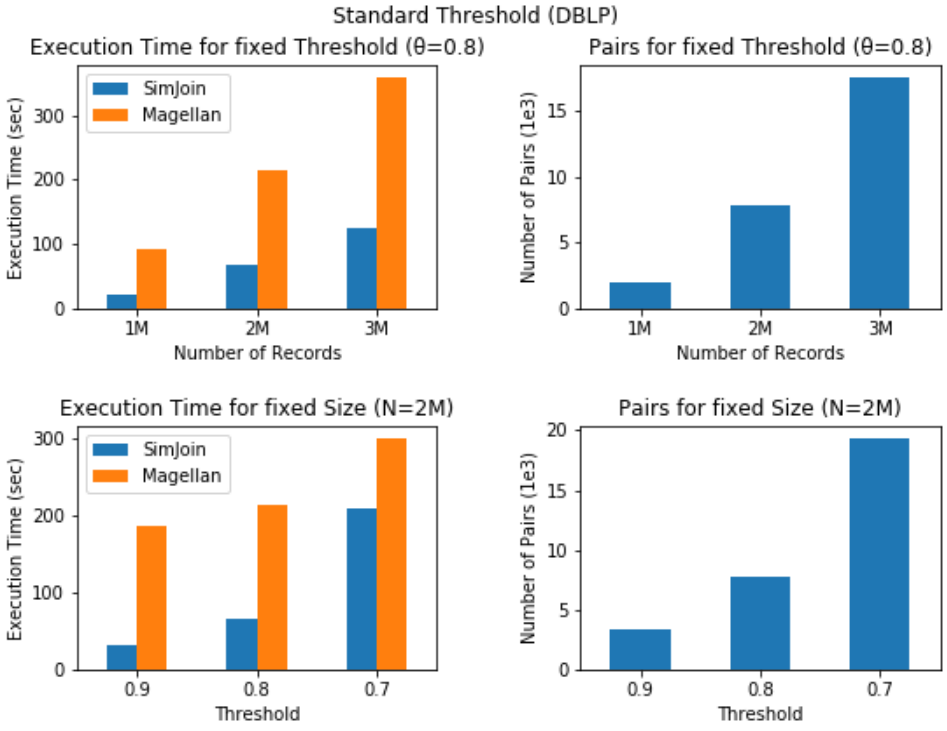


Figure 13: Results for threshold join on DBLP.

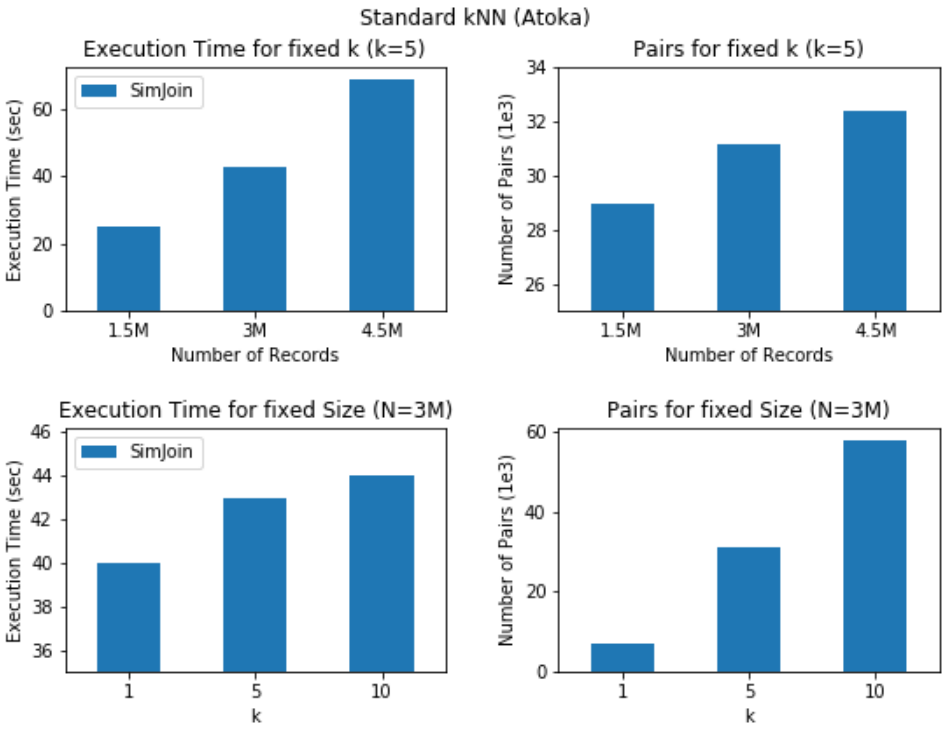


Figure 14: Results for kNN join on Atoka.

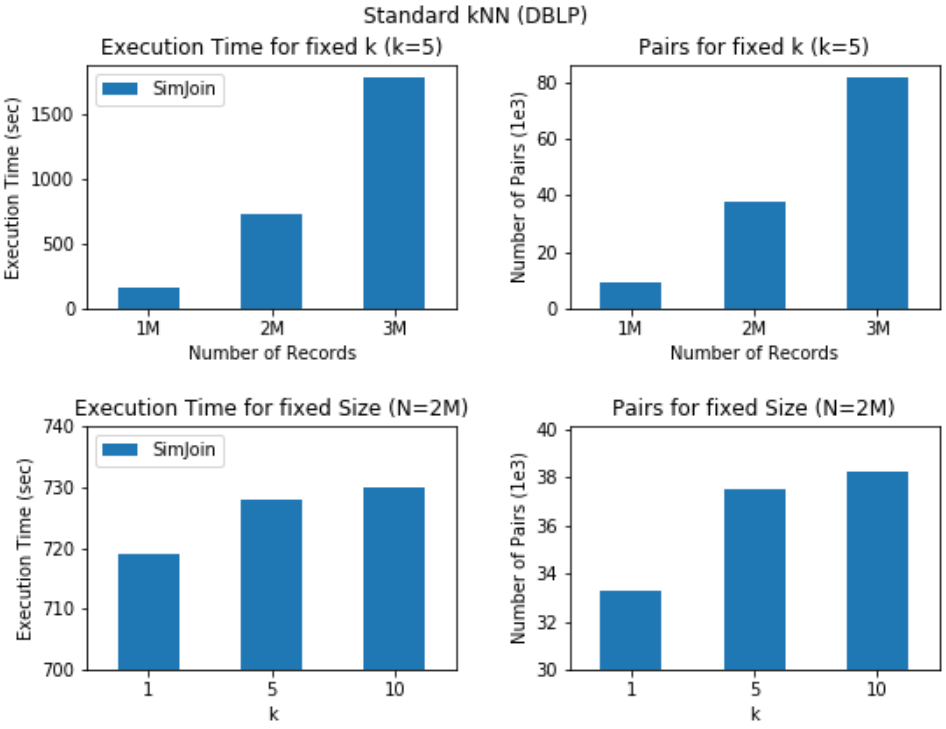


Figure 15: Results for kNN join on DBLP.

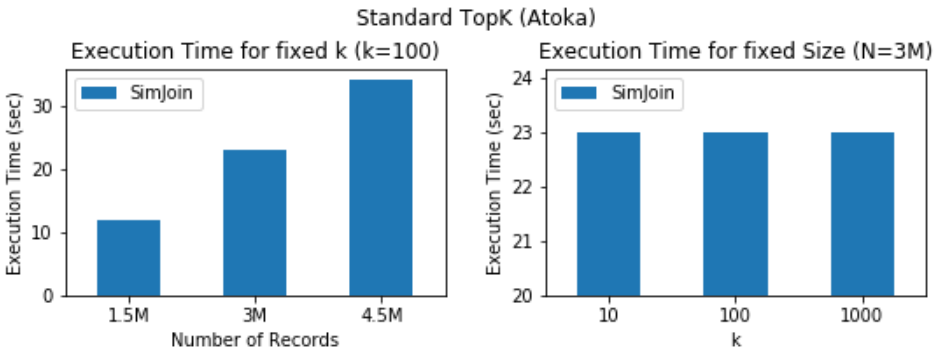


Figure 16: Results for top-k join on Atoka.

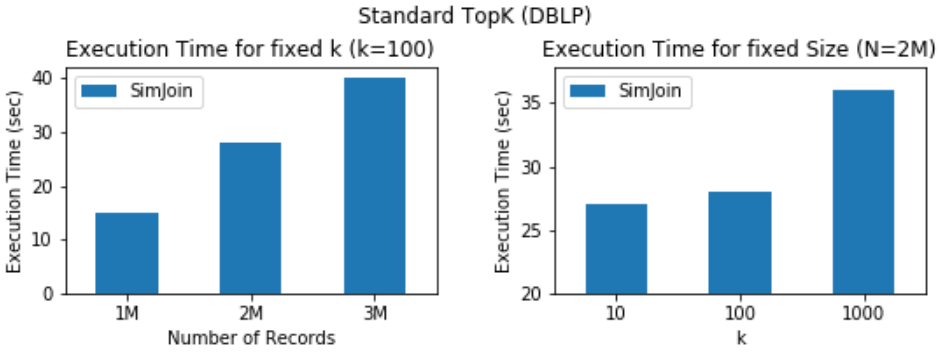


Figure 17: Results for top-k join on DBLP.

Next, we present an evaluation for fuzzy set similarity join. To test this mode, we use the following datasets: (a) GDELT – this is a collection of news articles obtained from the GDELT project[25]; we group articles per day and location, and we search for similar groups of articles, where each article is characterized by the set of persons and organizations mentioned in it; and (b) DBLP – in this case, we search for pairs of similar authors, where each author is represented by the set of his/her papers, and each paper is in turn represented by a set of tokens extracted from its title. The characteristics of the obtained datasets are listed in Table 3.

[25] https://www.gdeltproject.org/data.html

Table 3: Datasets used for standard fuzzy join.

Dataset	Number of records	Average elements per record	Average tokens per element
GDELT	520 K	6.2	9.6
DBLP	250 K	22.8	9.5

For GDELT, we perform a foreign join operation, randomly selecting 10,000 records as a query set. For DBLP, we perform a self-join operation. As in the previous experiments, we vary the collection size and the value of the similarity threshold or the number of top results, accordingly. For threshold-based fuzzy set similarity join, our implementation is based on the state-of-the-art Silkmoth algorithm [13], with a few modifications to improve memory utilization and efficiency. Thus, for these experiments, we include in the plots the execution time of our component both before and after these modifications (with the former being denoted as 'Silkmoth'). Notice that kNN and top-k join operations are not supported by the Silkmoth algorithm, hence for those experiments we only show our own method.

For threshold-based join, the results are shown in Figure 18 and Figure 19. In all cases, as expected, execution time increases as the collection size increases or the similarity threshold decreases, following the increase in the number of similar pairs that are detected. In DBLP the join selectivity is higher than in GDELT, in which case the effect of our modifications becomes apparent, resulting in significantly lower execution times in this dataset.

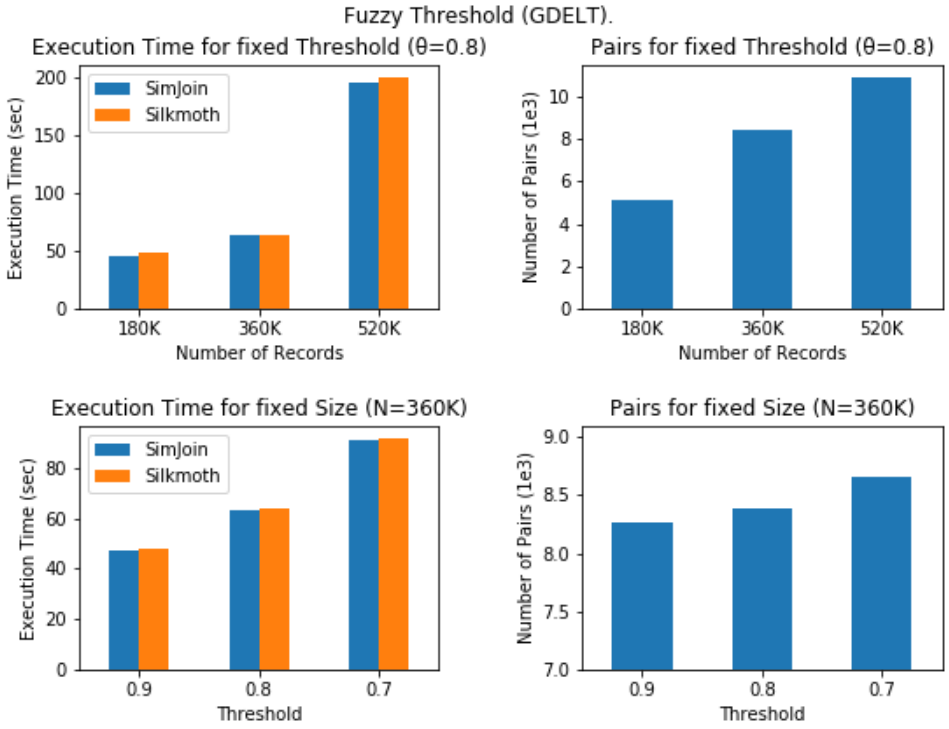


Figure 18: Results for threshold-based fuzzy join on GDELT.

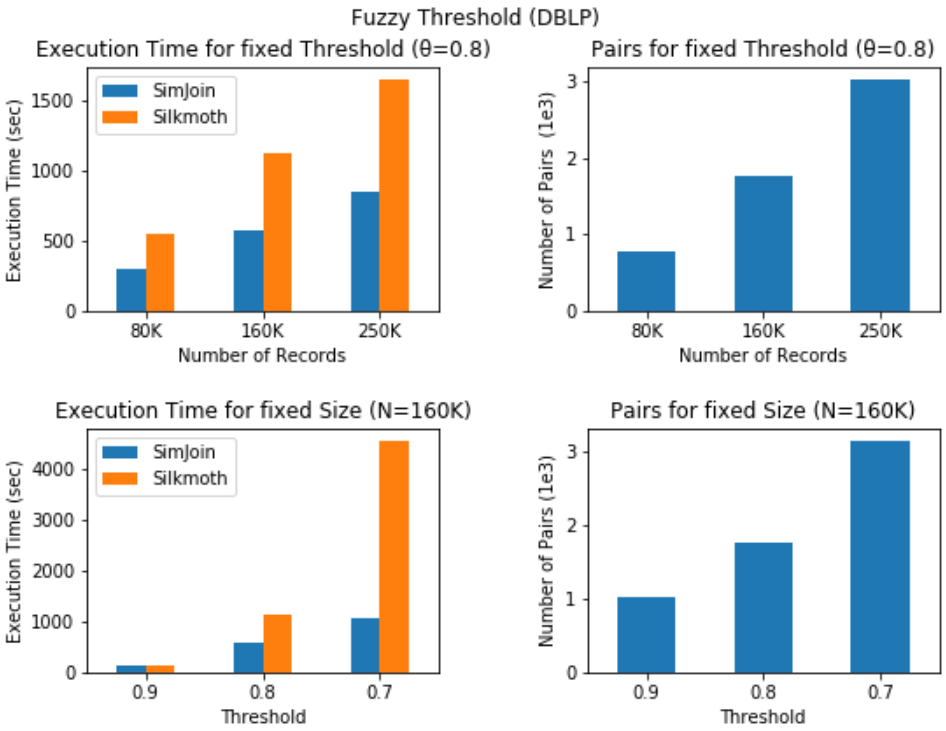


Figure 19: Results for threshold-based fuzzy join on DBLP.

Finally, the results for the kNN join and top-k join operations, for each of the considered datasets, are shown in Figure 20, Figure 21, Figure 22 and Figure 23, respectively. Again, increasing the collection size results in increasing execution time. However, in most cases, increasing the value of k does not seem to have a significant impact on execution time, which suggests that SimJoin can be efficient when retrieving results progressively.

