SmartDataLake

DELIVERABLE D3.1

Similarity search, entity resolution and ranking

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 16 (30/04/2020)
Actual Submission Date	27/04/2020
Work Package	WP3: Heterogeneous Information Network Mining
Tasks	Task 3.1: Similarity search and exploration & Task 3.2: Entity resolution and ranking
Type	Report
Lead Beneficiary	ATHENA RC
Approval Status	Submitted for approval
Version	1.0
Number of Pages	58
Filename	SmartDataLake-D3.1-Similarity_search_entity_resolution_and_ranking.pdf

Abstract

This report presents our approach for entity exploration in Heterogeneous Information Networks (HINs). Specifically, we address the problems of entity ranking, similarity search, and entity resolution. We first explain the use of metapaths to define different views of a HIN and derive relative rankings of entities with respect to relationships of different semantics. Then, we present our work on set similarity joins, similarity search over entities with heterogeneous attributes, and similarity search for time series. Finally, we introduce a novel method for entity resolution based on the concept of graph generating dependencies.

History

Version	Date	Reason	Revised by
0.1	14/01/2020	Table of Contents	Dimitris Skoutas
0.2	03/04/2020	First draft for internal review	Dimitris Skoutas
0.3	22/04/2020	Revised draft	Dimitris Skoutas
1.0	27/04/2020	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
UKON	Thilo Spinner	thilo.spinner@uni-konstanz.de

Executive Summary

Heterogeneous Information Networks (HINs) are graphs comprising different types of nodes (*entities*) and edges (*relationships*). HINs offer an intuitive and generic model for representing complex information in various domains. Hence, they are a convenient and suitable concept for exploring and analysing entities in data lakes. A core concept in HINs is that of *metapath*, which is a path defined on the schema of a HIN. Metapaths represent relationships of different semantics between entities of the same or different type, providing a mechanism for exploring and analysing a HIN from multiple perspectives. Thus, they are fundamental for several types of analyses in HINs. This report presents our approach for entity exploration in HINs, focusing in particular on the following tasks: (a) entity ranking, (b) similarity search and join, and (c) entity resolution.

In Section 1, we introduce the main concepts, provide an overview of our work, and explain its context in the SmartDataLake project.

Entity ranking is discussed in Section 2. We first present how different metapaths can be used to define different views of a HIN, indicating relationships with different semantics between entities. We focus on cyclic metapaths, which essentially transform a HIN to a homogeneous network so that well-known algorithms for node ranking in graphs can be applied to measure the centrality of entities. Consequently, relative entity rankings with respect to one or more metapaths can be derived and combined. Furthermore, we address the problem of ranking spatial regions. This takes into consideration spatial attributes of the entities, which are not represented in the network structure, and thus cannot be captured by the concept of metapaths.

Similarity search and join is addressed in Section 3. We first present the filter-verification framework and an overview of the state of the art, focusing on set similarity joins. These retrieve similar entities based on textual properties, such as categories, keywords or tags, as well as based on their sets of neighbours in the network, which in the context of HINs is relative to the metapaths under consideration. We present our adaptations to existing algorithms, focusing on top-k queries. We also introduce set weights, which allow to include ranking scores of entities in the similarity search process. Moreover, we present our approach for top-k similarity search queries on entities with heterogeneous attributes, which we model as a rank aggregation problem. Finally, we report our work on similarity search for time series, and its extension to geolocated time series.

Entity resolution is presented in Section 4. This task refers to matching entity profiles, within the same or across different data sources, that correspond to the same real-world entity. We propose a novel approach for entity resolution based on a new class of graph dependencies called Graph Generating Dependencies (GGDs). GGDs extend the idea of functional dependencies, conditional functional dependencies and differential dependencies from relational data to graph data. We introduce the syntax and semantics of GGDs and explain how to apply them for entity resolution in HINs.

Finally, Section 5 summarizes and concludes the report.

Abbreviations and Acronyms

CFD	Conditional Functional Dependencies
DAG	Directed Acyclic Graph
DD	Differential Dependencies
ER	Entity Resolution
FD	Functional Dependencies
GDD	Graph Differential Dependencies
GED	Graph Entity Dependencies
GFD	Graph Functional Dependencies
GGD	Graph Generating Dependencies
GPAR	Graph-Pattern Association Rules
GRR	Graph Repairing Rules
HIN	Heterogenous Information Network
MBR	Minimum Bounding Rectangle
MBTS	Minimum Bounding Time Series
RDF	Resource Description Framework
SAX	Symbolic Aggregate Approximation
TGD	Tuple-Generating Dependencies

Table of Contents

1. Introduction

1.1. Overview

Data lakes are raw data ecosystems, where large amounts of structured, unstructured and semi-structured data coexist in its natural format and in various models. Data lakes retain all data, aiming to support ad hoc, self-service analytics, as opposed to predefined parts of data that are known in advance to serve specific purposes. This opens up new opportunities for data scientists to tap into the data lake to analyse data from new sources, combine data of different types, come up with new business questions, test hypotheses, and derive new insights and knowledge. Thus, it can significantly increase the flexibility and speed of data-driven decision making.

One of the main challenges is how to efficiently and effectively sift through the vast and heterogeneous contents of the data lake. A data lake contains diverse information about various types of *entities* (e.g., companies, products, customers) and various types of *relationships* between them. Moreover, different facets and pieces of information for the same entity may be fragmented across different data sources (e.g., enterprise data, web, social networks, news, stock market).

Since the entities and relationships in real-world data often logically form a graph, Heterogeneous Information Networks (HINs) provide an intuitive way to represent this information. 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).

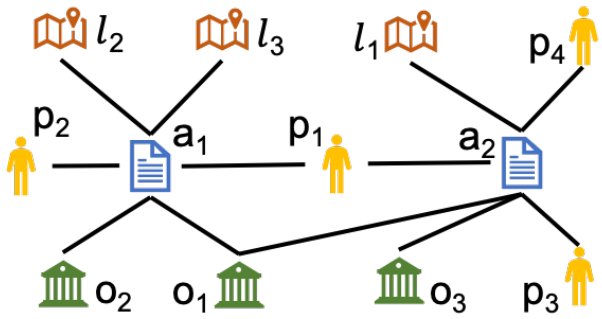


Figure 1: Illustrative example of a HIN containing news articles (A), persons (P), organizations (O), and locations (L).

A core concept for analysing HINs is that of *metapath*, which is a path defined on the schema of the HIN [2]. Metapaths represent relationships of different semantics between entities of the same or different type, providing a mechanism for exploring and analysing a HIN from multiple perspectives. In the example of Figure 1, we can consider, for instance, the metapath PAP, which

connects two persons (P) if they are mentioned in the same article (A), or the metapaths $PAPAP$, $PAOAP$ and $PALAP$, which connect two persons if they are mentioned in articles that also mention a common person, organization (O), or location (L), respectively. Then, we can observe that persons p_2 and p_3 are connected according to $PAPAP$ (specifically, via $p_2 - a_1 - p_1 - a_2 - p_3$) and $PAOAP$ (specifically, via $p_2 - a_1 - o_1 - a_2 - p_3$), but not according to PAP or $PALAP$.

Hence, the neighbourhood or, accordingly, the centrality of a node in a HIN become *relative* to the metapaths under consideration. This raises the need for methods and tools that can facilitate users in defining and computing different *views* of a HIN based on different combinations of metapaths. This can be used to compute and compare answers to various questions, such as, which entities are the most important (central) in the network or having the highest similarity (common or similar neighbours) to a query entity. The task becomes even more complex in the presence of entity types that are additionally associated with *spatial* or *temporal* properties (e.g., geospatial coordinates or timestamps). Spatial and temporal proximity are important factors in several analyses. Yet, since spatial and temporal relationships are typically not represented explicitly in the network structure, they cannot be captured by metapaths.

This report presents our approach and methods for addressing the following fundamental tasks for entity exploration in HINs.

Entity ranking. Entity ranking (Section 2) allows identifying the top entities in the network, where centrality is used as a measure of importance. Given that a data lake may contain information about thousands or millions of entities, ranking is important for providing initial insights and seeds for further exploration, thus guiding users in prioritizing their analysis. Ranking of entities in HINs needs to take into consideration the concept of metapaths, since the centrality of a node depends on the metapaths being considered. Thus, each entity has a different ranking score for each (combination of) metapath(s). Moreover, the top entities in the network may change over time. Finally, a different approach is required for ranking spatial entities, since spatial proximity is not typically represented by links in the network.

Similarity search. Similarity search and join (Section 3) aim at discovering the most similar entities to a query entity or all pairs of similar entities within an entity collection. For instance, starting from the top entities in the network, similarity search can be used to explore and navigate through the web of entities progressively. Moreover, similarity joins can be used to construct a similarity graph of entities, which can then provide the basis for further analysis tasks, such as link prediction or community detection. Entities in HINs are connected via different types of paths and are associated with different types of attributes, e.g., numeric, spatial, or temporal, which need to be considered during search. Thus, the similarity between entities is also relative to the respective metapaths and the preferences among different attributes.

Entity resolution. In a data lake, information about an entity may be fragmented across several sources. Entity resolution (Section 4) aims at matching entities, within the same or across different data sources, that refer to the same real-world entity. These instances need to be identified and linked so that duplicates can be removed or merged. This is a crucial step for data cleaning and integration. In the context of HINs, our approach focuses on leveraging the network structure and the properties of different entity types to perform entity matching.

1.2. Role in the project

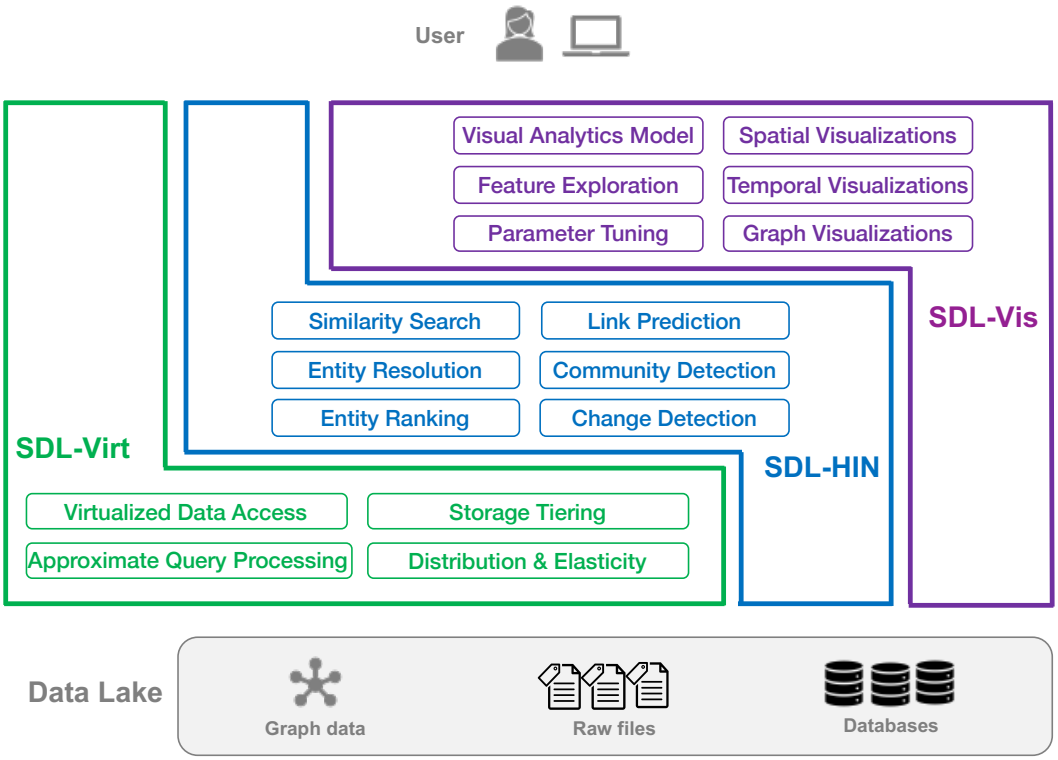


Figure 2: The three layers of SDL: SDL-Virt, SDL-HIN, and SDL-Vis. This deliverable covers SDL-HIN, and in particular Entity Ranking, Entity Resolution and Similarity Search.

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 Deliverable D1.2: "System architecture"). These parts, emphasizing on *exploration*, have been our main focus during the first period of the project, as they also lay the foundation for the rest of the components of SDL-HIN, i.e., link prediction, community detection and change detection, emphasizing on *mining*, which will be the main focus during the second period.

The components of SDL-HIN can ingest data directly, or they can leverage the data virtualization capabilities offered by the underlying layer (SDL-Virt) to query data in situ, thus hiding the complexities of dealing with data of different shapes and formats. Moreover, they interact with the SDL-Vis layer to provide the functionalities for interactive visual analytics. Through SDL-Vis, the data analyst can tune parameters and steer the execution of SDL-HIN components, e.g., by setting weight parameters or similarity thresholds.

This deliverable reports the work conducted so far in Tasks 3.1: "Similarity search and exploration" and 3.2: "Entity resolution and ranking". It covers the main concepts involved, the problems that are addressed, and the proposed approaches and methods. The respective software components that are being developed will be presented in Deliverable D3.2: "Initial version of the HIN mining engine". This work will continue throughout Tasks 3.1 and 3.2, and the final results will be presented in Deliverable D3.5: "Final version of the HIN mining engine".

2. Entity Ranking

2.1. Introduction

In the context of ranking entities, which are nodes in a homogeneous network, PageRank [52] [53] is the most widely known algorithm. It is a graph processing algorithm that measures the transitive influence or connectivity of nodes. It was originally introduced to estimate the importance of a Web page by counting the number and the quality of the links pointing to it. The underlying assumption is that Web pages of high importance are more likely to receive a high volume of incoming links from other Web pages. PageRank scores can be computed by either iteratively distributing one node's rank (initially based on its degree) over its neighbours, or by randomly traversing the graph and counting the frequency of hitting each node during these walks.

PageRank simulates the behaviour of a "random surfer" that traverses the network starting from a randomly selected node. Then, it either picks one node to visit from those being directly connected to the current node, or it chooses any other node in the network at random. The PageRank score of a node i indicates the probability of a random surfer visiting it and satisfies:

$$s_i = \sum_j P_{i,j} s_j + (1 - a) v_i$$

P is the network's transition matrix, $P_{i,j} = A_{i,j} / k_j^{out}$, and k_j^{out} is the out-degree of node j; (1-a) is the random jump probability, controlling how often the surfer chooses to visit a random node; and v_i is the landing probability of reaching node i after a random jump. Typically, each node is given a uniform landing probability of $v_i = 1 / N$, where N is the number of nodes in the network. It should be noted that in the case of "dangling nodes", which contain no outgoing edges, i.e., nodes j with out-degree $k_j^{out} = 0$, the value of the transition matrix is undefined. To address this, the standard technique is to set $P_{i,j} = 1 / N$, or to some other landing probability, whenever $k_j^{out} = 0$.

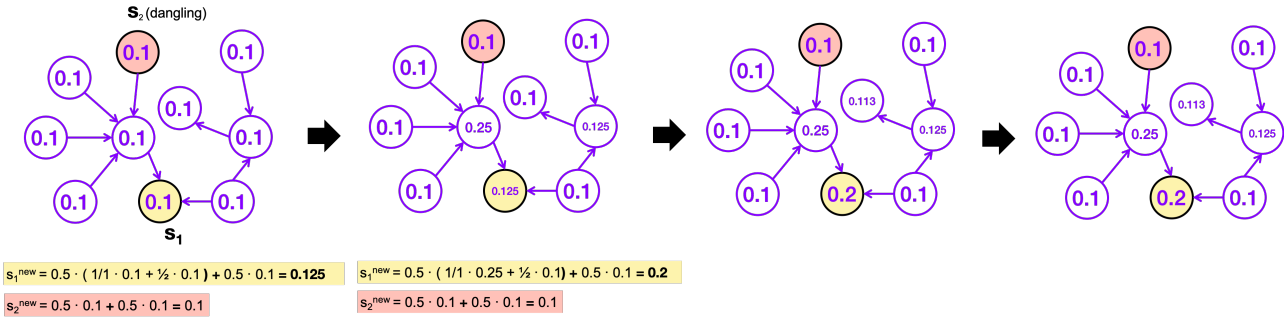


Figure 3: PageRank scores for a toy homogeneous network.

Figure 3 presents an example showing the computation of PageRank scores for a homogeneous network consisting of 10 nodes.

PageRank has a multitude of variants. The most popular of them is Personalised PageRank [54]. The characteristic of Personalised PageRank is that it is biased towards a set of particular nodes, meaning that during the random jump, it is not equally likely to move to any other node. Instead, some nodes have larger random jump probabilities than others. This is useful for many applications, such as recommender systems.