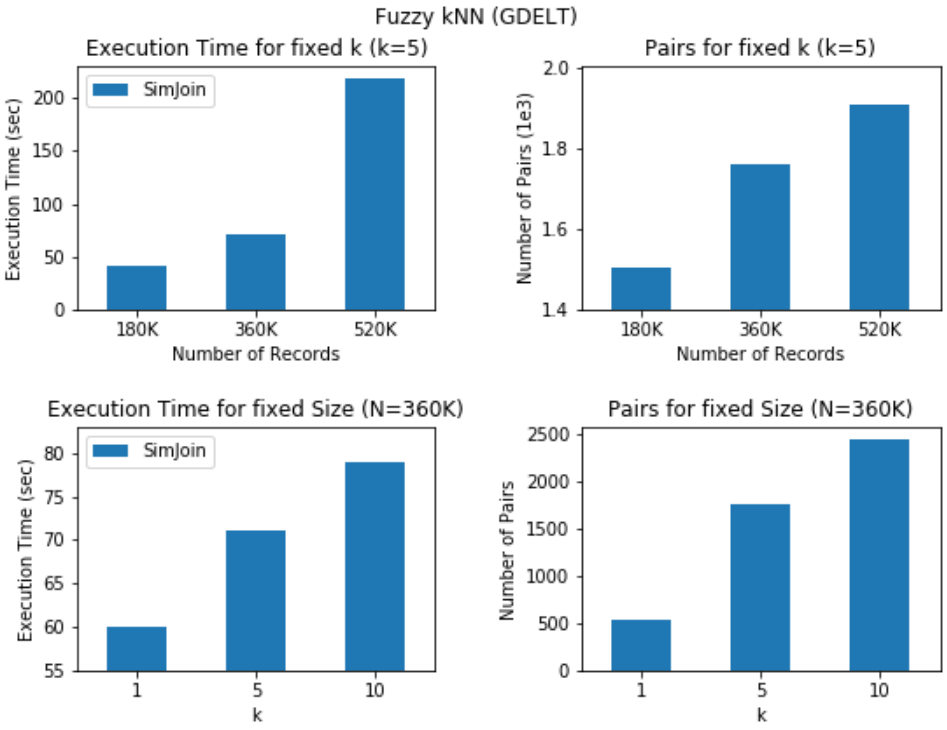


Figure 20: Results for kNN fuzzy join on GDELT.

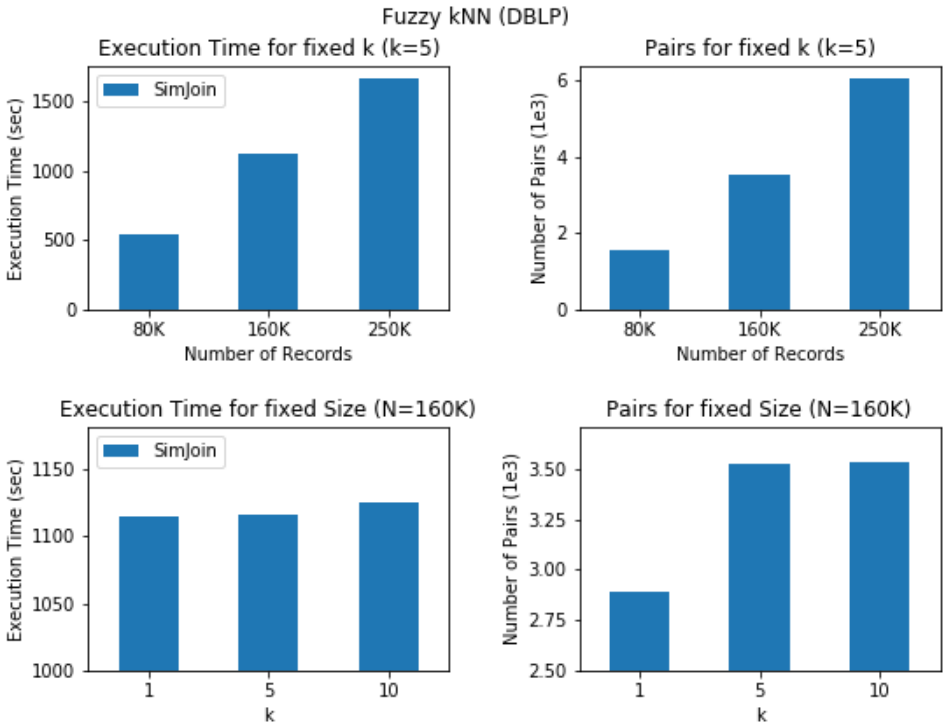


Figure 21: Results for kNN fuzzy join on DBLP.

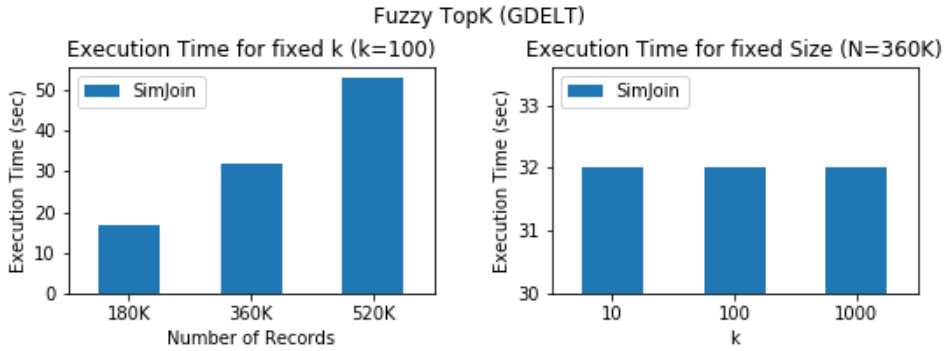


Figure 22: Results for top-k fuzzy join on GDELT.

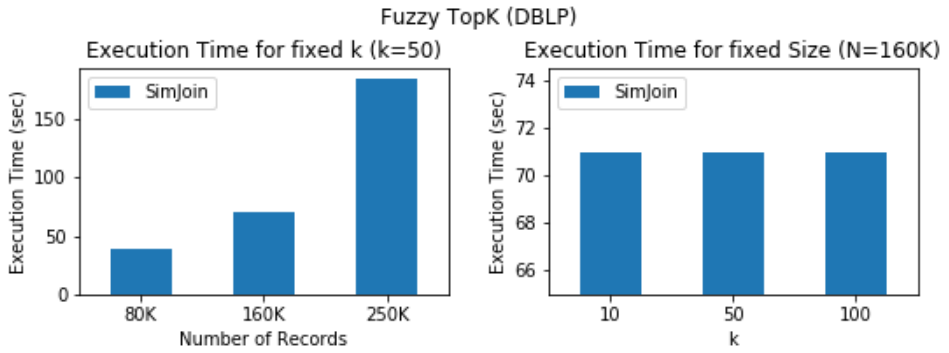


Figure 23: Results for top-k join on DBLP.

3.2. Similarity search over heterogeneous attributes

3.2.1. Implemented functionalities

Overview. The SimSearch component provides functionality for finding similar entities. Given a query entity described by one or more *properties*, it retrieves a ranked list of entities having the most similar values in these properties. This component can be used to explore the entities in the data lake, starting from some initial entities and gradually retrieving others that are similar to these. For instance, given a company characterized by its revenue, number of employees, description, date of establishment and location, the SimSearch component can retrieve other companies that are similar to it with respect to these criteria.

There are two main aspects characterizing the design and implementation of SimSearch: (a) support for properties that are of *different types*, and (b) support for properties that come from *different sources*. We describe how these are addressed next.

SimSearch supports the following types of properties:

- *numerical*, e.g., integers, floats, dates;
- *spatial*, e.g., geocoordinates;
- *categorical*, e.g., keywords, tags.

Details about distance calculations for these types of properties and, especially, how to scale the results so that ranking scores across different properties can be compared and aggregated are presented in Deliverable D3.1: "Similarity search, entity resolution and ranking" (see Section 3.3).

For each property being used for search, SimSearch offers two modes of operation:

- *ingest mode*. In this mode, the values for the specific property are ingested into SimSearch, which builds a corresponding index (B+ tree for numerical, R-tree for spatial, inverted index for categorical). Such indices reside in memory to allow executing kNN queries with low latency. Thus, this mode is preferable for interactive exploration and visual analytics.
- *in-situ mode*. In this mode, SimSearch relies on another query endpoint to retrieve top-k results for the specific property. Thus, if the data resides already in a DBMS that can process kNN queries over the desired attribute, then SimSearch can retrieve a ranked list of results by submitting a query to that DBMS instead of requiring to ingest and index the data internally. This avoids data replication and offers the ability to scale according to the scalability of the accessed DBMSs. Moreover, it allows to leverage data virtualization, which has several advantages, as it hides the complexities of dealing with input data in different shapes and formats. On the downside, this mode may increase query latency.

Modules. SimSearch comprises two main modules: (a) the *Query Backend*, which retrieves individual ranked lists for each property in the query on the fly, by either using the internal indices or issuing queries to the specified external query endpoints, and (b) the *Rank Aggregator*, which aggregates those individual ranked lists and returns the global top-k list of similar entities, ranked by aggregate scores across all considered properties.

Figure 24 illustrates an example scenario that involves instances of SimSearch against datasets available in both modes of operation. Regarding the ingest mode, the *Query Backend* module in SimSearch currently supports importing data from CSV files, but it is easy to extend it with importers for other data sources. Regarding external query endpoints, SimSearch currently supports JDBC APIs and is being extended to also support REST APIs. Using JDBC, the Query Backend can connect to any RDBMS (e.g., PostgreSQL). In that case, SimSearch can take advantage of the index support inherent in the DBMS. For example, PostgreSQL offers several types of indices[26], like B-trees for facilitating search over numerical values, generalized inverted indices (GIN)[27] for textual search, etc. Moreover, its spatial extension PostGIS[28] takes advantage of GIST

[26] https://www.postgresql.org/docs/current/indexes-types.html

[27] https://www.postgresql.org/docs/current/textsearch-indexes.html

[28] https://postgis.net/

indices and enables a wide range of spatial operations. In SmartDataLake, JDBC will also be used for connecting to Proteus, the data virtualization engine in SDL-Virt (see Deliverable D2.1: "Query engine over virtualized data"), thus being able to cover a large variety of underlying data sources and formats. Connectivity through REST APIs will be used to query data from Elasticsearch[29], a popular search engine, especially suited for full-text search, which is also used by project partners in certain use cases. Moreover, since SimSearch itself offers a REST API for submitting queries, REST API connectivity allows a SimSearch instance to use another SimSearch instance as an external query endpoint. This is an important feature, as it provides an additional means for SimSearch to scale horizontally. Recall that in the in-situ mode, SimSearch relies on the underlying DBMS(s) to satisfy scalability requirements, whereas in the ingest mode its scalability is limited, since the indices for the ingested properties need to be kept in memory. However, using this feature, it is possible to use different SimSearch instances in different machines to ingest and index different properties, and then query them from a "master" instance to answer requests combining all properties.

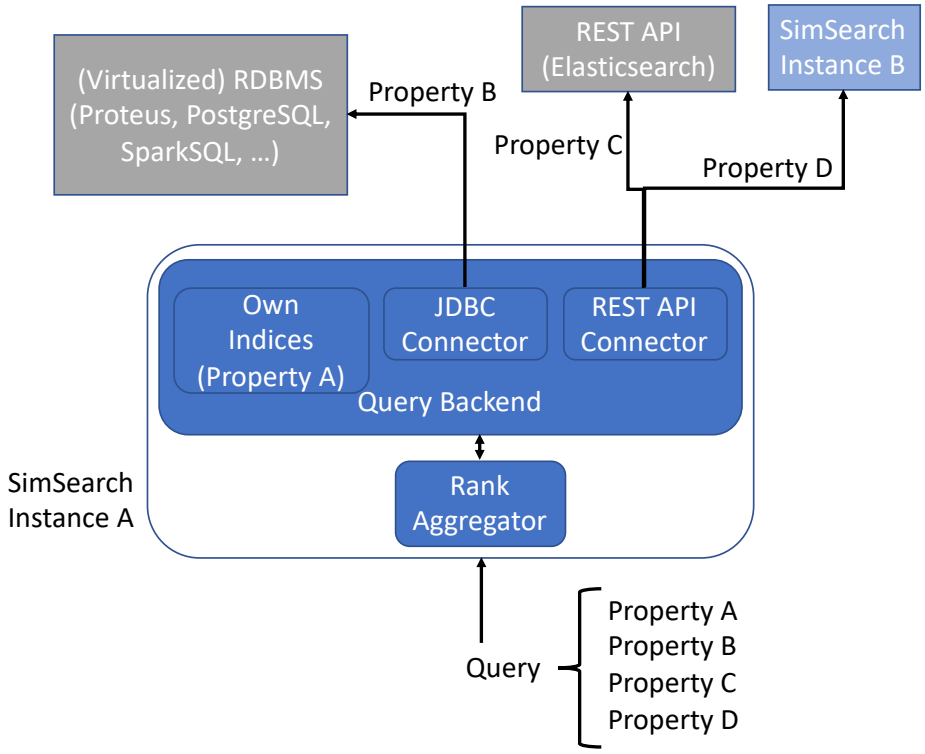


Figure 24: Similarity search example against several ingested and in-situ datasets.

The *Rank Aggregator* module accepts ranked lists of entities as issued by the Query Backend instances for each queried property, as depicted in Figure 24. Rank Aggregator is agnostic of whether the underlying data sources are ingested or queried in-situ, as long as identifiers of entities in each ranked list can be matched. As discussed in Deliverable D3.1, we assume that entities in each list are sorted by their respective scores as estimated by a scoring function. Each list should provide a sufficient number of candidates (typically, much larger than k) to ensure that

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

the top-k results with the highest aggregate scores are finally returned. The Rank Aggregator needs to examine enough candidates from each list to ensure correct top-k final results. Thus, to populate these per-property ranked lists, the Query Backend needs to execute a top-M search for each property, where $M \gg k$ is a configurable parameter that essentially controls a trade-off between ranking efficiency and quality of results.

We have implemented a uniform approach that allows ranking of entities by their similarity to a given query independently and in parallel for each property. Note that top-k processing against multiple lists may employ different access paradigms [15]. The main difference is whether random access is allowed or not. When only *sorted access* is allowed, entities in each list can only be accessed sequentially according to their score per property. If *random access* is allowed, then aggregate scores can be computed by fetching the missing property values from available lookup tables based on entity identifiers. In either case, it is assumed that all entities are present in all lists. Hence, the exact aggregate score of an entity can always be computed, either when the entity is first encountered in one of the lists (under the random access assumption) or when the entity has been eventually found in all lists (under the sorted access assumption). However, in SimSearch the situation differs, since each ranked list contains up to M entities, so an entity may appear only in some (or even none) of these lists. In that case, performing a random access for an entity that does not appear in a list, requires querying the underlying data for the missing value of that entity in that particular property. This operation may be costly or even prohibited.

The *Threshold Algorithm* (*TA*), already discussed in Deliverable D3.1 is a state-of-the-art ranking method [16] that employs unrestricted random access to the full contents of the underlying property values. In contrast, the *No Random Access* (*NRA*) algorithm [16] iterates over the sorted items in each list, without calculating exact scores through random access for missing values. Instead, it maintains a *lower* (LB) and an *upper bound* (UB) for the aggregate score of each seen entity. More specifically, LB is based on scoring the known property values per entity; its corresponding UB increments the computed LB with the maximum scores obtained for the unknown property values from other entities in the current iteration. NRA keeps inspecting sequentially all lists in parallel and also refines the bounds of all seen entities according to scores of the entities obtained in each iteration. The next final result is added to the top-k list once its LB is not below the UB of every other entity seen so far. However, note that top-k results can be reported with no exact aggregate scores, as their LB and UB scores may differ. Since this method relies on strict sorted access to the entities in each list, it typically incurs significant overhead in maintaining their bounds, especially for ranked lists of considerable size (i.e., larger M values). In our case, where lists may not contain the same entities as is typically assumed, NRA may also involve too many iterations over the lists, and sometimes can terminate after exhausting all their items. Thus, this ranking approach may incur considerable cost, especially for larger k values.