However, node ranking algorithms for homogeneous networks cannot be used without adaptation to rank nodes in HINs. This is because the different types of nodes and edges in the HIN have diverging interpretations, and the algorithms for homogeneous networks are unable to take advantage of such information. As a result, ranking the nodes of a HIN using PageRank or Personalised PageRank may result in a ranked list of diverging entities without clear semantics. Thus, it is evident that node ranking algorithms tailored to HINs should be used in this case.

2.2. Entity ranking in HINs

Before ranking entities in HINs, the user has to determine the criteria based on which they should be ranked. The user preferences can be expressed by exploiting the notion of *metapaths* (see also Section 1.1). In particular, the user provides one or more (cyclic) metapaths of the HIN that relate the entities that need to be ranked to each other based on the desired semantics. Each provided metapath essentially defines a *view* of the HIN, which is a homogeneous network that connects the entities based on the particular type of paths in the original network. Thus, the most important entities based on a particular metapath can then be easily revealed by executing a traditional entity ranking algorithm like PageRank or Personalised PageRank (see Section 2.1) on the corresponding view of the HIN. The previously described approach is the one followed by the HRank method [55].

Producing the view of the HIN based on a particular metapath is a demanding, resource-consuming computational task. Recalling that a metapath can be defined as a sequence of different edge types of the HIN, the input of the computational task to produce a metapath-based view of the HIN consists of all the edges of the node types that are involved in the particular metapath. The edges of each type can be represented by a corresponding adjacency matrix. The same holds for the edges of the output view. Thus, the task to produce the metapath-based view of the HIN can get as input the adjacency matrices of all different edge types involved in the metapath and provide as output the adjacency matrix that represents the output view. In that case, the processing corresponds to applying the matrix multiplication operation to the input adjacency matrices.

The order of the matrix multiplications involved in the previous process can significantly affect its memory footprint and execution time. Exploiting this observation, a dynamic programming-based approach to select an efficient order for multiplications has been proposed in the literature [55]. It estimates the cost of each matrix multiplication operation to be performed, and uses dynamic

programming to select the multiplication order that minimizes the estimated cost. Our implementation builds upon this approach, extending it to achieve significant further speedups. In particular, since most of the involved matrices are very sparse, we replace traditional matrix representations with other data structures that are more suitable to represent sparse matrices. This enables us to exploit an efficient matrix multiplication operation tailored to sparse matrices. In addition, we replace the initial multiplication cost estimator with one that is more suitable to estimate the cost based on sparse matrices [56].

In particular, given a matrix A with dimensions (m x n) and an (n x l) matrix B the exact cost of multiplying A with B can be exactly determined in the case of dense matrices and is equal to m·n·l. However, for sparse matrices, the multiplication cost is highly dependent on the density of the matrices being multiplied. The density of a sparse matrix A [m x n] is defined as:

$$\rho_A = \frac{nonZero(A)}{m \cdot n}$$

Assuming that non-zero elements are uniformly distributed in the multiplied matrices, the density of the result according to the average case estimator is given by:

$$\rho_C = 1 - (1 - \rho_A \cdot \rho_B)^n$$

Where ρ_A and ρ_B are the densities of the multiplied matrices A and B respectively. Note that the average case estimator is a best-effort estimator as it is based only on matrices metadata with no additional overhead.

Based on the above, we adopt the following formula [56] to estimate the cost of sparse matrix multiplication:

$$Cost_{sparse}(A, B) = a \cdot nonZero(A) + \beta \cdot N_X + \gamma \cdot nonZero(C)$$

where

- $nonZero(A) = m \cdot n \cdot \rho_A$,
- N_X is the estimated number of operations during the multiplication and is equal to $m \cdot n \cdot \rho_A \cdot l \cdot \rho_B$
- $nonZero(C)$ is the estimated number of non-zero elements in the result and equal to $m \cdot n \cdot \rho_C$.

The constants α, β, γ abstract several other parameters of the sparse matrix multiplication such as the algorithm used for the multiplication, the memory bandwidth, the pattern of non-zero elements of the input and result matrices. The values of these constants can be determined by multilinear regression using means square fit. This sparsity-aware cost estimation can then be adapted to the dynamic programming approach to estimate the order of multiplications with the minimum cost. We only need to store at each step the densities of the optimal sub-chains.

It should be noted that, while determining the desired metapaths to be considered for entity ranking, the user can also provide a set of hard constraints on them. These constraints involve the values of attributes of the entity types that are contained in the particular metapath. By determining a constraint, the user determines that she does not want all the paths of the particular type to be considered, only those paths that satisfy the given condition. This feature can be easily

supported by the previously described matrix multiplication-based approach since the constrained can be applied just by multiplying the corresponding adjacency matrix with a diagonal matrix that has 1 values in the diagonal only for the elements that satisfy specified constraints.

As mentioned, after producing the metapath-based HIN view, the desired entity ranking can be calculated by executing a traditional entity ranking algorithm for homogeneous networks on the view. In some cases, it is desired to consider multiple metapaths simultaneously for ranking. This can be handled as follows: the ranked lists for each of the metapaths is computed by following the previously described approach; then, a rank aggregation algorithm [51] is used to provide the final ranked list.

2.3. Ranking of spatial regions

2.3.1. Preliminaries

So far, we have considered the problem of ranking entities based on their position in the network. In the following, we turn our focus to entities that are associated with geolocations (e.g., companies having offices or points of sales at certain locations). The spatial dimension plays a crucial role in many application areas (e.g., in geo-marketing, logistics, or urban planning), as characteristically stated by Tobler's first law of geography, according to which "everything is related to everything else, but near things are more related than distant things" [3]. Specifically, we study the problem of Best Region Search (BRS), which refers to ranking spatial regions based on their contents. This can be used, for example, to find those areas in a given city or country that contain the largest number of companies of a certain type or generate the largest amount of revenue. Formally, the BRS problem [4] is defined as follows.

Definition 1 (Best Region Search). Given a set of spatial points P, a monotone scoring function f, and the width w and height h of an axis-aligned rectangle R, find the best placement of R such that the value of f over the points enclosed by R is maximized.

The state-of-the-art algorithm for this problem [4] discovers only a single result, i.e., a single position for R that achieves the highest possible score for f. However, this is not sufficient for data exploration, where the user typically needs to examine and analyse several alternatives. This shortcoming has been addressed by an algorithm that progressively retrieves top-k results [5]. Moreover, this algorithm allows diversifying these results by skipping or penalizing results that have an overlap with already retrieved ones. An overview of the process is illustrated in Figure 4. The algorithm first divides the input area into a uniform grid of cells with dimensions $w \times h$ and computes an upper bound for the score of each cell. Then, the cells are processed in decreasing order of their upper bounds, with intermediate results being maintained in a priority queue. Processing each cell involves a series of scans that result in smaller regions, each one also associated with an upper bound. The algorithm refines these intermediate results progressively until eventually the top-k regions have been discovered. During the process, the degree of overlap

of each next candidate to the already selected results is computed before determining the final score of each region.

Although this algorithm provides greater flexibility for data exploration, its centralized nature limits its scalability. To address this drawback, in the following, we describe our parallel and distributed solutions to this problem.

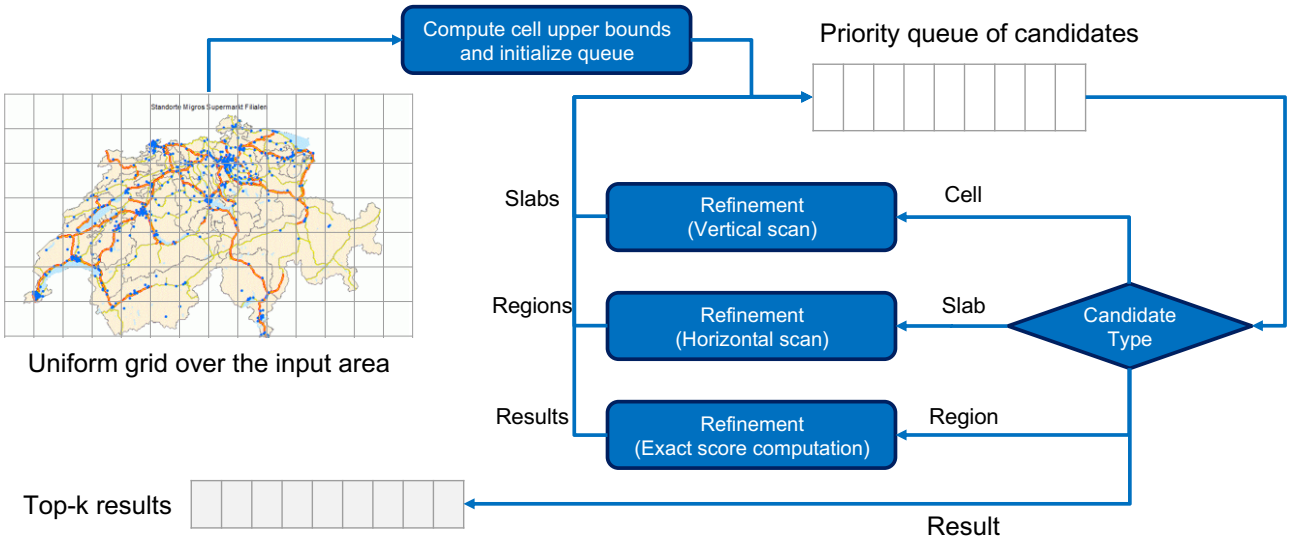


Figure 4: Overview of the centralized algorithm for Best Region Search.

2.3.2. Our approach

In the following, we present our approach towards a parallel and distributed solution to the diversified top-k BRS problem.

A straightforward idea for scaling out is to spatially segment the input area into several partitions, then process each partition in parallel, using the aforementioned centralized algorithm to compute top-k results in each partition, and then merge these local top-k results at the coordinator to obtain the final, global top-k results. However, as explained next, this method cannot guarantee that the global top-k results are the correct ones when the diversification criterion needs to be taken into account. Diversification of the results is an important requirement, since returning regions that overlap each other to the user offers no new insights for exploration.

Consider the example illustrated in Figure 5, comprising an input area segmented in two spatial partitions N_i and N_j, assigned to respective workers for processing in parallel. Assume that worker N_i computes the following local top-k results: $R_2 > R_3 > R_5 > R_8$, whereas worker N_j computes $R_1 > R_4 > R_6 > R_7$ (lower indices indicate higher score, i.e., $R_i > R_j$ if $i < j$). If $k = 3$, then N_i will return the following local top-3 results to the coordinator: R_2, R_5, R_8. Notice that R_3 is skipped since it overlaps with R_2, which has a higher score and is thus selected first. Similarly, N_j will return the following top-3 results: R_1, R_4, R_7. Again, R_6 is skipped because it overlaps with R_4, which has a

higher score. Given these two sets of local top-3 results, the coordinator will merge them to obtain the following global top-3 results: R_1, R_4, R_5. Notice that R_2 is skipped since it overlaps with R_1, which has a higher score and is thus selected first. However, the correct global top-3 list should have been the following: R_1, R_3, R_6 since R_3 is better than R_4, and thus it should have been selected as the second top result. This error was caused by the fact that R_3 was omitted locally due to its overlap with R_2; however, R_2 was eventually skipped by the coordinator due to its overlap with R_1. Hence, workers cannot safely discard overlapping results because those causing the overlap may in turn overlap with other neighbouring partitions. This creates a chain of dependencies between two or more partitions, complicating the decision which of the local results to select.

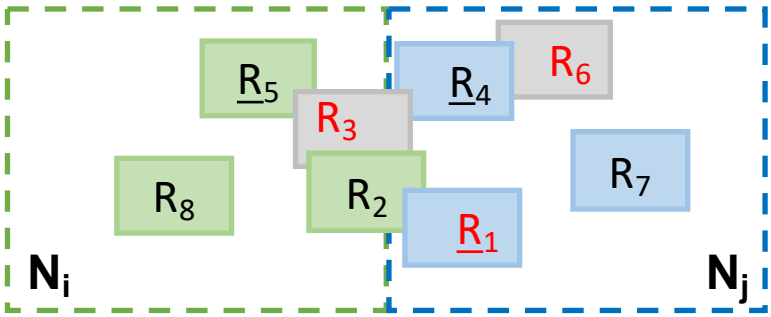


Figure 5: Example showing how region overlaps prevent assembling the correct global top-k results from the local ones. Green and blue regions are those returned by workers N_i and N_j, respectively. Underlined results are those finally selected by the coordinator. Red ones are the correct top-3 results.

Next, we briefly present our approach to addressing the above challenge (see Figure 6). We outline the basic ideas of our solution; a detailed description is provided in [6].

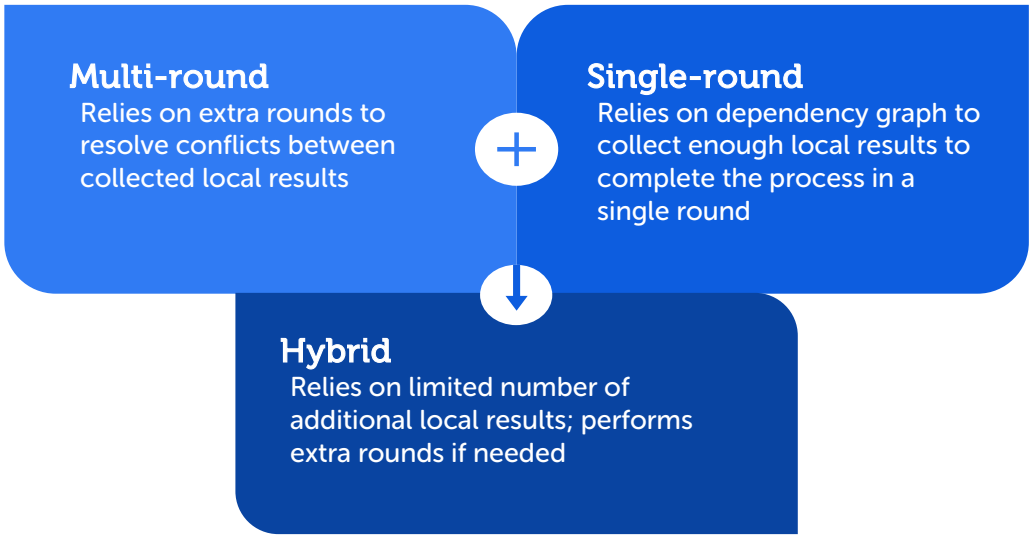


Figure 6: Proposed algorithms for diversified top-k best region search.

Multi-round algorithm. Our first method is an incremental, multi-round algorithm that gradually builds the global top-k list of results by retrieving local top-k results from each worker at each round. While aggregating the local top-k results received at the end of each round, the coordinator resolves overlaps. If needed, the coordinator contacts the affected workers again, informing them about the occurred overlaps and asking them for accordingly revised top-k results. This process may take up to k rounds to complete. Hence, in this algorithm, the problem of dealing with conflicts is handled by the coordinator. The advantage of the algorithm is that workers can readily exploit the centralized top-k algorithm for the BRS problem with minor adaptations since the decisions for forming the global top-k list are made at the coordinator level. The downside is that when overlapping results are identified and discarded, a new round has to be executed. The relevant workers have to be informed about these results and requested to compute new results accordingly. These extra rounds introduce additional overhead.

Single-round algorithm. To overcome the drawback of multiple rounds, we devise a single-round algorithm. This requires formulating appropriate conditions to reason about the uncertainty of the validity of the local results. In this algorithm, each worker proactively anticipates which results may be disqualified due to overlaps with regions from other partitions. Assessing each scenario, it produces a sufficiently extended local set of results, guaranteeing that the process can be completed within a single round of communication. However, the computation of these extended results imposes additional overhead on each worker, implying higher execution time of the local processes.

The single-round algorithm maintains auxiliary information per region that is used during the reduction phase for handing overlapping regions from different partitions. The intuition is the following: when processing each partition, a sufficient number of regions (typically larger than k) is computed and sent to the coordinator. This guarantees that the coordinator has all candidates needed to assemble the global top-k results without further communication to the workers. The challenge is to determine how many, and which, additional regions need to be sent, such that the coordinator is guaranteed to hold sufficient information for extracting the final answer. As shown above, the challenge arises from the presence of overlaps between candidate regions that are detected in different partitions.

In the following, we demonstrate this approach by using the example illustrated in Figure 7. Region r_1 of partition $P_{2,2}$ overlaps with the green region of partition $P_{2,1}$, which has a higher score. Therefore, r_1 will not be in the final results, unless the green region is eventually disqualified. This makes r_2 a possible result. Unfortunately, the existence of long chains of such overlaps prohibits local solutions that rely on data replication. The single-round algorithm addresses this issue by forming a *dependency graph* of the identified candidate regions at each partition. These graphs allow each node to establish a lower bound on the number of regions contained in the local results that are accepted by the coordinator if their score is sufficiently high.