To overcome the aforementioned issues, we designed and implemented in SimSearch a hybrid variant of *TA*, called *Partial Random Access* (*PRA*), which operates under the assumption that random accesses are allowed only to the contents of the retrieved lists and not against the entire collection of entities. Thus, it performs random access against much smaller lookup tables built locally over property values only available from the ranked lists, no longer assuming that datasets are available in their entirety for lookup as in TA. Since lookups for PRA concern only a part of the

property values, they have a reduced memory footprint. However, unlike TA, exact aggregate scores may not be always available (unless an entity appears in all lists), so we must resort to bounding the aggregate scores per entity as in NRA. To do so, PRA first creates one local lookup table per ranked list based on the returned property values. Then, it iterates in parallel over the ranked lists, and for each encountered entity in the current iteration it calculates bounds on its aggregate score. Lower bound LB for this entity is calculated as in NRA, but now only using random access for a missing property value against its respective local lookup. Upper bound UB is also estimated by incrementing LB with the lowest scores available for the missing property values; these scores are those of the last items available in the respective lists. In contrast to NRA that progressively restricts the interval between the two bounds LB and UB per entity, these bounds in PRA remain fixed once computed. As a means of sorting these intervals in order to identify qualifying entities, we calculate the *average* MB of both bounds per entity and retain it in a priority queue. Exactly as in TA, we also maintain a *threshold* after scoring property values obtained from the lists in the current iteration. A new final result is issued in the top-k list once its LB is not below the current threshold and also exceeds the average bound MB of any other entity seen so far. By convention, we assign the estimated MB as the approximate aggregate score of each reported result. The method terminates once top-k results are reported or the lists are exhausted. In the latter case, k' < k qualifying entities may be found by the threshold condition, so these are complemented with the top (k-k') available entities in the priority queue based on MB values.

3.2.2. API

SimSearch[30] is implemented in Java. It provides an API including the methods described below.

mount(*sourceJSON*): This method mounts specific attributes from data sources (either in-situ or ingested) and makes them available for querying with SimSearch. If necessary (for ingested data), it also builds suitable indices on their attribute values. It takes a JSON specification (or the path to a JSON file) containing the details for connecting to the data sources and accessing specific attributes. This operation must be applied before any queries are submitted. Typically, this is carried out by the system administrator once, unless there are changes in the underlying data sources. The JSON configuration has the following format:

```
{
 "sources":[{   "name":"localPostGISdatabase",
                "type": "jdbc",
                "driver": "org.postgresql.Driver",
                "url": "jdbc:postgresql://localhost:5432/myDatabase",
                "username":"postgresName",
                "password":"postgresPassword",
                "encoding":"UTF-8" },
            {   "name":"localPath1",
```

[30] https://github.com/smartdatalake/simsearch

```
                "type":"csv",
                "directory":"./test/"  }
          ],
 "search":[ {  "operation":"categorical_topk",
                "data_source":"localPath1",
                "dataset":"dataset1.csv",
                "search_column":"persons"  },
            {  "operation":"numerical_topk",
                "data_source":"localPostGISdatabase",
                "dataset":"companies",
                "search_column":"timestamp" },
            {  "operation":"spatial_knn",
                "data_source":"localPostGISdatabase",
                "dataset":"companies",
                "search_column":"location"  }
         ]
}
```

As shown above, this configuration consists of two basic blocks:

- *sources*: This block specifies the sources containing the data that will be used in SimSearch. For in-situ data, JDBC connection details must be specified as in this example for a PostgreSQL database. For ingested data, the path and the type (i.e., format) of the data must be specified. Each source must be given a unique name, which is then used for reference in the next block.
- *search*: This block defines the queryable properties in the aforementioned sources, as well as the type of operation (*spatial_knn*, *numerical_topk*, *categorical_topk*) enabled in each one. The "*search_column*" indicate the unique name of the queryable property available in the data source; similarity search queries must mention this name.

delete(*removeJSON*): This method can be used when a specific property (or properties) must be disabled in queries. A JSON specification indicates which properties to exclude from querying. These properties may be enabled again and additional ones can be specified with the *mount()* method without shutting down a running instance of the software. As shown in the example JSON below, only the property ("*search_column*") and its supported operation need to be specified:

```
{
   "remove":[ {
                "operation":"spatial_knn",
                "search_column":"location" },
            {  "operation":"categorical_topk",
```

```
                "search_column":"persons" }
        ]
}
```

listDataSources(): This method returns a JSON array listing the currently queryable properties and the type of operation (*categorical*, *numerical*, or *spatial*) supported for each one in order to enable users to specify or verify their similarity search requests. This method does not take any arguments.

search(*queryJSON*): This method allows specification of a top-k similarity search query. The user must provide a JSON specification (or the path to a JSON file) containing the query properties and values, as well as the weight parameters. Such a JSON (e.g., *query.json*) is like this:

```
{
   "k":"20",
   "algorithm":"partial_random_access",
   "queries":[
            {  "search_column":"persons",
               "search_value":["Donald Trump","Joe Biden"],
               "weights":["0.4","0.6"] },
            {  "search_column":"timestamp",
               "search_value":"20190124142000",
               "weights":["0.3","0.2"] },
            {  "search_column":"location",
               "search_value":"POINT (10.404658 43.708845)",
               "weights":["0.5","0.7"] }
        ]
}
```

Users must specify how many (*k*) results they want to fetch in the top-k list with SimSearch. Parameter "algorithm" specifies the ranking method to be applied, which can be one of the following: "*threshold*" (default), "*partial_random_access*", or "*no_random_access*". In the *queries* block, the user in the "*search_column*" should specify at least one queryable property (having a unique name that matches those specified earlier from the data sources), as well as a suitable query value. This value must be of a data type consistent to the one in the underlying data (e.g., a numerical value for a real-valued property, a WKT string for a geometry, etc.). Finally, the user must specify *weights* for each queried property. One or multiple combinations of weights can be configured (e.g., two combinations in this example); these calibrate the importance of each property in the final rankings.

Output. A separate array of top-k results is issued for each combination of weights. The top-k results are issued in JSON format, along with their rankings, aggregate scores, as well as their

respective values and scores per queried property. In addition, a *pairwise similarity matrix* is computed for each set of top-k results, based on the similarity of their respective property values.

3.2.3. Installation and usage

SimSearch is implemented in Java and the source code is publicly available under the Apache License 2.0. All functionality is under a single Maven project and has been built and tested using Java JDK v.1.8.

The code makes use of the following core libraries:

- *Guava*[31] is a set of core Java libraries that includes new collection types (such as multimap and multiset), immutable collections, and many other utilities. In SimSearch, it assists in maintaining priority queues of entities with their estimated (lower, upper, average) bounds on aggregate scores.
- *Java-string-similarity*[32] provides implementation of various string similarity and distance algorithms: Levenshtein, Jaro-winkler, n-Gram, Q-Gram, Jaccard index, etc. used in categorical search over sets of strings.
- *Java Topology Suite* (JTS)[33] is an API of 2D spatial predicates and functions, conforming to the OGC Simple Features Specification for SQL for vector geometries (points, lines, polygons). In SimSearch, this offer all required functionality for ingesting point locations and building in-memory R-trees.
- *Apache Commons DBCP* software implements Database Connection Pooling. In SimSearch, this is used for managing JDBC connections to databases.
- *Apache Commons CSV* enables reading and writing files in variations of the Comma Separated Value (CSV) format. This is used for ingesting data files available in CSV format.
- *Jackson-core*[34] includes implementation of handler types (parser, generator) that handle JSON format. This is used for issuing top-k results in JSON format, as well as for reading configuration files.

To install and use the SimSearch component, the following steps should be performed:

Step 1. Download or clone the source code:

```
git clone https://github.com/smartdatalake/simsearch.git
```

Step 2. Open a terminal inside the root folder and compile with Maven by running:

```
mvn clean package
```

[31] https://github.com/google/guava

[32] https://github.com/tdebatty/java-string-similarity

[33] https://github.com/locationtech/jts

[34] https://github.com/FasterXML/jackson-core

Step 3. Edit the parameters for the various data sources and their queryable properties in a JSON file (e.g., *sources.json* mentioned in Section 3.2.2). Typically, this is carried out by the system administrator once, unless there are changes in the data sources or their contents.

Step 4. Edit the parameters in a JSON file that specifies the attributes involved in the top-k similarity search and the query values. Such a JSON looks like the *query.json* mentioned in Section 3.2.2.

Step 5. Invoke the SimSearch software by running:

```
java -jar target/simsearch-0.0.1-SNAPSHOT-jar-with-dependencies.jar
```

Next, users can choose a number corresponding to a functionality they want to apply:

- 1: This invokes method *mount()* mentioned in Section 3.2.2 and enables specification of the queryable attributes and (if necessary) builds suitable indices on their values. The user must also specify the path to a JSON file (as in *sources.json* file listed above) containing the specification of data sources and attributes. This operation must be applied before any queries are submitted.
- 2: This option calls method *delete()* (Section 3.2.2) and thus disables properties from querying; these or other properties may be enabled again using functionality (1) without shutting down.
- 3: This triggers the API method *listDataSources()* and returns a list of the currently queryable properties and the type of operation (categorical, numerical, or spatial) supported for each one in order to specify or verify a similarity search request.
- 4: This option allows specification of a top-k similarity search query by calling the *search()* method in the API. The user must provide the path to a JSON file containing the query specification and the weight parameters (like the *query.json* file listed in Section 3.2.2). Results are issued in JSON format.

3.2.4. Experimental evaluation

Experimental Setup. To evaluate our implementation for SimSearch, we conducted experiments using a publicly available real-world dataset of news articles extracted from the GDELT project[35]. We collected all 5,514,963 articles available for January 2019. Each article represents an enriched geospatial entity associated with a spatial property (**G**eographical coordinates), a numerical one (**T**imestamp), and three set-valued properties (**P**erson names, **O**rganization names and **L**ocation names).

We ran tests considering that (i) all data may be *ingested* (from original CSV files) in memory; (ii) all data is queried *in-situ* (using JDBC connection to a local PostgreSQL/PostGIS database); and (iii) a *mixed* situation, where some properties are ingested and others can be queried in-situ.

[35] https://www.gdeltproject.org/data.html/

We randomly selected 100 articles as queries for each test. For each query, we retrieve its top-k results using the algorithms *TA*, *NRA* and *PRA*, varying the values of parameters *k* and *M*, as well as the number *m* of queried properties. We consider that all properties are equally weighted for the aggregate scores. Parameter values are listed in Table 4; default values are shown in bold.

Table 4: Parameters used for testing the SimSearch functionality.

Parameter	Values
Result size *k*	5, **10**, 15, 20
Ranked list size *M*	500 · *k*, **1000 · *k***, 2000 · *k*
Queried attributes	⟨G, T⟩, **⟨G, T, P⟩**, ⟨G, T, P, O⟩, ⟨G, T, P, O, L⟩

Algorithms are compared in terms of (i) their *average query response time* and (ii) accuracy of results. For the latter, we use *Normalized Discounted Cumulative Gain* (*NDCG*), which is a standard measure in Information Retrieval for evaluating the effectiveness of search engine algorithms [17]. Given a ranked list of *k* items, where rel_i denotes the relevance score of its i^{th} item, the *Discounted Cumulative Gain* (DCG) is defined as follows:

$$DCG = \sum_{i=1}^{k} \frac{2^{rel_i} - 1}{log_2(i+1)}$$

NDCG is obtained by normalizing the above score with respect to the one obtained by the ideal ranking of the items, denoted as IDCG:

$$NDCG = \frac{DCG}{IDCG}$$

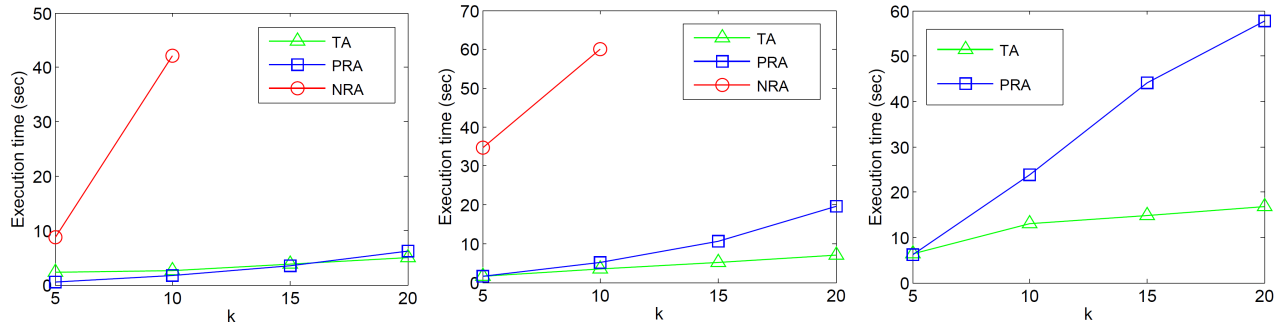
NDCG takes values between 0 and 1, with higher values indicating better quality (1 denoting the score of the ideal ranking). Intuitively, NDCG captures the assumption that items with higher relevance should appear earlier in the top-k results. In our case, the relevance score rel_i of each item *i* corresponds to its exact similarity score to the given query. Hence, we consider the results obtained by *TA* as the ideal ranking, since *TA* accurately computes the similarity score of each item thanks to the random accesses performed for all encountered items. Then, we compute the NDCG score for the top-k results obtained by *PRA* and *NRA* by normalizing them with *TA*'s IDCG. Plots show the average NDCG over all executed queries.

The experiments were conducted on a server with Intel Xeon E5-2420 v2 CPU with 2.20GHz processor and 64GB RAM running Ubuntu.

Experimental Results. In the first set of experiments, we consider queries over *m* = 3 attributes ⟨G, T, P⟩, and we measure the average *response time per query* for each algorithm with varying *k* and list size *M*. Figure 25 depicts results when data for all properties is ingested in memory; Figure 26 shows results when all data is queried in situ (in a PostgreSQL database); finally, Figure 27 displays

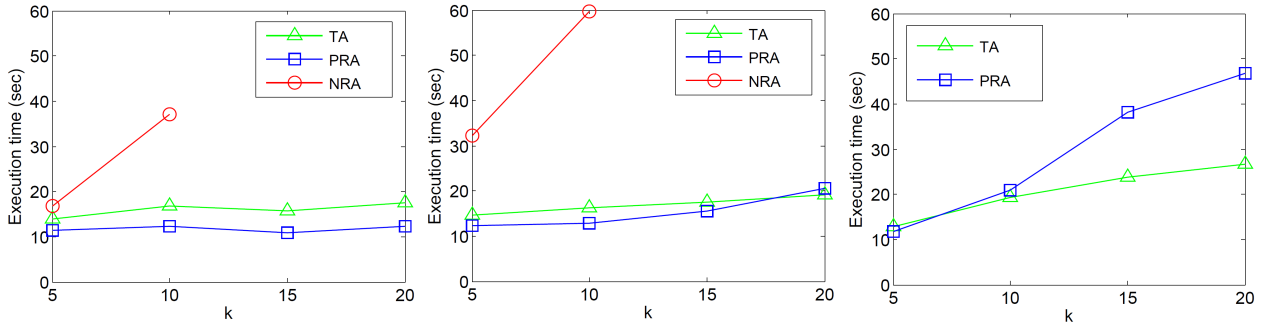
the case when some properties (geospatial coordinates *G*, numerical timestamp *T*) are queried in situ and another (categorical) property is ingested (names of persons *P*).

Regardless the mode of operation for SimSearch, execution cost for *NRA* soars with increasing values of *k*, even for moderate list sizes. Clearly, for $k > 10$ the overhead for *NRA* to answer each query exceeds the timeout limit of 60 sec, which we have set as the maximum allowable response time. When the list sizes become larger, *NRA* cannot produce results in time even for k = 5 (as shown in Figure 25(c),Figure 26(c), andFigure 27(c)), since this method has to perform too many iterations while also continuously refining the bounds of all seen entities. In contrast, *TA* and *PRA* fare much better and manage to return the requested top-k results in all cases with no timeout. Specifically, *TA* provides results quite fast, thanks to its unrestricted random access to the lookup tables preconstructed over the entire attribute data. However, if no full random access is possible against the data sources, *TA* is no longer applicable. In this case, *PRA* can be used instead, allowing random access to missing attribute values from the ranked lists only, while still offering acceptable performance. As Figure 25(a) testifies, *PRA* is competitive to *TA* for ingested data and small list sizes, as it can issue top-k results with a comparable number of iterations, whilst probing much smaller lookup tables. However, for larger list size *M*, the overhead to build these lookup tables and estimate score bounds incurs an almost linear increase on response times per query for *PRA* as shown in Figure 25(b) and Figure 25(c). Query response times are generally similar when all (Figure 26) or part of the data (Figure 27) is queried in-situ (in PostgreSQL). Not surprisingly, for smaller list sizes ($M = 500 \cdot k$), *PRA* generally overtakes *TA*, since the size of lookup tables is much smaller, whereas repeated random accesses by *TA* to the database incurs extra overhead. Instead, when the size *M* of ranked lists grows, *PRA* falls behind *TA*. Indeed, despite its reduced overhead for random access, *PRA* needs to perform many more iterations over the ranked lists compared to those with *TA*. For many queries, PRA often exhausts the ranked lists, whereas *TA* finishes after examining much fewer candidates. Regarding its mode of operation, it is clear that the cost is reduced when all data is ingested in memory (Figure 25). Thanks to already built indices for each query type (numerical, spatial, categorical) involved in the top-k similarity search, our implemented modules manage to offer the final results with reasonable latency. But when some (Figure 26) or all data (Figure 27) are queried in situ, latency generally increases, although not at a prohibitive level. From closer inspection of the query workload, we noticed that the extra overhead is due to the textual search performed over categorical attributes (in this test, property P). Despite the availability of a GIN index built over this property, an additional cost is incurred by the not natively supported Jaccard distance computation between the query value (set of keywords) and the property values filtered by the index (Figure 26). When this categorical property is ingested while the rest are queried in situ (Figure 27), average response time per query slightly improves especially for smaller list size *M*, although it remains higher than the query cost against fully ingested data (Figure 25).



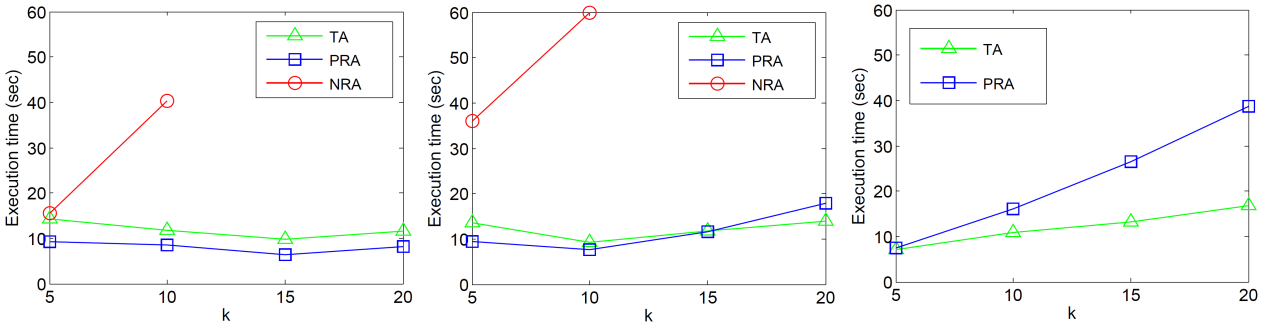
(a) $M = 500 \cdot k$ (b) $M = 1000 \cdot k$ (c) $M = 2000 \cdot k$

Figure 25: Average query response time for similarity search against $m = 3$ attributes ⟨G, T, P⟩ from *ingested* data (CSV) with varying k and M.



(a) $M = 500 \cdot k$ (b) $M = 1000 \cdot k$ (c) $M = 2000 \cdot k$

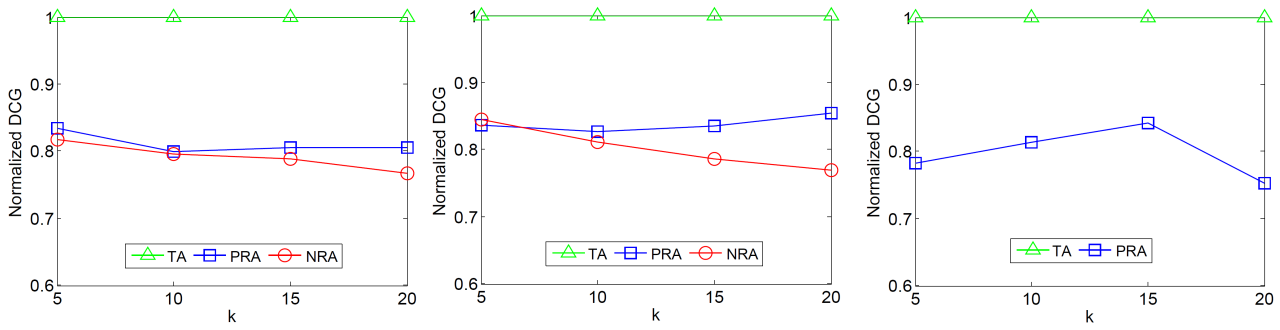
Figure 26: Average query response time for similarity search against $m = 3$ attributes ⟨G, T, P⟩ queried *in-situ* (Postgres) with varying k and M.



(a) $M = 500 \cdot k$ (b) $M = 1000 \cdot k$ (c) $M = 2000 \cdot k$

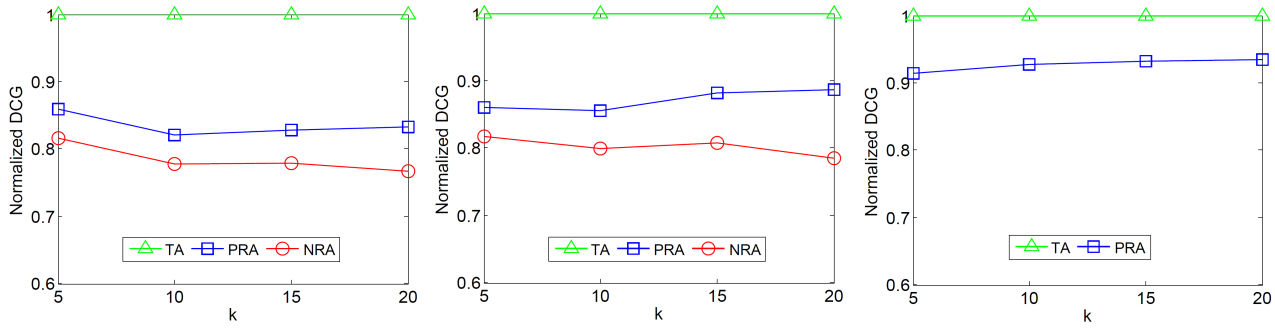
Figure 27: Average query response time for similarity search against $m = 3$ properties ⟨G, T, P⟩ available *in-situ* (G, T) and *ingested* (P) with varying k and M.

Regarding *quality* of the returned top-k results, we examine the *average NDCG per query* for each ranking algorithm under the same settings as in the previous set of experiments, i.e., varying k and M values for the same m=3 queried properties. Figure 28 concerns quality of results when data for all properties is ingested in memory; Figure 29 when all data is queried in situ; and Figure 30 when two properties are queried in situ and one property is ingested. As already mentioned, *TA* provides the ideal ranking by definition, hence its results are considered as ground truth for assessing those received from the other two methods. In all parametrizations and modes of operation tested, *PRA* maintains a stable quality exceeding 80% (and sometimes even 90%) relevance to the accurate results obtained from *TA* for varying k and list size M. The only exception is shown in Figure 28(c) and occurs when $k = 20$ for the largest list size tested ($M = 40000$) for fully ingested data, because some of the tested queries require too many iterations and are eventually suspended due to timeout (the average response time is very close to 60 sec as shown in Figure 25(c)). Regarding *NRA*, Figure 25(a) and Figure 25(b) indicate that its quality against fully ingested data is comparable to that of PRA for $k \leq 10$, i.e., when queries manage to return results before timing out. When $k > 10$ results are requested, many queries fail to complete, thus their quality deteriorates with respect to those from *PRA*. However, when some (Figure 29) or all data (Figure 30) is queried in situ, PRA seems consistently better than NRA. This is due to the fact that NRA has significant overhead for continuously maintaining bounds for scores for all seen entities, hence in general it times out without performing enough iterations over the ranked lists and has not yet issued all top-k results. Missing results are complemented with entities having the largest LB on their scores, which may diverge from the accurate ones (issued by *TA*) or those based on average bounds MB on scores (issued by *PRA*).



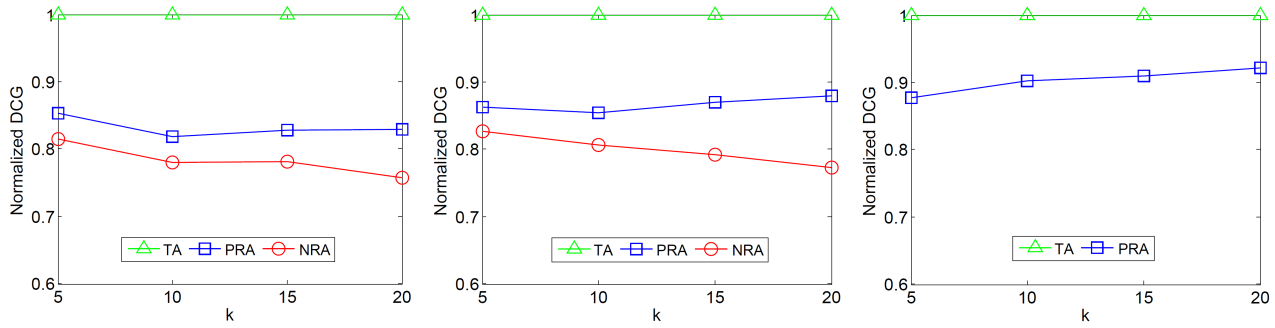
(a) $M = 500 \cdot k$ (b) $M = 1000 \cdot k$ (c) $M = 2000 \cdot k$

Figure 28: Quality of top-k results from SimSearch for $m = 3$ queried properties ⟨G, T, P⟩ over *ingested* data (CSV) with varying k and M.



(a) $M = 500 \cdot k$ (b) $M = 1000 \cdot k$ (c) $M = 2000 \cdot k$

Figure 29: Quality of top-k results from SimSearch for $m = 3$ properties ⟨G, T, P⟩ queried *in-situ* (Postgres) with varying k and M.

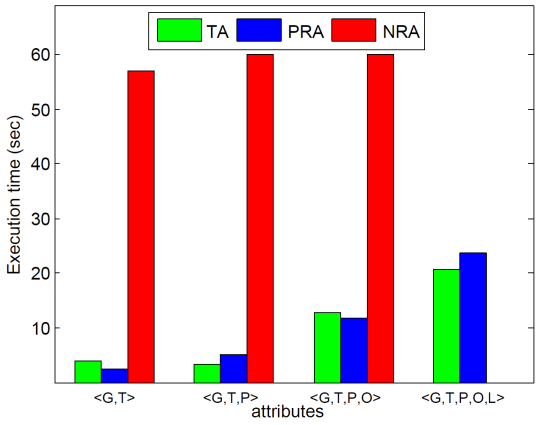


(a) $M = 500 \cdot k$ (b) $M = 1000 \cdot k$ $M = 2000 \cdot k$

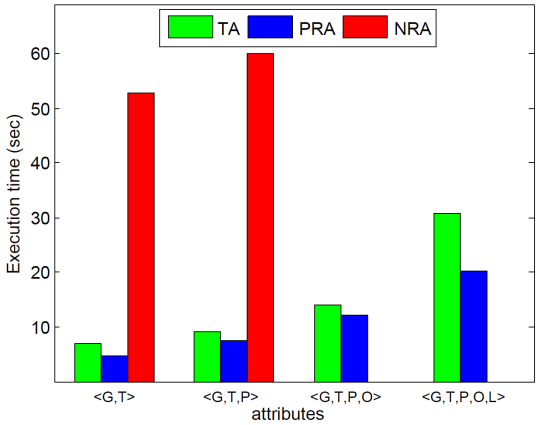
Figure 30: Quality of top-k results from SimSearch for m = 3 queried properties ⟨G, T, P⟩ available *in-situ* (G, T) and ingested (P) with varying k and M.

In the last set of experiments, we evaluate the algorithms when *different sets of properties* are queried, fixing $k = 10$ and $M = 1000 \cdot k$. We examine two cases: when all data is ingested, and when two properties (G,T) are queried in situ while the remaining categorical ones (P,O,L) are ingested. Regarding average response times per query depicted in Figure 31(a) and Figure 32(a), it is clear that *NRA* has significant overhead even when only two attributes are queried, regardless of the mode of operation. If queries involve $m > 2$ properties, most of them time out before all necessary iterations are performed. In contrast, TA and PRA take much less time even for increasing number of properties, although the cost is generally higher when some properties reside in DBMS and are queried in situ (Figure 31(b)). A similar effect can be observed in the plots concerning the quality of results obtained by *NRA*, which are consistently inferior to those from the other two methods as shown in Figure 32(b). Indeed, *TA* provides accurate top-k results with generally minimal overhead, provided of course that random access to full property lookups is possible. Otherwise, *PRA* seems a good alternative with a competitive execution cost and generally acceptable quality of the returned top-k results.

Overall, it seems that *PRA* manages to alleviate the prohibitive execution cost of *NRA*, while also returning results of better quality. Although such results are clearly less accurate, they can still be issued almost as fast as those from *TA* without requiring random access to all attribute values.

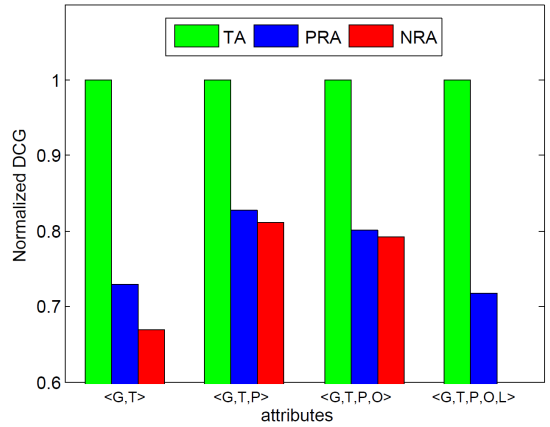


(a) Only ingested data

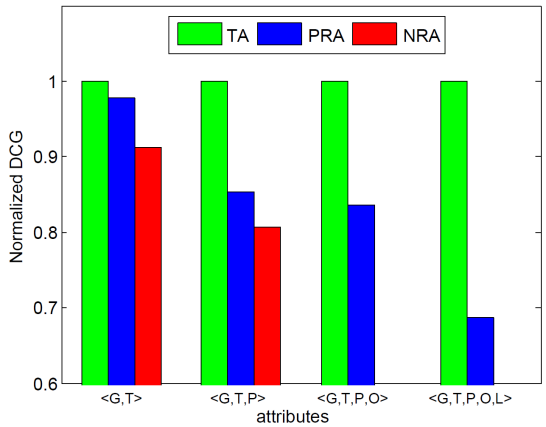


(b) In-situ (G,T) and ingested (P,O,L) data

Figure 31: Query response time of SimSearch against varying queried properties for k = 10 and M = 10000.



(a) Only ingested data



(b) In-situ (G,T) and ingested (P,O,L) data

Figure 32: Quality of top-k results from SimSearch against varying queried properties for k = 10 and M = 10000.

3.3. Similarity search over time series

3.3.1. Implemented functionalities

The SimSearchTS component provides functionality for similarity join and search on time series data, based on local similarity, as defined in Deliverable D3.1, Section 3.4. It also supports two transformations useful for exploring and smoothing time series data, specifically z-normalization and Piecewise Aggregate Approximation (PAA) generation. In the following, we present some implementation details regarding the functionalities of the SimSearchTS component.