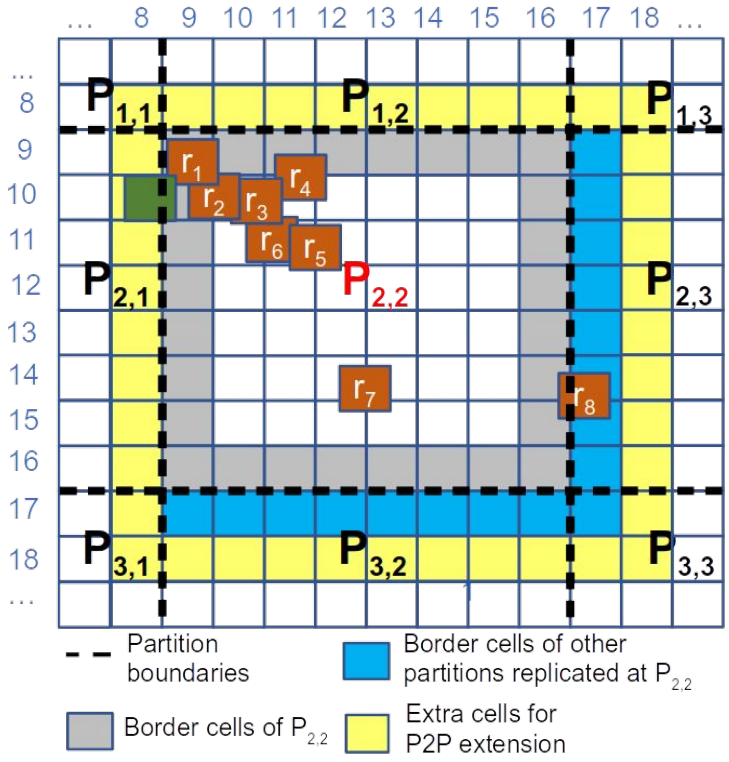


Figure 7: Example showing identified regions within a worker's partition. The regions are labeled in decreasing score, i.e., score(r1) > score(r2).

Definition 2 (Dependency graph). Let $R = (r_1, \dots, r_n)$ denote the list of candidate regions detected at a partition P, ordered by their score $score(r_i) \geq score(r_{i+1})$. A dependency graph $G(R)$ is a directed acyclic graph (DAG) that contains all candidate regions from R as vertices and has an edge between any two regions r_i and r_j if they overlap. The edge is directed from the region with the higher score towards the region with the lower score. Ties between regions are broken consistently by preferring the region with the lowest x coordinate, and then the region with the lowest y coordinate.

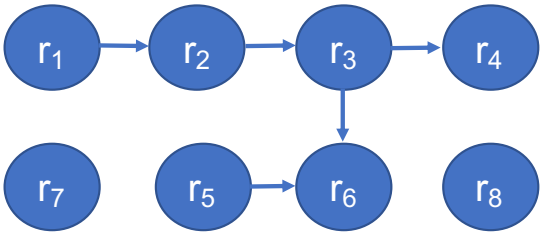


Figure 8: Dependency graph corresponding to the example of Figure 7.

Figure 8 depicts the dependency graph for the detected regions in Figure 7. Constructing the dependency graph for each partition $P_{x,y}$ is a local process. It utilizes only the partition's data and the data in the border cells of the neighbouring partitions $P_{x+1,y}$, $P_{x,y+1}$, and $P_{x+1,y+1}$, which is

replicated in the worker holding the partition. The regions that overlap the border cells of each partition (grey-shaded cells in Figure 7) may overlap with candidate regions detected in adjacent partitions. Thus, the dependencies induced by these regions cannot be depicted in the dependency graph. To represent these dependencies, we extend the dependency graph by adding artificial dependencies for all locally unknown candidate regions that potentially overlap with regions detected at neighbouring partitions.

Definition 3 (Extended dependency graph). The extended dependency graph $X(R)$ of a partition is a DAG containing: (a) $G(R)$, and (b) for each vertex $v \in G(R)$ with an upper-left corner contained in a border cell (i, j), one vertex v' for each one of the cells adjacent to (k, j) that belongs to a different partition, and an edge pointing from v' to v.

Intuitively, the extended dependency graph encodes the possible dependencies from regions detected at other partitions. Since these regions are unknown at the node constructing the graph, they are represented with the coordinates of the cell that would contain their upper-left corner. For example, region r_1 of partition $P_{2,2}$ has an upper-left corner at cell (9,9). The adjacent cells belonging to other partitions are the cells (8, 8), (8, 9), (8, 10), (9, 8), and (10, 8) of partitions $P_{1,1}$, $P_{1,2}$, and $P_{2,1}$. Any region from another partition that has a non-empty overlap with r_1 has its upper-left corner at one of these cells. Figure 9 depicts the extended dependency graph of partition $P_{2,2}$.

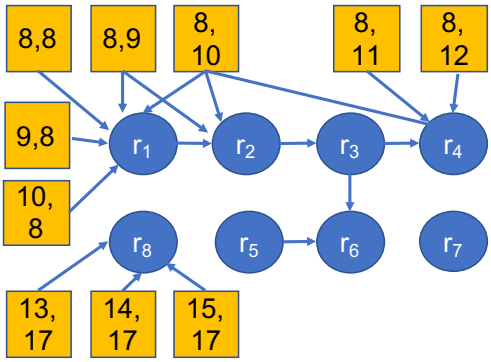


Figure 9: Extended dependency graph corresponding to the example of Figure 4.

The advantage of the extended dependency graph is that it allows establishing an upper bound on the number of regions in the partition that may be excluded from the results due to (chains of) overlapping regions contained in other partitions. Conversely, it allows each partition to derive a *lower bound* on the number of safe regions, i.e., the regions that will not be invalidated from the contents of other partitions, even by long chains of overlaps.

The progressive construction of the extended dependency graph takes place alongside the centralized algorithm that produces the local results progressively. Precisely, we slightly modify the local algorithm in two ways: (1) for every identified region that passes the filter of acceptable overlap, the modified algorithm invokes a function to add the region and its dependencies in the extended dependency graph, (2) it terminates when a sufficient number of safe regions has been detected.

Hybrid algorithm. The single-round algorithm is very conservative, requiring k safe regions to be obtained from every partition. In practice, this leads to additional load for extending the local graphs. To avoid this drawback, we also propose a hybrid algorithm, which strikes a balance between the pros and cons of the multi-round and single-round strategies. This hybrid algorithm covers the space between the single-round and multi-round algorithms by balancing the number of rounds and the number of results expected from each round. Intuitively, we can execute the single-round algorithm, requesting a smaller number of safe regions $k' \ll k$ per partition, aggregate the partial results and progressively ask for more results only from the partitions from which we already consumed at least one safe region. The partial results collected per round are a sorted subset of the final results (at least the next k' answers, but typically much more), and can be presented progressively to the user. For computationally expensive algorithms, the ability to progressively compute intermediate results is an essential requirement for later real-time exploratory data analysis [79].

3. Similarity Search and Join

In this section, we present our work on similarity search and join, which are fundamental operations for entity exploration. We consider different problems and settings that come up when dealing with heterogeneous entities in data lakes. Initially, we focus on set similarity joins, which are relevant for entities having textual or set-valued attributes. Then, we deal with similarity search for entities having multiple, heterogeneous attributes, such as textual, numeric and spatial. Finally, we present our work on similarity search for time series, representing entities with attribute values that evolve over time.

3.1. Preliminaries

3.1.1. Definitions

We assume a similarity function f that computes a similarity score between 0 and 1 for a pair of entities e_1 and e_2. We distinguish between *search* and *join* operations:

- In search operations, a query entity e is given, and the goal is to find in a collection of entities E the ones that are similar to e.
- In join operations, the goal is to find all pairs of similar entities between two collections E_1 and E_2.

Moreover, we distinguish between *threshold-based* and *top-k* operations:

- When a similarity threshold θ is given, the goal is to find all entity pairs with similarity at least θ.

- When a number k is given, the goal is to find the k pairs with the highest similarity.

More formally, we define the following operations.

Definition 4 (Threshold similarity search). Given an entity e, a collection of entities E, a similarity function f and a similarity threshold θ, threshold-based similarity search refers to identifying all entities $e' \in E$ having similarity to e at least θ, i.e., $f(e, e') \geq \theta$.

Definition 5 (k-NN similarity search). Given an entity e, a collection of entities E, a similarity function f and an integer k, k-nearest neighbour similarity search refers to identifying k entities $e' \in E$ having the highest similarity to e, i.e., $f(e, e') \geq f(e, e'')$ for any other entity $e'' \in E$.

Definition 6 (Threshold similarity join). Given two entity collections E_1 and E_2, a similarity function f and a similarity threshold θ, threshold-based similarity join refers to identifying all pairs of entities $e \in E_1$ and $e' \in E_2$ having similarity at least θ, i.e., $f(e, e') \geq \theta$.

Definition 7 (k-NN similarity join). Given two entity collections E_1 and E_2, a similarity function f and an integer k, k-nearest neighbour similarity join refers to identifying, for each entity $e \in E_1$ its k nearest neighbours $e' \in E_2$.

Definition 8 (Top-k similarity join). Given two entity collections E_1 and E_2, a similarity function f and an integer k, k-closest pairs similarity join refers to identifying k pairs of entities $e \in E_1$ and $e' \in E_2$ having the highest similarity scores among all possible pairs of entities in E_1 and E_2.

Notice that a join operation can also be executed within a single entity collection, i.e., $E_1 \equiv E_2$. In this case, it is referred to as a *self-join*.

The results of a similarity search or join operation can be represented as a *similarity graph*, where the vertices denote the entities and the edges are weighted according to the similarity between its vertices.

3.1.2. The filter-verification framework

Given a set of N entities, a naive method to execute a similarity join is to compare each entity with each other. However, this requires $N \times N$ comparisons, which has a high computational cost and does not scale for large numbers of entities.

Several algorithms have been proposed to avoid exhaustive pairwise comparisons for similarity joins. They typically follow a filter-verification framework [7], involving two main stages:

- The *filtering* stage computes a set of candidates for each entity, pruning all those that cannot match. In other words, it discards all true negatives, while allowing some false positives.
- The *verification* stage computes the actual similarity between the given entity and each of its candidates to remove the false positives.

Instead of comparing each entity with all other entities, the comparisons are only performed between *candidates*. This makes the process more efficient and scalable, assuming that: (a) the number of candidates is much smaller than the total number of entities, and (b) identifying

candidates is not too expensive. Hence, filtering algorithms need to be both *effective*, i.e., returning as few candidates as possible, and *efficient*, i.e., the applied filters being lightweight compared to the cost of verification. Otherwise, they may not pay off in practice.

3.2. Set similarity joins

The types of filters applied in the filtering stage depend on the type of entities and the similarity measure used to compare them. Next, we focus on *set similarity joins* for the following reasons. First, it is common for entities to be characterized by sets of tokens, such as keywords or tags; in other words, to have *categorical attributes* in their profiles. For instance, a scientific publication is often associated with a set of keywords, which may have been explicitly assigned by its authors or automatically extracted from its title or abstract. Similarly, a product or a company is often associated with keywords or tags, either assigned manually or by automatically extracting them from textual descriptions. Moreover, entities in a HIN can be associated with sets of other entities of the same or different type found in their neighbourhood. The neighbourhood of an entity can be determined by one or more specified metapaths.

Hence, in the following, we consider similarity joins between sets of tokens, and we adopt Jaccard similarity as the similarity measure. The Jaccard similarity of two sets A and B is defined as: $J(A,B) = |A \cap B| / |A \cup B|$. Other common set similarity measures, such as Cosine or Dice, can be handled similarly. This is because the employed filtering algorithms essentially work internally by converting the given similarity threshold to a corresponding *overlap* threshold, i.e., a threshold on the minimum number of common tokens between two sets. For the same reason, it is possible to handle similarity joins among strings with edit distance. This is handled by converting strings to sets using tokenization or q-gram extraction. The given threshold on string edit distance can also be converted to an equivalent threshold on set overlap [7].

3.2.1. State of the art

An overview of the main filtering methods for string and set similarity joins in the literature is presented in Table 1. We have classified the methods in four groups: (i) centralized algorithms offering exact solutions for similarity joins with a single predicate; (ii) algorithms for parallel and distributed processing; (iii) approximate algorithms; and (iv) methods that are dealing with more complex matching criteria. The listed methods are characterized by the type of operation they perform (e.g., search or join), the similarity measure they assume (e.g., token-based or character-based), the type of filters they use (e.g., prefix-based or partition-based) and the index structure they employ (e.g., inverted index or tree).

A survey discussing these works in more detail, as well as their relationship to Entity Resolution methods based on the Blocking framework, is published in [8].

Table 1: String and set similarity join methods.

Method	Operation	Similarity	Filters	Index
(I) Exact, centralized, single predicate algorithms				
GramCount [9]	string join	Edit Distance	length, count, position	q-grams table
MergeOpt [10]	set join	Overlap	Count	inverted index
FastSS [11]	string join	Edit Distance	deletion neighbourhood	dictionary
SSJoin [12]	set join	Overlap	prefix	DBMS
All-Pairs [13]	vector join	Cosine	prefix	inverted index
DivideSkip [14]	string search	Edit Distance, Overlap	length, position, prefix	inverted index
Ed-Join [15]	string join	Edit Distance	prefix+mismatching q-grams	inverted index
QChunk [16]	string join	Edit Distance	prefix+q-chunks	inverted index
VChunkJoin [17]	string join	Edit Distance	prefix+chunks	inverted index
PPJoin [18]	set join	Overlap	prefix, positional	inverted index
PPJoin+ [18]	set join	Overlap	prefix, positional, suffix	inverted index
MPJoin [19]	set join	Overlap	min-prefix	inverted index
GroupJoin [20]	set join	Overlap	prefix+grouping	inverted index
AdaptJoin [21]	set join	Overlap	adaptive prefix	inverted index
SKJ [22]	set join	Overlap	prefix-based+set relations	inverted index
TopkJoin [23]	top-k set join	Overlap	prefix-based	inverted index
JOSIE [24]	top-k set search	Overlap	prefix, position	inverted index
PartEnum [25]	set join	Hamming, Jaccard	partition-based	clustered index
PassJoin [26]	string join	Edit Distance	partition-based	inverted index
PTJ [27]	set join	Overlap	partition-based	inverted index
Bed–Tree [28]	string search/join	Edit Distance	string orders	B^+-tree
PBI [29]	string search	Edit Distance	reference strings	B^+-tree
MultiTree [30]	set search	Jaccard	tree traversal	B^+-tree

Trie-Join [31]	string join	Edit Distance	subtrie pruning	trie
HSTree [32]	string search	Edit Distance	partition-based	segment tree
Trans [33]	top-k set search	Jaccard	transformation distance	R-tree
(II) Parallel & distributed algorithms				
FuzzyJoin [34]	set/string join	Hamming, ED, Jaccard	ball-hashing, splitting, anchor points	lookup tables
VernicaJoin [35]	set join	Overlap	prefix, positional, suffix	inverted index
MGJoin [36]	set join	Overlap	multiple prefix	inverted index
MRGroupJoin [27]	set join	Overlap	partition-based	inverted index
FS-Join [37]	set join	Overlap	segment-based	inverted index
Dima [37]	search, join, top-k	Jaccard, ED	segment-based	global & local
(III) Approximate algorithms				
ATLAS [38]	vector join	Jaccard, Cosine	random permutations	inverted index
BayesLSH [39]	set join	Jaccard, Cosine	All-Pairs / LSH	All-Pairs / LSH
CPSJoin [40]	set join	Jaccard	LSH-based	sketches
(IV) Algorithms for complex matching				
LS-Join [41]	local string join	Edit Distance	length, count	inverted index
pkwise [42]	local set join	Overlap	k-wise signatures	inverted index
pkduck [43]	abbreviation matching	Custom	extension of prefix filter	trie
Fast-Join [44]	fuzzy set join	Bipartite graph matching	token sensitive signatures	inverted index
SilkMoth [45]	fuzzy set join	Bipartite graph matching	weighted token signatures	inverted index
MF-Join [46]	fuzzy set join	Bipartite graph matching	partition-based	inverted index
MultiAttr [47]	set search/join	Overlap	tree traversal	prefix tree
Smurf [48]	string matching	Jaccard, ED	random forest	inverted index
AU-Join [49]	string join	Syntactic, Synonym, Taxonomy	pebbles	inverted index

3.2.2. Basic filters and algorithms

In our work, we employ two filters for set similarity joins: size filter [25] and prefix filter [13]. The former is a lightweight but effective filtering technique which is typically used as a first filtering step by all algorithms in the literature. The latter constitutes the basic filtering condition employed by most state-of-the-art methods. Several extensions and adaptations to prefix filtering have been proposed in subsequent works. Nevertheless, a recent experimental survey has shown that the additional overhead of applying more elaborate filtering criteria usually does not pay off in practice [50]. We briefly present size filter and prefix filter below.