Read Input Data: The input data are read from a file. As a prerequisite, the data have to be pre-processed and stored in CSV format, containing one time series per line. The first column contains the ID of each time series and the subsequent ones contain the per-timestamp values. In order for the self-join (also referred to as pair discovery) to function properly, the time series must be co-evolving, i.e., aligned in the time axis.

Z-Normalization: This function z-normalizes the data. This procedure ensures that the mean of the output time series is approximately 0 while the standard deviation is in a range close to 1. It is calculated using the formula below:

$$x'_i = \frac{x_i - \mu}{\sigma} \qquad i \in \mathbb{N}$$

where μ is the mean value of the time series and σ is the standard deviation.

PAA: This function generates the Piecewise Aggregate Approximation of all given time series [18]. PAA is essentially a dimensionality reduction technique for time series, which is calculated by splitting a time series in equal-size segments and obtaining the mean value for each segment. Assuming a given time series $X = \{X_1, X_2, \dots, X_k\}$, where X_i is the value at the i-th timestamp and k its length, this function calculates its PAA using the following formula:

$$\overline{X}_i = \frac{M}{k} \cdot \sum_{j=k/M(i-1)+1}^{(k/M)i} x_j \qquad i, j \in \mathbb{N}$$

where M is the length of the obtained vector. Thus, the obtained PAA for a time series is a vector $\overline{X} = \{\overline{X_1}, \overline{X_2}, \dots, \overline{X_k}\}$. This function calculates the PAA for all given time series.

Build Index: This function builds an index on the given co-evolving time series, which is used to execute the self-join (pair discovery) and similarity search. As mentioned in Deliverable 3.1, Section 3.4, to reduce the candidate pairs that need to be checked at each timestamp during self-join, we discretize the values of all time series in bins of size ε. The time series index is essentially a tree-map (i.e., sorted keys), where, for each timestamp, we store a hash-map containing the generated bins as keys, each containing the corresponding time series values. Time series with values within the same bin at any timestamp form candidate pairs. To avoid false negatives, we need to check adjacent bins for additional candidate pairs whose values differ by at most ε. Time series having

values at non-adjacent bins are certainly farther than ε at that specific timestamp, so we can avoid these checks. Figure 33 depicts this process for a specific timestamp.

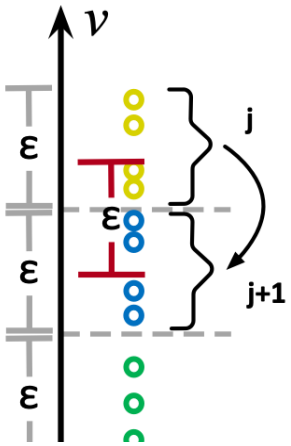


Figure 33: Discretization of time series values at timestamp t.

However, using the above procedure, the calculation of self-joins would require a new index to be built for different ε threshold values. To overcome this, we build the index using an initial ε_0 value and then we can compute any self-join process with different ε thresholds, using the same computed and loaded index. We can do this by simply considering the following two cases during calculation:

- $\boldsymbol{\varepsilon \leq \varepsilon_0}$: In this case, we can only find pairs in adjacent bins, so the procedure takes place as previously.
- $\boldsymbol{\varepsilon > \varepsilon_0}$: In this case, we may find results also in non-adjacent bins, so, it suffices to check for candidate pairs within the next $\lceil \varepsilon / \varepsilon_0 \rceil$ bins.

A trade-off of this procedure can be detected in the case that the given ε threshold is slightly larger by a multiple of ε_0, when a larger number of candidates has to be checked. However, this ensures no false negatives and the slightly larger computation time is negligible compared to the time needed to rebuild the index for each new self-join operation with a different ε threshold value.

Self-Join (Pair Discovery): This function performs the self-join operation on the given co-evolving time series dataset. Given the ε (value threshold) and δ (duration threshold) parameters as input, the process calculates and returns for each time series within the given dataset, all the locally-similar (see Deliverable 3.1, Figure 18) time series among the rest, along with the corresponding similarity intervals. The followed procedure is described in detail in Deliverable 3.1, Section 3.4.3.

Local Similarity Search: This function operates on the results of self-join and retrieves all the locally-similar time series to a given query time series ID along with the corresponding similarity intervals, according to the pre-specified threshold ε. Specifically, the resulting pairs are scanned and only the ones that contain the query time series ID as their first or second member are kept and stored in the corresponding variable.

Store Results: This function stores the self-join or local similarity results in a JSON file within a specified file path.

3.3.2. API

In the following, we provide details regarding the API of the SimSearchTS component. Specifically, we go through all the offered functions and describe the input and output format for each one, as well as the parameters they require. The component is developed in Java. All the necessary input parameters are read from the corresponding JSON configuration files. The Javadoc for the component along with the source code can be found on GitHub[36].

Read Input Data: This function is provided by the `ReadInput` class. It contains the following public method:

- readInput(String dataFile, int tsLength)

In this method, `dataFile` is the path to the pre-processed (see Section 3.3.1) input file containing the co-evolving time series and `tsLength` is the length of the time series (all must be of the same length). The method returns a `HashMap<String, double[]>` where each time series is in the form of a double array indexed to its corresponding ID.

Z-Normalization: This function is provided by the `Normalization` class. It contains the following public method:

- zNormalize(HashMap<String, double[]> data)

In this method, `data` is a collection of co-evolving time series read from a file. The method returns a `HashMap<String, double[]>` containing the normalized time series along with their corresponding IDs.

PAA Generation: This function is provided by the `PAAGeneration` class. It contains the following public method:

- generatePAA(HashMap<String, double[]> data, int numSeg)

In this method, `data` is a collection of co-evolving time series read from a file and `numSeg` is the number of PAA segments to be generated. The method returns a `HashMap<String, double[]>` containing the generated PAA vectors along with their corresponding IDs.

Build Index: This function is provided by the `BuildIndex` class. It contains the following public method that builds the index:

- buildIndex(HashMap<String, double[]> data, double epsilonInit, int tsLength)

In this method, `data` is a collection of co-evolving time series read from a file, `epsilonInit` is the ε_0 value for building the index (see Section 3.3.1) and `tsLength` is the length of the co-evolving time series. The method returns a `TreeMap<Integer, HashMap<Short, ArrayList<IndexEntry>>>`, which constitutes the index. Each timestamp (`Integer`) points to a HashMap (`index`), where each bin number (`Short`) points to an `ArrayList` of `IndexEntry` instances, containing the time series values for that bin.

Self-Join (Pair Discovery): This function is provided by the `SelfJoin` class. It contains the following public method that performs the self-join:

[36] https://github.com/smartdatalake/time_series_explorer

- discoverPairs(TreeMap<Integer, HashMap<Short, ArrayList<IndexEntry>>> index, double epsilon, double epsilonInit, int delta, int tsLength)

In this method, `index` is the previously generated time series index, epsilon is the ε threshold of the query, `epsilonInit` is the ε_0 used to build the index, `delta` is the δ parameter for the query and `tsLength` is the length of the co-evolving time series. The method returns a `HashMap<Set<TimeSeries>, ArrayList<int[]>>`, where the time series that form pairs are returned as a `Set` and each pair points to an `ArrayList` of intervals where the local similarity is detected, in the form of an `int` array.

Local Similarity Search: This function is provided by the `LocalSimSearch` class. It contains the following public method that performs the local similarity search:

- simSearch(HashMap<Set<TimeSeries>, ArrayList<int[]>> pairs, String query)
- In this method, `pairs` contains the discovered locally-similar time series pairs during self-join, and query contains the ID of the time series for which similarity search is run. The method returns a `HashMap<Set<TimeSeries>, ArrayList<int[]>>`, where the query time series along with each locally-similar time series are paired as a `Set` which points to an `ArrayList` of intervals where the local similarity is detected in the form of an `int` array.
- **Store Results**: This function is provided by the `StoreResults` class. It contains the following public method:
- storeResults(HashMap<Set<TimeSeries>, ArrayList<int[]>> results, String resultFile)

In this method, `results` contain all the results obtained either by self-join or local similarity search and `resultFile` is the path to the file where they will be stored.

3.3.3. Installation and usage

In the following, we provide details regarding the installation and usage of the SimSearchTS component. It is implemented in Java as a Maven project. Thus, to build the library, it suffices to simply run the following command via a terminal (from within the folder where the pom.xml file and source code is located):

```
mvn clean package
```

This will generate the following JAR file within the `target` directory:

simSearchTS-0.0.1-SNAPSHOT-jar-with-dependencies.jar

The SimSearchTS component has dependencies to the following existing libraries, which are automatically imported by Maven during building:

- commons-io v2.6
- jackson-databind v2.11.0
- jackson-core v2.11.0
- json-simple v1.1

The component can be executed as a standalone application. This is done by placing the component's JAR file in a folder and executing the following command:

```
java -jar simSearchTS-0.0.1-SNAPSHOT-jar-with-dependencies.jar
```

A JSON configuration file for each functionality must be provided when prompted. In the following, we present examples for each operation, along with its output format.

Read Input Data: This function requires a JSON configuration file that has the following format:

```
{
      "tsLength":"k",
      "logFile":"run.log",
      "dataFile":"input.csv"
}
```

The file under `logFile` contains possible logs during the execution, while the input file under `dataFile` contains the input co-evolving time series. The input file is a CSV file that must contain one time series per line in the following format:

```
TS1, X1, Y1, v1_1, v1_2, v1_3, …, v1_k
TS2, X2, Y2, v2_1, v2_2, v2_3, …, v2_k
…
TSn, Xn, Yn, vn_1, vn_2, vn_3, …, vn_k
```

In the above, `Xn` and `Yn` are two separate attributes per time series that are ignored by all the operations in this component (i.e., they are used to denote coordinates in case the time series are geolocated). The first column is each time series ID and the rest of the columns (`vn_1, vn_2, vn_3, …, vn_k`) are the time series values. In this example, we have a total of `n` time series of length `k`. A sample input file is included in the GitHub repository, along with the source code for testing purposes.

Z-Normalization: This function operates on input data read during the previous step and does not require a configuration file. Its output can be stored in a file with the same format as the input.

Piecewise Aggregate Approximation (PAA) Generation: This function requires a JSON configuration file that has the following format:

```
{
   "noSeg":"S"
}
```

The function calculates the PAA of the given input time series using the given `noSeg` and its output can be stored in a file with the same format as the input.

Build Index: This function builds the time series index and requires a JSON configuration file that has the following format:

```
{
      "tsLength":"k",
      "epsilonInit":"e_0"
}
```

The function builds the index using the given `tsLength` and `epsilonInit` parameters.

Self-Join (Pair Discovery): This function requires a JSON configuration file that has the following format:

```
{
      "tsLength":"k",
      "epsilonInit":"e_0",
      "epsilon":"e",
      "delta":"d
}
```

The function performs self-join using the provided `tsLength`, `epsilonInit`, `epsilon` and `delta`. The results can be stored as a JSON file and have the following format:

```
{
   "Pair 1": {
      "Member 1": "ID1",
      "Member 2": "ID2",
      "Interval 1": "[i1_1, i1_2]",
      "Interval 2": "[i2_1, i2_2]"
   },
   "Pair 2": {
      "Member 1": "ID3",
      "Member 2": "ID4",
      "Interval 1": "[i1_1, i1_2]"
   }
}
```

Specifically, `Member 1` and `Member 2` contain the IDs of the time series that form a pair. The intervals where the local similarity is detected are labelled as `Interval 1`, `Interval 2`, ..., `Interval X` and are assigned to the same JSON object.

Local Similarity Search: This function requires a JSON configuration file that has the following format:

```
{
   "query":"qID"
}
```

The function operates on the results of self-join, by selecting those detected pairs that contain a member ID that is the same as the given `query`. The results can be stored as a JSON file and have the following format:

```
{
   "Result 1": {
      "Query": "qID",
```

```
        "Result": "ID1",
        "Interval 1": "[i1_1, i1_2]",
        "Interval 2": "[i2_1, i2_2]"
    },
    "Result 2": {
        "Query": "qID",
        "Result": "ID2",
        "Interval 1": "[i1_1, i1_2]"
    }
}
```

Specifically, for each detected result, `Query` contains the ID of the query and `Result` contains the ID of the time series that are similar to the query. The intervals where the local similarity is detected are labelled as `Interval 1`, `Interval 2`, ..., `Interval X` and are assigned to the same JSON object.

Store Results: This function requires a JSON configuration file that has the following format:

```
{
    "resultFile":"results.json"
}
```

The function stores the self-join or similarity search results in a file, whose path is determined by the parameter `resultFile`.

3.3.4. Experimental evaluation

In this section, we present an experimental evaluation of the SimSearchTS component. We conducted two sets of experiments, using a dataset containing co-evolving stock data from SPRING and a synthetic dataset. The stock dataset was used for qualitative and quantitative assessment of our local similarity self-join algorithm, while the synthetic dataset was used for efficiency evaluation. We compare our checkpoint (referred to as CP) approach versus a sweep line (referred to as SL) method that has to check every timestamp in order to retrieve candidate pairs.

3.3.4.1. Experimental Setup

Table 5 lists the datasets used in our experiments. Next, we describe their characteristics.

Table 5: Datasets used in the experiments.

Dataset	Size	Time Series Length
Stock	12,232	1,566
Synthetic	50,000	1,000

Stock dataset: We extracted the stock dataset from SPRING's historical data. Specifically, we harvested all available longer daily histories for each stock. Each file contains the daily (only for working days, 261 in total per year) closing values of a specific stock. Then, we extracted for each file (stock) only the daily closing values for years 2007-2012, thus, obtaining time aligned time series of length 261 (working days) x 6 (years) = 1,566.

Synthetic Dataset: We generated a synthetic dataset of 50,000 time series, each with a length of 1,000 timestamps (50 million data points in total). To generate the dataset, we followed the procedure described in [18].

All experiments were conducted on a Dell PowerEdge M910with 4 Intel Xeon E7-4830 CPUs, each containing 8 cores clocked at 2.13GHz, 256 GB RAM and a total storage space of 900 GB.

3.3.4.2. Evaluation results for stock data

Initially, we performed experiments on the local similarity self-join algorithm, comparing the CP and SL approaches. In the following, we discuss the results obtained for various parameter values. The dataset was z-normalized to eliminate amplitude discrepancies among time series and focus on structural similarity. Table 6 lists the range of values for the parameters used (default values are in bold). These ranges of values were determined after several preliminary tests, ensuring that the algorithms return a reasonable number of results. Parameter δ is expressed as a percentage of the duration of the time series and ε is expressed as a percentage of the value range (i.e., difference $max - min$ in values encountered across the dataset). Notice that even though the time series are z-normalized (the values are expressed as standard deviations from the mean, which is zero), the value range is rather large (approximately 55.6 standard deviations). This is due to the fact that in the stock dataset there are a few stocks with either really low (close to zero) or really large closing values, thus generating this large value interval. However, the vast majority of the stocks behaves normally, with value ranges of up to two or three standard deviations around zero. Thus, the value range percentages that we chose for ε are rather small.

Table 6: Parameters for tests over the stock dataset.

Parameter	Values
δ (% of time series length, i.e., 1,566)	2% - 3% - **4%** - 5% - 6%
ε (% of value range, i.e., approx. 55.6)	0.02% - 0.04% - **0.06%** - 0.08% - 0.1%

Varying ε. Figure 34 depicts the performance results for several executions of the local similarity self-join algorithm, each with a different ε threshold value. For small ε values of up to 0.08% of the value range, the CP algorithm returns results within a few seconds. The SL approach is much slower, with its performance deteriorating more rapidly with increasing ε values. As expected, the number of results is growing as we loosen the ε threshold (i.e., larger values), especially for ε equal to 0.1% of the value range.