Size filter. If two sets R and S have Jaccard similarity $\varphi(R,S) \geq \theta$, then the size of S must be at least $\lceil |R| \times \theta \rceil$ and at most $\lfloor \frac{|R|}{\theta} \rfloor$. Hence, given a query set R and a similarity threshold θ, we need to consider only those sets with sizes between the two bounds mentioned above as candidates.

Prefix filter. If two sets R and S have Jaccard similarity $\varphi(R,S) \geq \theta$, then the $\boldsymbol{\pi_R}$-prefix of R (i.e., the first $\boldsymbol{\pi_R}$ tokens of R in a given order) and the $\boldsymbol{\pi_S}$-prefix of S (in the same order) must share at least one token. Hence, given a query set R, we need to consider only those sets whose prefix contains one of the tokens in the prefix of R as candidates. By sorting the tokens in increasing order of frequency, we can obtain prefixes that comprise the least frequent tokens, thus yielding a smaller number of candidates.

Based on the above filters and state-of-the-art algorithms [13][23], we use the following processes to perform threshold-based, k-NN, and top-k set similarity joins. In the description, we assume self-join operations; joins between two different collections are performed analogously.

Threshold join. The process is illustrated in Figure 10. The algorithm iterates over each query set R in the input collection. First, the required prefix of R is computed, based on the given similarity threshold. At this stage, we do not yet have a candidate set S, so the selected prefix of R is computed pessimistically (i.e., to accommodate any valid length of S). For each token in the prefix, a look-up is performed on an inverted index that contains all sets according to the tokens in their prefixes. This look-up produces the complete set of candidates for R, since these are the sets that contain in their prefix a common token with the prefix of R. Then, for each candidate set S, we apply the two aforementioned filters. First, the size filter is applied, to prune candidates whose size exceeds the required limits. Then, for each pair (R,S), the prefix filter is applied, adjusting the prefixes to the two given sets and checking whether they still contain a common token. Each candidate that survives both filters is verified to compute the exact similarity score. If this score is above the specified threshold, the pair (R,S) is added to the output.

k-NN join. As shown in Figure 11, the algorithm iterates over each query set R in the input collection to compute its k-nearest neighbours. However, instead of computing the whole prefix of R, as before, it examines the tokens progressively. Moreover, when the next token is selected, an upper bound (here called $Unseen$) is established for the similarity score of any future candidates. For each examined token, it performs a lookup in the inverted index to retrieve candidates. These candidates are filtered using the size and prefix filters, as before. The successful candidates are verified, and the results are inserted into a priority queue based on their similarity score, instead of

being sent directly to the output. If the top element of the priority queue is higher than the $Unseen$ upper bound, this result can be sent to the output, since it is guaranteed to be the next best result. If k results have been sent to the output, the process for this query set R terminates; otherwise, the next token of R is examined.

Top-k join. The process is similar to the one of k-NN join, as illustrated in Figure 12. The difference is that the query sets are not examined sequentially. Instead, in each iteration, the algorithm decides which query set R to consider next. The order depends on the current $Unseen$ upper bound of the existing query sets in the input collection. Thus, all query sets are considered and processed "simultaneously" to produce candidates and generate results that populate the global top-k list.

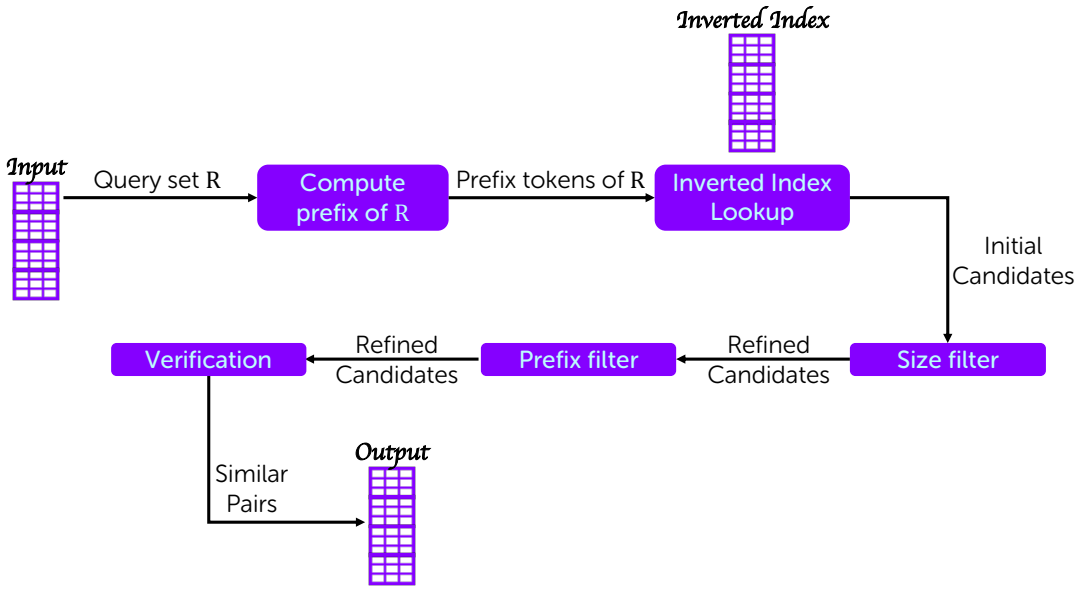


Figure 10: Flow of a threshold-based set similarity join operation.

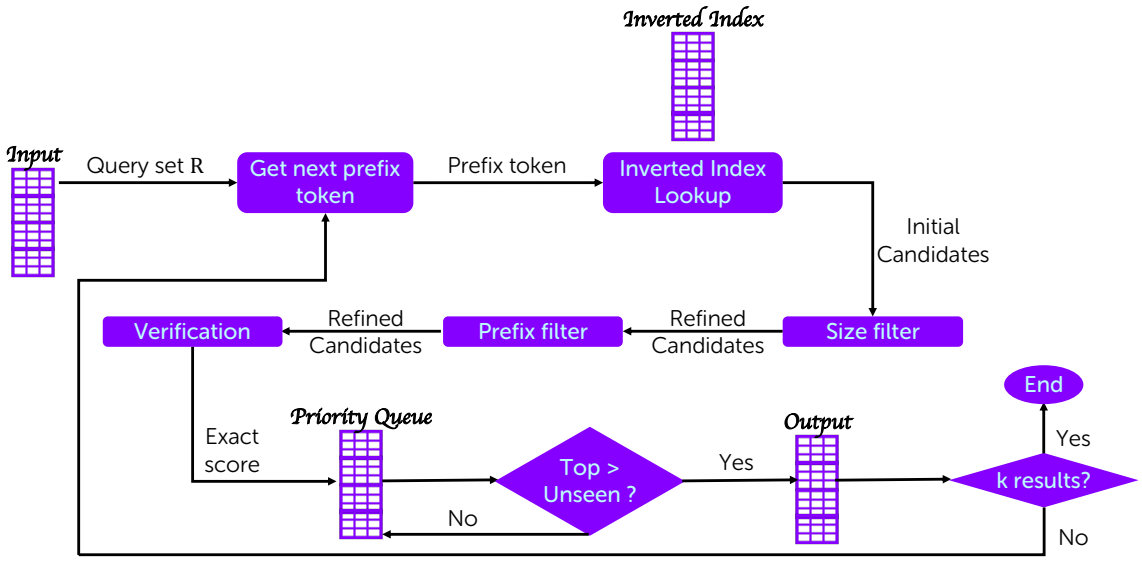


Figure 11: Flow of a k-NN set similarity join operation.