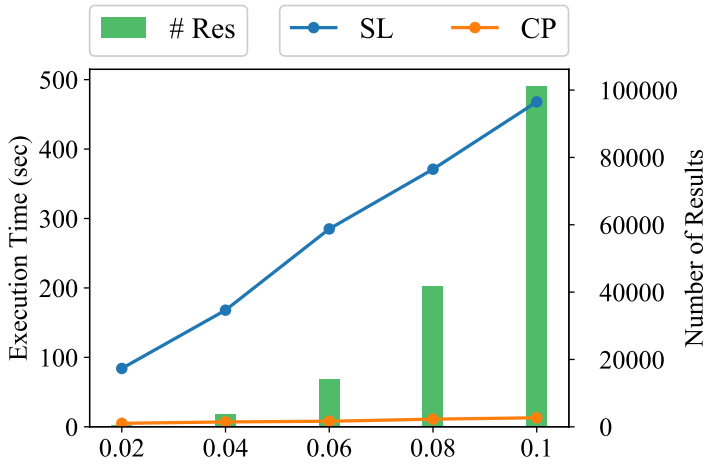


Figure 34: Assessment against stock data for varying ε.

Varying δ. Figure 35 depicts the results for the execution performance of the local similarity self-join algorithm, this time for different δ threshold value ranges. In this case, the threshold is getting tighter as the δ value increases, as it is harder to detect qualifying pairs that last longer. Thus, more candidate pairs are pruned during the procedure. However, in both cases, the execution time reduction is rather slow. This is due to the fact that, despite most candidate pairs do not qualify to be included in the final results, they are still considered candidates and have to be verified. The number of results for smaller δ values is naturally much larger, as far more pairs are expected to be verified if pairs exist. Increasing δ yields significantly less results, which leads to the conclusion that δ threshold is rather sensitive for this specific dataset.

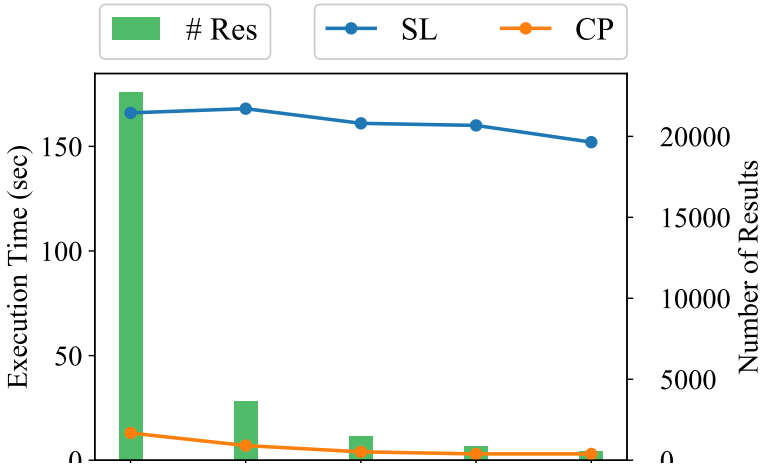


Figure 35: Assessment against stock data for varying δ.

3.3.4.3. Evaluation results for synthetic data

To evaluate the efficiency of our local similarity self-join algorithm, we used the synthetic dataset. As previously, we initially preformed preliminary tests to extract ranges of parameter values where

the algorithms return a reasonable number of results. Table 7 lists the range of values for all parameters used in the efficiency tests, with the default values emphasized in bold.

Table 7: Parameters for tests against synthetic data.

Parameter	Values
Dataset Size	10,000 - 20,000 - **30,000** - 40,000 - 50000
Time Series Length	600 - 700 - **800** - 900 - 1,000
δ (% of time series length)	2.5%
ε (% of value range)	0.2%

Varying Dataset Size. Figure 36 depicts the performance comparison between local similarity self-join CP and SL algorithms. Observe that, as the number of time series in the dataset grows, significantly more pairs are detected (green bars), hence, there is a linear increase in the execution cost for the CP algorithm. On the other hand, baseline SL requires more time, as it must check many more combinations of time series (for each consecutive timestamp). This leads to an exponential performance deterioration as the dataset size increases, taking more than 2 hours to finish for 50,000 time series for the SL approach. On the other hand, CP scales much better, managing to finish in a few minutes in the worst case (50,000 time series). Conclusively, CP is more than an order of magnitude faster than SL.

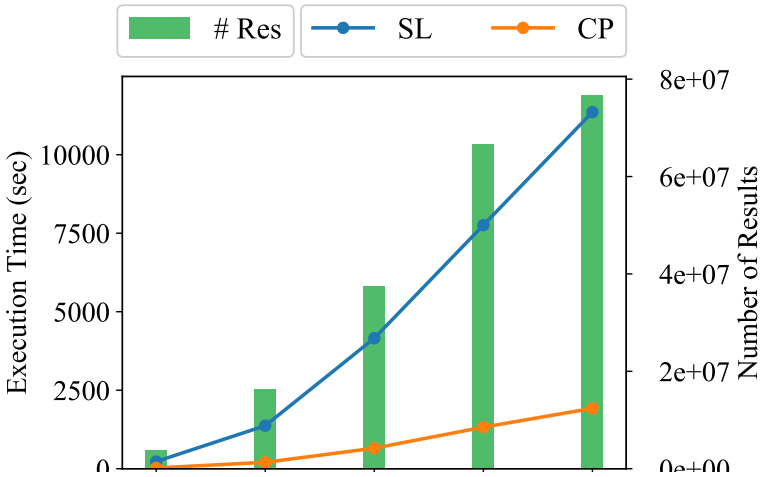


Figure 36: Efficiency for varying dataset size.

Varying Time Series Length. As depicted in Figure 37, for time series with increasing length, the CP algorithm again constantly outperforms SL. As in previous experiments, δ is expressed as a percentage of the time series length and it is by default set to 2.5% (a rather loose threshold), in order to further stress-test the methods (more candidates will be generated). As a reminder, the default dataset size is 30,000. We observe that, as the time series length (and thus δ) gets larger, the performance of both algorithms slightly worsens. Only in the case of 1,000 timestamps the

execution time starts to drop for the CP algorithm due to the even larger δ. This is not the case with the SL method, which has to evaluate more timestamps.

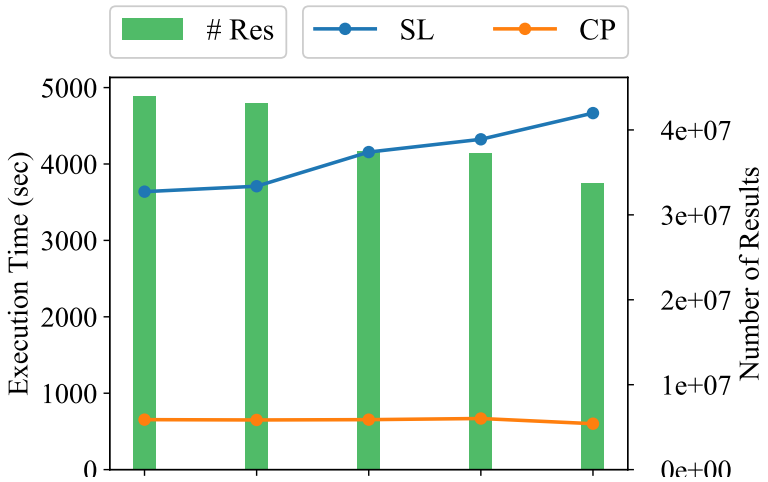


Figure 37: Efficiency for varying time series length.

4. Entity Resolution

In this section, we describe sHINER, the component we have implemented for Entity Resolution in HINs. This component identifies and links nodes in a HIN that represent the same real-world entity. We present its functionalities, explain its API, installation and usage, and conduct an experimental evaluation. The functionality of sHINER is based on the concept of Graph Generating Dependencies, which was presented in D3.1: "Similarity search, entity resolution and ranking" (Section 4).

4.1. Implemented functionalities

Overview. A graph generating dependency (GGD) is a graph dependency of the form $Q_s[\overline{x}], \phi_s \rightarrow Q_t[\overline{x}, \overline{y}], \phi_t$ where: (1) $Q_s[\overline{x}]$ and $Q_t[\overline{x}, \overline{y}]$ are graph patterns, called **source** and **target** graph pattern, respectively, (2) ϕ_s is a set of differential constraints defined over the variables $\overline{x}$ (variables of the graph pattern Q_s), and (3) ϕ_t is a set of differential constraints defined over the variables $\overline{x} \cup \overline{y}$, in which $\overline{x}$ are the variables of the source graph pattern Q_s, and $\overline{y}$ are any additional variables of the target graph pattern Q_t.

The GGDs can encode rules/conditions to perform entity resolution based on the topology of the entities in a HIN and the similarity of their attributes. Given a set of GGDs, we generate new edges that link the same real-world entities in a HIN according to the GGDs definition. For example, given

the GGD in Figure 38, if the names of the teacher vertices x and a are similar according to a threshold and the names of the university vertices y and e are also similar according to a threshold, then there should exist a sameAs link between these vertices. In case it does not exist, then the sameAs edge will be generated.

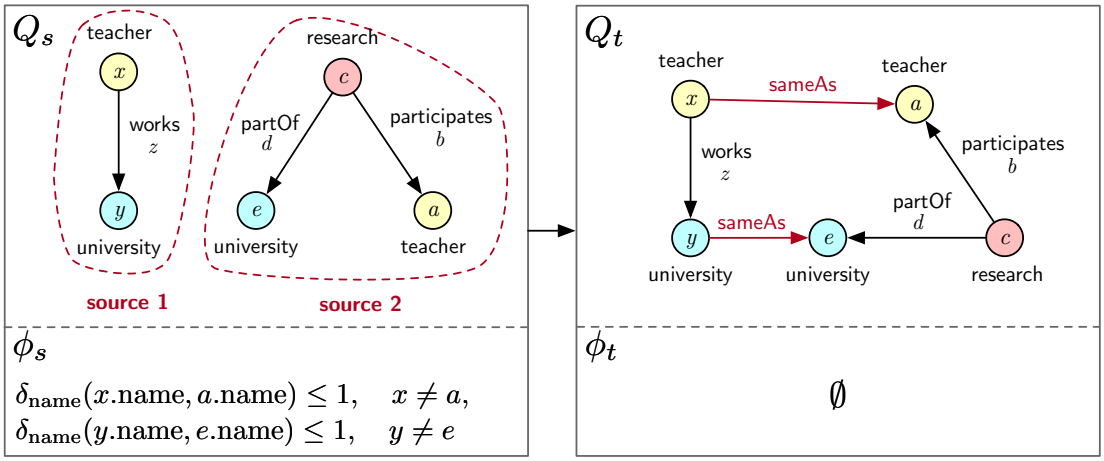


Figure 38: Example GGD.

This process consists of mainly two algorithms: (1) Validation and (2) Graph Generation. The Validation algorithm is responsible for checking if all the GGDs are valid in the graph, which means that for each match of the source graph pattern that satisfies the source constraints should exist a match of the target graph pattern that satisfies the target constraints. If the match for the target graph pattern does not exist or does not satisfy the target constraints it means that the GGD is violated. If a GGD is violated, we then invoke the Graph Generation to "fix" the GGD (turn it into a valid GGD) by typically adding new nodes/links which satisfy the target graph pattern. This process is repeated until all GGDs are valid and there are no changes in the target graph. Observe the general process of repairing GGDs in Figure 39. More details on the syntax, semantics and application of GGDs are presented in D3.1: "Similarity search, entity resolution and ranking".

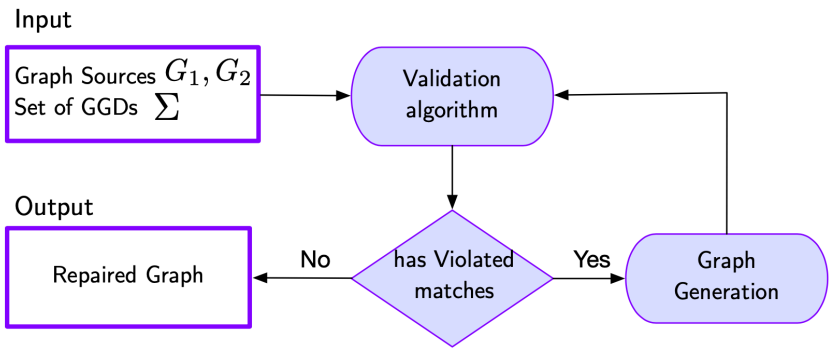


Figure 39: Repairing process of GGDs.

The implementation of the GGDs consists of two main modules: (1) the G-Core language interpreter and (2) the GGD module. Figure 40 shows the general architecture of the sHINER component.

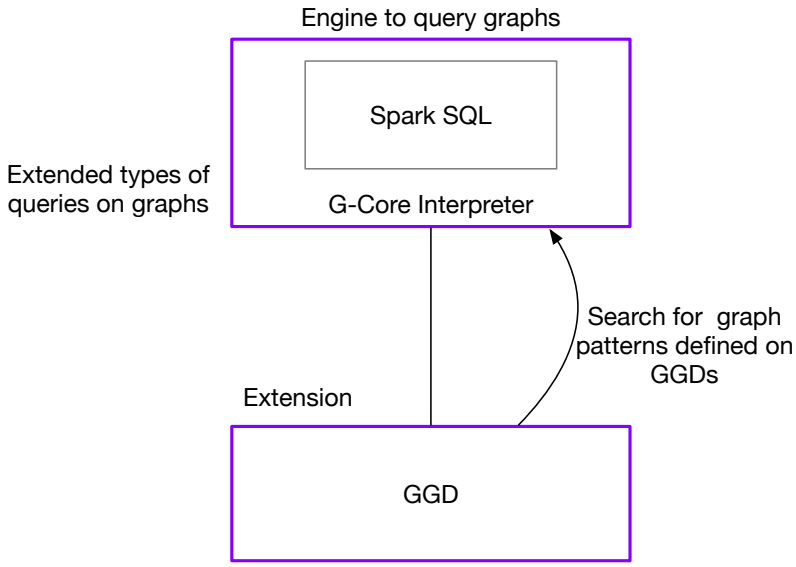


Figure 40: The Entity Resolution component.

The G-Core language interpreter is an interpreter of the G-Core graph query language built over Spark. Its implementation is open source and can be accessed in the Linked Data Benchmark Council repository[37]. 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. For example, given the graph named Social-Graph with the schema presented in Figure 41, the G-Core query *CONSTRUCT (a) MATCH (a:teacher)-[b:works]->(c:university) ON Research-Graph* will return a graph containing all the vertices labelled Teacher that works in a University. More information on the G-Core query language is available in [19].

[37] https://github.com/ldbc/gcore-spark

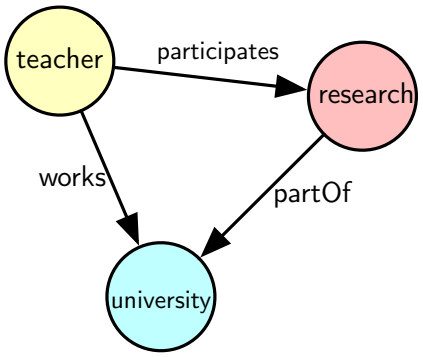


Figure 41: Example graph schema.

The G-Core project interprets graph queries written in G-Core language and rewrites the queries into SQL which are executed using Spark SQL. In order to give support to all types of queries that the GGD module needs, we extended the G-Core project with new functionalities following the proposed language. The mainly extended operations from the G-Core project were:

- SELECT operator – Returns a tabular projection of the graph matching clauses;
- UNION operator – Used to union the result graph of a query with another graph available in the database;
- OPTIONAL operator – Similar to SPARQL optional queries for RDF, the OPTIONAL operator in G-Core is used when the user needs extra piece of information added to the solution when the information is available, but when it is not available the solution is not rejected. An example of OPTIONAL query in G-Core is: SELECT * MATCH (t:Teacher) OPTIONAL (t)-[p:participates]->(r:research) ON Research-Graph. This query's result is a tabular projection of all matches of the vertex teacher and, in case a vertex teacher participates in a research this extra information will also be added to the tabular projection.

The GGD module is an extension of the G-Core project which implements the GGDs operations presented in Deliverable D3.1 for Validation and Graph Generation. The GGD module uses the G-Core project to query the graph patterns defined in the input GGDs and is responsible for checking which GGDs are valid and repairing the GGDs that are violated by using Graph Generation. To perform entity resolution, the graph generation function generates 'sameAs' edges to represent the entity resolution result.

The G-Core and the GGD are the core modules for the Entity Resolution component. Further extensions to this component are modules to integrate with other components of the project, for example, visualization (WP4). We are currently studying the possibilities of integration with these two components and further information will be available in Deliverable D1.4. The next sections provide details on the component's API, input and output formats and its installation and usage.

4.2. API

sHINER can be run by using a standalone jar file. In this section, we provide details on the input and output format of the component as well as the required parameters. sHINER was developed in Scala language and uses the Apache Spark framework. The input information needed is mainly read from JSON configuration files. The source code can be found in the SmartDataLake GitHub repository[38]. Installation and usage of the jar file are described in the next section.

The input format for a Graph Generating Dependencies is composed of mainly two types of JSON files: a set of JSON files with information on the GGDs and a configuration file used by the component to load the set of GGDs. Each Graph Generating Dependency is defined in one JSON file in the following format:

```
{
     "sourceGP": [{
     "name": "dbpediaURL",
     "vertices":[
     {
     "label": "dbpedia",
     "variable":"z"
     }],
     "edges": []
},
{
     "name": "dbpediaURL",
     "vertices":[
      {
            "label": " yago",
            "variable": "x"
      }],
     "edges":[]
}],
     "sourceCons": [{
     "distance": "edit",
     "var1": "x",
     "var2": "z",
     "attr1": "name",
     "attr2": "name",
     "threshold": 1,
     "operator": "<="
}],
"targetGP": [{
     "name": "dbpediaURL",
     "vertices":[
     {
     "label": "dbpedia",
     "variable":"z"
     },{
            "label": " yago",
            "variable": "x"
      }],
     "edges": [{
```

[38] https://github.com/smartdatalake/gcore-spark-ggd

```
      "label": "sameAs",
      "variable": "y",
      "fromVariable": "x",
      "toVariable": "z"
     }]
}],
"targetCons": []
}
```

A graph generating dependency, as introduced earlier, is composed of a source graph pattern and source constraints and a target graph pattern and constraints. In entity resolution, the source graph pattern and constraints are treated as a condition to link matching entities in the target graph pattern. In this context, the JSON file for a GGD has mainly four properties: sourceGP, sourceCons, targetGP, targetCons which refer respectively to the source graph pattern, source constraints, target graph pattern and target constraints. The sourceGP is a list of graph patterns in which each graph pattern is defined by:

```
[{
     "name": "dbpediaURL",
     "vertices":[
     {
     "label": "dbpedia",
     "variable":"z"
     }],
     "edges": [{
      "label": "sameAs",
      "variable": "y",
      "fromVariable": "x",
      "toVariable": "z"
     }]
}
```

In the above, name is the name of the graph in the database, vertices is the list of vertices in the graph pattern and the edges is the list of edges in the graph pattern. The parameter targetGP is a single graph pattern that contains which links should be added in the target graph. Some of the restrictions on the graph patterns are: (1) the variable names should be unique to each vertex/node unless it always refers to the same entity (vertex/edge) and (2) disconnected vertices/edges should be declared as a second graph pattern in the list.

The sourceCons and the targetCons are lists of constraints in which each constraint should be declared in the following format:

```
 [{
     "distance": "edit",
     "var1": "x",
     "var2": "z",
     "attr1": "name",
     "attr2": "name",
     "threshold": 1,
     "operator": "<="
}]
```

This format translates to the following constraint: $\delta_{edit}(x.name, z.name) \leq 1$, which means that the similarity according to the edit distance between the name attribute of the variable x (referring to a vertex/edge in the graph pattern) and the name attribute of the variable z is not greater than 1. The currently implemented (dis)similarity measures in the component and the keywords to be used in the JSON files are:

- Edit distance - "edit";
- Euclidean distance – "euclidean";
- Differential distance – "diff"

The operator can be one of the following symbols: "<", ">", "<=", "=>", "=" and "!=".

In case the constraint is of the type $x = y$ (when it means that variable x refers to the same entity as y) the constraint should be declared by using the "equal" keyword in the following format:

```
 [{
     "distance": "equal",
     "var1": "x",
     "var2": "z",
     "attr1": "",
     "attr2": "",
     "threshold": 0,
     "operator": "="
}]
```

The second JSON is the GGDs configuration file. The configuration file is a file responsible for giving information to the ER component on which GGDs you want to apply. The JSON should be given in the following format:

```
{
      "path": "/home/user/Documents/ggds/"
      "names": ["ggd1", "ggd2", "ggd3"]
}
```

The path field should contain the path to the folder where the GGDs files are stored locally, while the names is the list of the names of the GGD files. The GGDs configuration JSON file is the file that should be the input for GGDs in the component, according to the attributes in the configuration json the component loads automatically the defined GGDs in the specified path.

The graph input format follows the G-Core interpreter input format. Given the graph schema in Figure 41, the input format for the graph should have the structure shown in Figure 42.

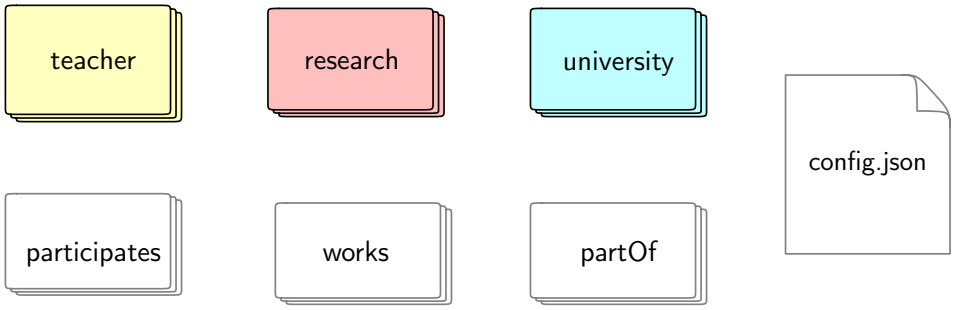


Figure 42: Graph input format.

Each vertex and edge label should have a folder in which each folder contains one or more JSON files with information on the vertices/edges of that specific label. For edges, besides the attributes for each edge it should also contain information on the source and target vertices by setting the attributes *fromId* and *toId*. For example, an edge instance of the label *works* should have the following JSON format:

```
{
        "id": 1,
        "since": "May 2019"
        "fromId": 101
        "toId": 102
}
```

In the above, *fromId* and *toId* refer to the source and target ids of the vertices it links. The config.json file is the configuration file of a graph and should declare information on the graph schema and also where the graph data is stored. The config.json is of the following format:

```
{
  "graphName": "research_graph",
  "graphRootDir": "defaultDB/research_graph",
  "vertexLabels": ["teacher","research","university"],
  "edgeLabels": ["participates", "works", "partOf"],
  "pathLabels": [  ],
  "edgeRestrictions": [
    {"connLabel": "participates",
      "sourceLabel": "teacher",
      "destinationLabel": "research"
    },{
      "connLabel": "works",
      "sourceLabel": "teacher",
      "destinationLabel": "university"
    },{
      "connLabel": "partOf",
      "sourceLabel": "research",
      "destinationLabel": "university"}],
  "pathRestrictions": []
}
```

The field *graphName* is the graph name that it should be saved and referred to in the dataset, *graphRootDir* is the path of the graph folder (folder which contains the config.json file and the vertices/edges folders), *vertexLabels* and *edgeLabels* are the labels present in the graph (name of the folders) and *edgeRestrictions* gives information on the schema of the graph and which edge labels connect which vertices labels. With G-Core there is also the possibility of storing graph paths, however, graph paths are not currently being used for the ER component presented in this deliverable. For more information on how to store paths on G-Core refer to the G-Core language paper and the original G-Core interpreter project.

Graphs stored in the *defaultDB* folder of the project are by default loaded into the component and can be used. It is also possible to load an external graph file in the project (see next Section). In case the user wants to save a graph that was loaded in the project database the output format of the graph is the same as the input format.

4.3. Installation and usage

sHINER is implemented in Scala using the Apache Spark framework. All the code is under a single Maven project and has been built and tested using Java v.1.8, Scala 2.2 and Spark 2.4.

The project uses the following extra libraries/frameworks:

- Apache Spark framework[39] – Used for querying and processing graph data
- FasterXML Jackson project[40] – JSON library for Java, in this case we are using the Scala module of Jackson.
- JSON4s[41] – This library works together with Jackson project to handle JSON reading and writing in the project.
- Spoofax[42] – This library is used for parsing G-Core queries according to the declared grammar.
- Simple Logging Facade for Java (SLF4J)[43] – Handles logging of the project.
- Apache Commons[44]

In order to use the component, first download the project from the SmartDataLake GitHub repository. As mentioned before, this project was built as a Maven project so it is necessary to have Maven installed in your computer. After downloading the project, go to the root folder of the project in the terminal and build the project by typing the following commands:

```
mvn clean
mvn install -DSkipTests
```

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

[40] https://github.com/FasterXML/jackson

[41] https://github.com/json4s/json4s

[42] http://www.metaborg.org/en/latest/

[43] http://www.slf4j.org/

[44] https://commons.apache.org/

The -DSkipTests command avoids running tests when generating the executable jar file. The install command generates a jar file named gcore-interpreter-ggd-1.0-SNAPSHOT-jar-with-dependencies.jar in the target folder of the project. As the name suggests, this is a jar file that also contains all the library dependencies of the project. To run the jar file, type the following command in the terminal:

```
java -jar target/gcore-interpreter-ggd-1.0-SNAPSHOT-jar-with-dependencies.jar
```

To submit this jar to a Spark cluster it follows the same instructions as G-Core interpreter but with a different main class. The following command submits the jar to a spark cluster:

```
spark-submit \
    --class ggd.ERRunner \
    --master local[2] \
    --conf "spark.driver.extraClassPath=/path_to/guice-4.0.jar" \
    target/gcore-interpreter-ggd-1.0-SNAPSHOT-jar-with-dependencies.jar
```

Similar to G-Core, Spoofax uses Guice (https://github.com/google/guice) as a dependency injection framework. In order to submit this to a Spark cluster you should download the guice-4.0.jar from the provided link and pass its location as a *spark.driver.extraClassPath* otherwise the driver does not find it and it is not possible to run the jar file.

If the execution of the jar file is successful, you will see the command-line application ER-Runner shown in Figure 43.

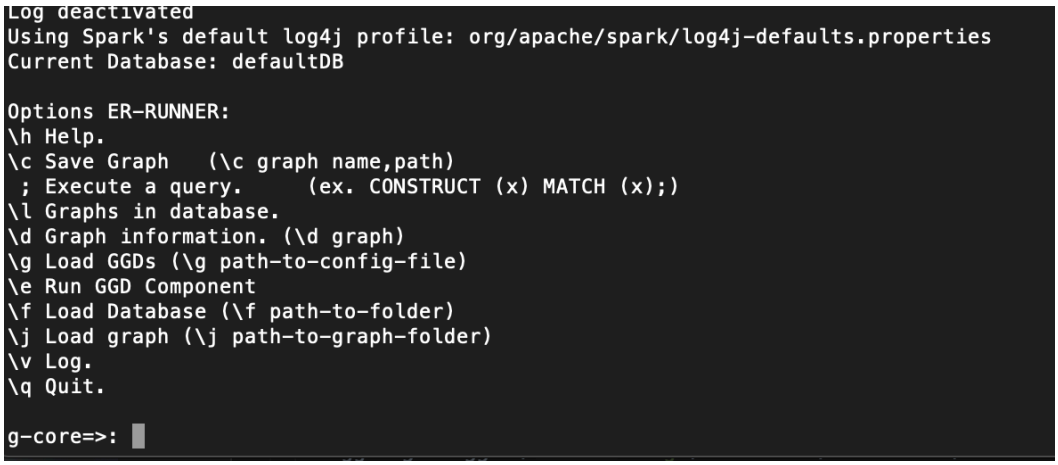


Figure 43: Command line GGD component.

We will now go through each option in this panel.

- \f Load Database and \j Load graph - As mentioned before, the default folder from which the application loads graphs is the project defaultDB folder. However, there are other ways to load a new graph into the running application by using the options \f and \j. With the

Load Database option you load multiple graphs into the application by specifying the root folder in which all these graphs are stored. With the Load Graph option you can load one single graph by specifying the root folder in which this graph is stored. Both options will identify and load automatically information from the config.json files.

- \l Graphs in database and \d Graph information – Both functions will give information about which graphs are loaded in the application. \l option will show the names of the available graphs while the option \d graph-name will show information about the schema of the graph with the name passed as parameter.
- \c Save Graph – This option will save the specified graph in the specified path. The saving format of this graph is the same as the input format. This is handy in order to reuse the saved graph with the application.
- ; Execute query – Query the available graphs in database by using G-Core queries. The result will appear in the terminal.
- \g Load GGDs – Load the GGDs by inputting the config file path. This function will load GGDs that are later used to run the ER Component.
- \e Run GGD Component – As the name suggests, this is the main function to run the ER Component (GGD Component). Through this function, it will trigger the Validation and Graph Generation processes of the GGDs. First, it will run the Validation algorithm to check if there are any GGD being violated in the data and, if yes, which instances violate the data. Then, if a GGD is violated, the Graph Generation function runs in order to repair the GGDs by generating new nodes or edges (in the case of Entity Resolution mainly "same_As" edges). This process is repeated until there are no violated GGDs. The resulting graph with generating edges is the same graph as specified in the target GGDs constraints. The resulting graph can be stored using the SaveGraph option.
- \q – Stop running the application.

4.4. Experimental evaluation

Next, we present our initial experimental results using GGDs. In this initial version of the entity resolution component, we focused on experiments that showcased the execution time and number of matches in relation to the similarity threshold. The goal of these experiments is to describe the general behaviour of the current implementation and identify the main drawbacks that should be addressed in the future.

For these experiments, we used benchmark datasets for binary entity resolution[45] used in [20], specifically the Abt-Buy and the Amazon-Google datasets. The distributed datasets are in a single table CSV format. Since our component requires a graph input, we transformed the tables into

[45] https://dbs.uni-leipzig.de/research/projects/object_matching/benchmark_datasets_for_entity_resolution

property graphs. Figure 44, Figure 45 and Table 8 give more details on the used datasets and the property graph schema.

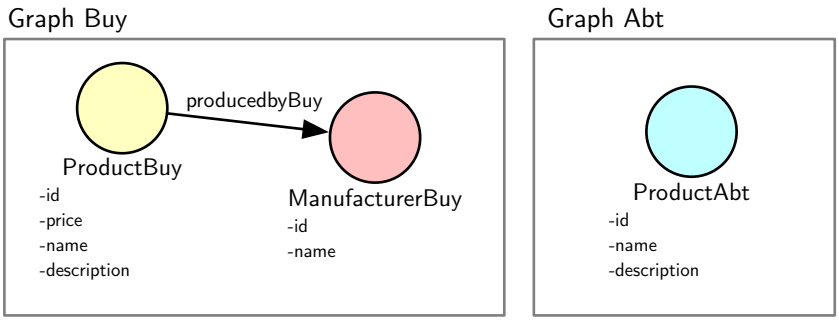


Figure 44: Abt-Buy Graph Schema.

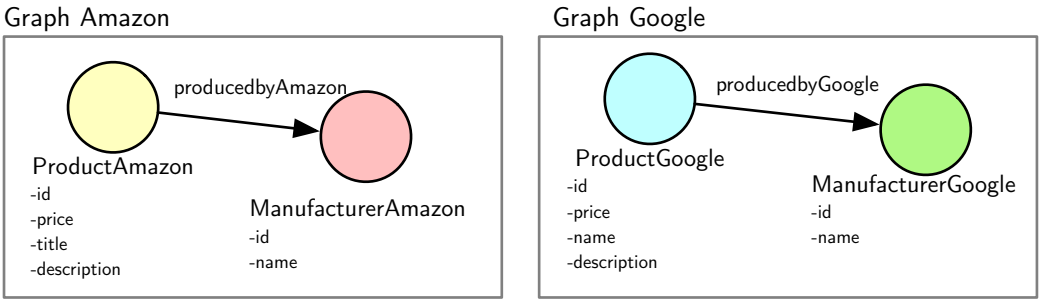


Figure 45: Amazon-Google Products Graph Schema.

Table 8: Number of elements in the property graph.

Label	Number of Elements
ProductBuy	1092
ManufacturerBuy	116
ProductAbt	1081
ProductAmazon	1363
ManufacturerAmazon	350
ProductGoogle	3226
ManufacturerGoogle	67

Given these datasets, we performed three different groups of experiments to measure the execution time of the GGD repair process (including validation and graph generation phases): (1) Execution time according to the number of input GGDs in a graph without source differential constraints, (2) Execution time according to the number of GGDs applied in a graph with source differential constraints, and (3) Execution time and the number of satisfied matches according to the threshold in the source differential constraints. In the next subsections, we explain with more details the experiments and their results.

4.4.1. Execution time according to number of GGDs

In the first group of experiments, we were interested in measuring the execution time of the application of the GGDs (Figure 39) according to the number of input GGDs. Given the small size and the simple schema of the entity resolution datasets, we iteratively applied 1 to 5 GGDs to the Abt-Buy Graph and 1 to 6 GGDs for the Amazon-Google. In both datasets, all of the applied GGDs had no target differential constraints and each target graph pattern enforced the generation of a single new type(label) of vertices/edges. All the source graph patterns of the GGDs were composed of at least one vertex of each graph source in each dataset. As for the source constraints, we considered two scenarios:

(1) GGDs with no source differential constraints: in this case, a new node/edge was generated for every match of the source graph patterns;

(2) GGDs with source differential constraints: in this case, a new node/edge will be generated for every match of source graph patterns that satisfies the source differential constraints.

For the source differential constraints, we considered string attributes and used the edit distance to evaluate. Given the small number of attributes in the dataset, each applied GGD is composed of one source differential constraint. In the following, we present the results on the execution time according to the increasing number of applied GGDs.

Figure 46 shows the results for both Abt-Buy and Amazon-Google datasets in the first scenario and Figure 47 shows the results for the second scenario. In dataset Abt-Buy, we applied 5 GGDs while in the Amazon-Google dataset we applied 6 GGDs. As expected, as the number of GGDs increases, the execution time also increases. The input GGDs is also available in our GitHub repository as an example.

The increase in execution time depends on how many matches of the source graph pattern we need to evaluate. Currently, when searching for multiple graph patterns (i.e. one graph pattern per source and 2 different graph sources) in a single query the final result is represented by a cartesian product of all the matches of each pattern. This leads to an increase in the number of combinations of graph patterns to evaluate.

The drop in time execution for 2 GGDs in Figure 46(a) happens because when graph patterns of GGD1 and GGD2 applied in this case are similar and the search results gets cached after the first time the source graph patterns are queried from G-Core. Observe in Table 9 that the number of matches in Scenario 1 (before checking the source differential constraints) is the same as well.

Table 9 shows the number of satisfied graph patterns in both scenarios according to the GGD applied. This drawback will be optimized in the following versions of the component.

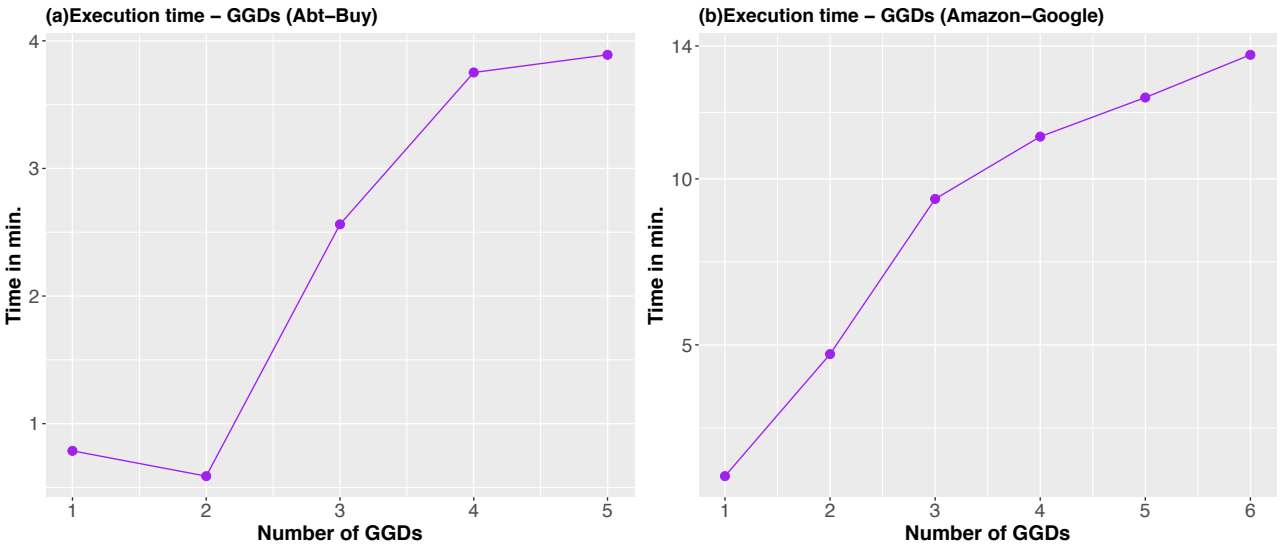


Figure 46: Execution time according to the number of GGDs.

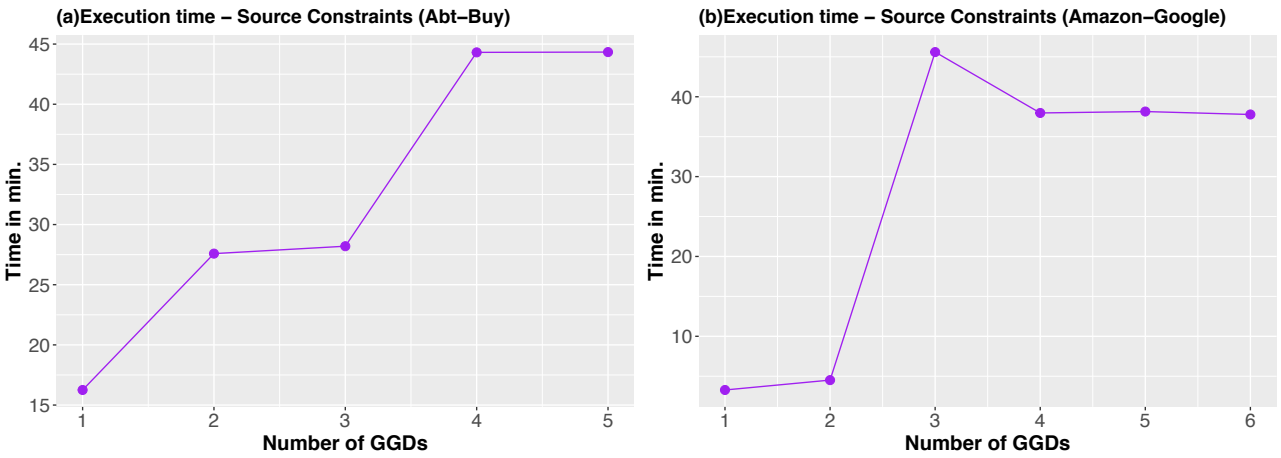


Figure 47: Execution time according to the number of GGDs with Source Differential Constraints.

In the second group of the experiments, we declared source differential constraints in the input GGDs. For the input GGDs in this experiment, we used the same graph patterns for source and target that we used in the first group of experiments and added, in each GGD, one source constraint of the type $\delta_{edit}(x.A, x'.B) \leq t_{A,B}$. For these source differential constraints, we used edit distance to compare one attribute of each different graph sources.

When applying the source differential constraints, the number of satisfied source patterns decreases significantly compared to the first group of experiments. However, the source differential constraints need to be checked result-by-result over the matched source graph patterns. This process is complex and time-demanding and currently cannot be handled directly by G-Core or SparkSQL. We plan to push the evaluation of differential constraints into G-Core in our future work.

Table 9: Number of matched source patterns.

GGD	Number of matched patterns in the source			
	Amazon-Google		Abt-Buy	
	Scenario 1	Scenario 2	Scenario 1	Scenario 2
GGD1	316216	8	1180452	94
GGD2	316216	47236	1180452	255
GGD3	4397038	22903	1173966	735
GGD4	23450	646	116	116
GGD5	1363	1363	1081	1081
GGD6	1363	1363		

4.4.2. Performance according to threshold increase

In the last group of experiments, our goal is to measure the execution time of the GGD repair process according to the threshold increase. Towards this, we applied GGDs with the same source and target graph patterns and a single source constraint of the type $\delta_{edit}(x.A, x'.B) \leq t_{A,B}$ but with increasing threshold $t_{A,B}$. The source graph patterns are composed of one vertex of each graph source and the target graph pattern adds an edge between these two vertices.

In Figure 48(a), we show the execution time according to the threshold increase in the Amazon-Google dataset and in Figure 48(b) the number of graph pattern matches that satisfy the source constraint according to the threshold increase. As expected, as the similarity threshold increases, the number of satisfying matches increases as well. Nevertheless, the execution time does not present big changes, given the current implementation of the source differential constraint checking.

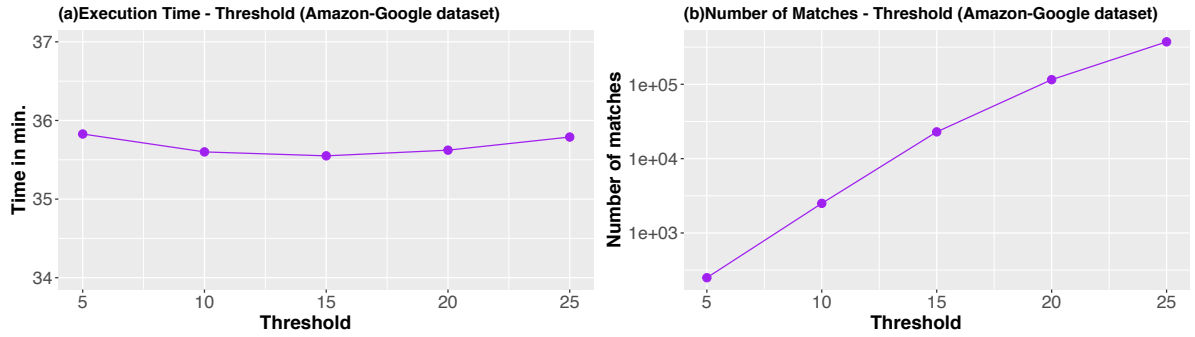


Figure 48: Execution time and number of matches according to threshold.

According to our preliminary experiment results, we conclude that the GGDs can be applied to add vertices/edges in a graph according to the defined conditions in the input GGDs, however, its performance is defined by the graph pattern query and the complexity of checking given source differential constraints.

For this reason, in our next steps, we will focus on optimizing our current implementation. Especially, the graph pattern queries and the differential constraint checking. We will also work on the possibility of integration with the other components in the SmartDataLake project.

5. Conclusions

In this report, we have presented our ongoing work on the SmartDataLake components for entity ranking, similarity search and entity resolution. For entity ranking, we have developed two components: (a) HMiner, which ranks entities represented as nodes in a HIN, and (b) BRS, which ranks geospatial regions represented by rectangular areas of user-defined width and height. For similarity search, we have implemented three components: (a) SimJoin, which is designed for discovering pairs of similar entities represented as sets of elements, (b) SimSearch, which retrieves the most similar entities to a given entity based on textual, numeric and spatial attributes, and (c) SimSearchTS, which provides support for similarity search and join over time series. Finally, for entity resolution, we have developed a component based on Graph Generating Dependencies which, given the conditions for similarity search, can generate new vertices/edges in the HIN in order to represent nodes referring to the same real-world entity. For each one of these components, we have described its functionalities, listed its API and installation instructions, and presented an experimental evaluation. The integration of these components with those for data virtualization (WP2) and for interactive visual analytics (WP4) will be described in Deliverable D1.4: “Initial integrated system”.

References

[1] C. Shi, Y. Li, J. Zhang, Y. Sun, P. Yu. A Survey of Heterogeneous Information Network Analysis. IEEE Trans. Knowl. Data Eng. 29(1): 17-37 (2017)

[2] S. Chatzopoulos, K. Patroumpas, A. Zeakis, T. Vergoulis, D. Skoutas. SPHINX: A system for metapath-based entity exploration in heterogeneous information networks. VLDB Demo paper, 2020 (to appear).

[3] K. Patroumpas, D. Skoutas. Similarity search for enriched geospatial data. In Proc. of the 7th International ACM SIGMOD Workshop on Managing and Mining Enriched Geo-Spatial Data (GeoRich), 2020.

[4] G. Chatzigeorgakidis, D. Skoutas, K. Patroumpas, T. Palpanas, S. Athanasiou, S. Skiadopoulos. Local pair and bundle discovery over co-evolving time series. SSTD 2019, pages 160-169.

[5] L. Shimomura, G. Fletcher, N. Yakovets. GGDs: Graph Generating Dependencies. arXiv, 2020.

[6] R. Motwani L. Page, S. Brin, T. Winograd. 1999. The PageRank Citation Ranking: Bringing Order to the Web. Technical Report. Stanford InfoLab.

[7] C. Shi, Y. Li, P. Yu, B. Wu. Constrained-meta-path-based ranking in heterogeneous information network. Knowl. Inf. Syst. 49(2): 719-747 (2016)

[8] D. Kernert, F. Köhler, W. Lehner. SpMacho - Optimizing Sparse Linear Algebra Expressions with Probabilistic Density Estimation. EDBT 2015: 289-300.

[9] K. Feng, G. Cong, S. Bhowmick, W. Peng, C. Miao. Towards best region search for data exploration. ACM SIGMOD 2016: 1055-1070.

[10] H. Shahrivari, M. Olma, O. Papapetrou, D. Skoutas, A. Ailamaki. A parallel and distributed approach for diversified top-k best region search. EDBT 2020: 265-276.

[11] R. J. Bayardo, Y. Ma, R. Srikant. Scaling up all pairs similarity search. WWW, pages 131–140, 2007.

[12] C. Xiao, W. Wang, X. Lin, H. Shang. Top-k set similarity joins. IEEE ICDE, pages 916–927, 2009.

[13] D. Deng, A. Kim, S. Madden, M. Stonebraker. Silkmoth: An efficient method for finding related sets with maximum matching constraints. PVLDB, 10(10):1082–1093, 2017.

[14] G. Papadakis, D. Skoutas, E. Thanos, T. Palpanas. Blocking and filtering techniques for entity resolution: a survey. ACM Comput. Surv. 53(2) (2020).

[15] I. Ilyas, G. Beskales, M. Soliman. A survey of top-k query processing techniques in relational database systems. ACM Comput. Surv. 40(4): 11:1-11:58 (2008).

[16] R. Fagin, A. Lotem, M. Naor. Optimal aggregation algorithms for middleware. J. Comput. Syst. Sci., 66(4):614–656 (2003).

[17] Y. Wang, L. Wang, Y. Li, D. He, and T. Liu. A theoretical analysis of NDCG type ranking measures. In COLT, volume 30, pages 25–54 (2013).

[18] E. Keogh, K. Chakrabarti, M. Pazzani, S. Mehrotra. Dimensionality reduction for fast similarity search in large time series databases. Knowl. Inf. Syst. 3(3): 263-286 (2001).

[19] R. Angles, M. Arenas, P. Barceló, P. Boncz, G. Fletcher, C. Gutierrez, T. Lindaaker, M. Paradies, S. Plantikow, J. Sequeda, O. van Rest, H. Voigt. G-CORE: A Core for Future Graph Query Languages. SIGMOD Conference 2018: 1421-1432.

[20] H. Köpcke, A. Thor, E. Rahm. Evaluation of entity resolution approaches on real-world match problems. Proc. VLDB Endow., vol. 3, no. 1, pp. 484–493, 2010.