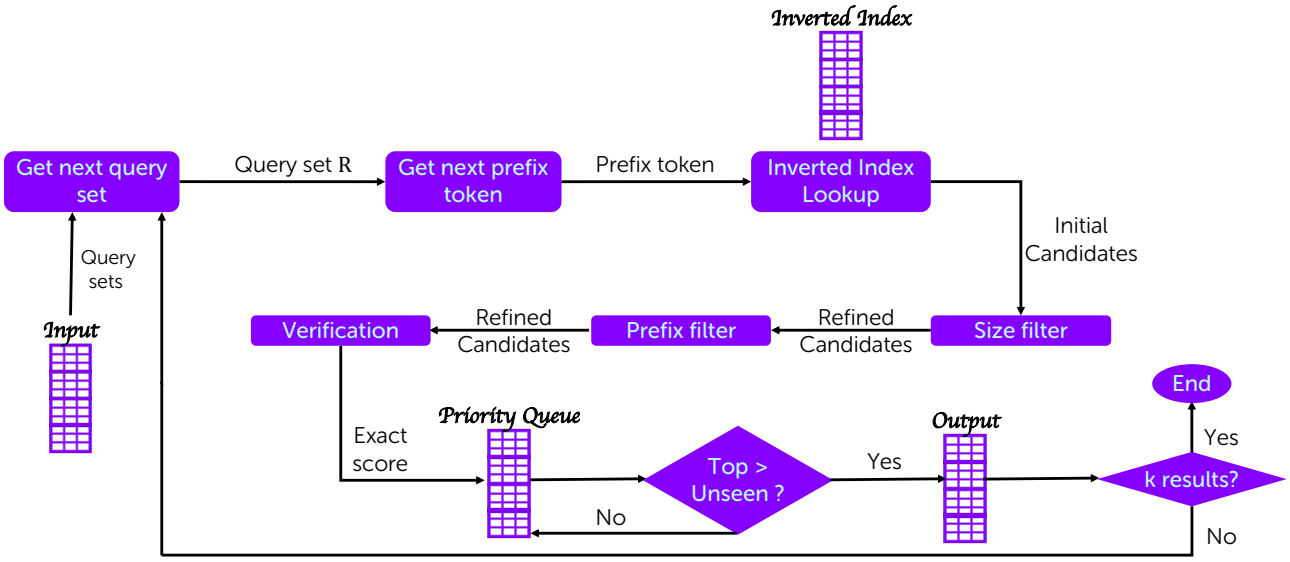


Figure 12: Flow of a top-k set similarity join operation.

3.2.3. Fuzzy set similarity joins

So far, we have assumed Jaccard similarity as the similarity measure used in set similarity joins. As well as other similar set-based measures such as Cosine or Dice, it assumes a binary match between pairs of tokens. That is, two tokens are considered as either identical or different. However, in many cases, it is desired to allow a *fuzzy* match over tokens. For example, this need occurs when comparing companies based on the descriptions of their products, or authors based on the titles or keywords of their publications. Such cases can be handled by fuzzy set similarity joins [44][45][46]. In this setting, each element of a set is represented also as a set. Thus, two elements can have a continuous similarity score between 0 and 1.

Given two sets R and S, their similarity is defined as the maximum matching score of the bipartite graph formed by their elements. The example illustrated in Figure 13 depicts two sets $R = \{r_1, r_2, r_3\}$ and $S = \{s_1, s_2, s_3, s_4\}$ where the pairwise similarities among their elements are shown as the edges of the respective bipartite graph. The maximum matching in this bipartite graph is the one illustrated in purple color. Specifically, it comprises the edges (r_1, s_1) and (r_2, s_2), resulting in a similarity score of 0.32 for the pair of sets R and S.

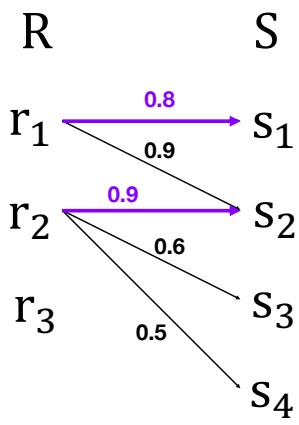


Figure 13: Example illustrating the matching score between two fuzzy sets.

The approach for executing a fuzzy set similarity join operation follows the filter-verification framework. The size filter presented above still holds and can be used to prune pairs of sets (R, S) based on the number of elements they contain. However, the prefix filter cannot be applied in the same way, since now each set comprises several sets that contain tokens of their own. Following the state-of-the-art algorithm [45], this part of the process is substituted by the mechanisms described below.

Signature generation. Given a query set R and a similarity threshold θ, an algorithm is used to assign a similarity threshold θ_r to each element $r \in R$ and accordingly derive a prefix for r. The union of the prefix tokens of all elements constitutes the signature of R and is the set of tokens used to retrieve candidates from the index.

Check filter. A pair (R, S) is a candidate pair if there exists at least one element $r \in \mathrm{R}$ and one element $s \in \mathrm{S}$ such that $sim(r, s) \geq \theta_r$, where θ_r is the threshold assigned to element r. Otherwise, this pair can be pruned.

NN filter. For each element $r \in R$, its nearest neighbour $s \in S$ is identified, and the similarity score between r and s is computed. The sum of these similarity scores constitutes an upper bound for the matching score. Hence, if this sum is below θ, the pair can be pruned.

Next, we first describe how the state-of-the-art algorithm [45] uses the above filters to perform threshold join on fuzzy sets. Then, we explain how we extend it to additionally support k-NN and top-k joins.

Threshold join. As illustrated in Figure 14, the algorithm iterates over each fuzzy set R in the input collection. For each R, it first computes its signature and uses the tokens in the signature to retrieve candidates from the inverted index. Then, for each candidate S, the three filters, namely *size filter*, *check filter* and *nearest neighbour filter*, are applied sequentially. If the candidate successfully passes all filters, the verification function is invoked to compute its matching score with R. If the score exceeds the given threshold, the pair (R, S) is added to the output.

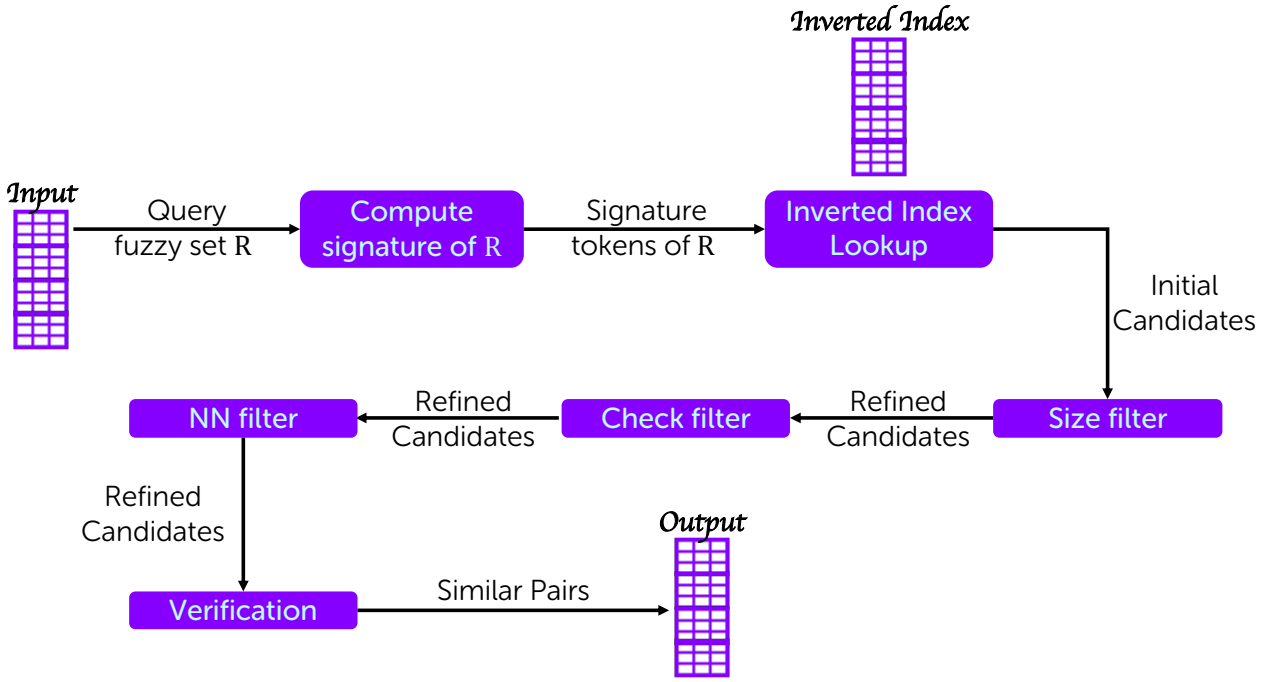


Figure 14: Flow of a threshold-based fuzzy set similarity join operation.

k-NN join. The algorithm iterates over each fuzzy set R in the input collection. However, as shown in Figure 15, we introduce three main differences compared to threshold join, as explained below.

First, instead of computing the entire signature of R in the beginning, this is done progressively, expanding the signature by one token at a time. The reason is that the length of the signature, i.e., the number of tokens it comprises, depends on the desired similarity threshold. For threshold-based join, this is fixed. Hence the signature can be computed a priori. In contrast, in the case of k-NN join, the threshold varies during the execution of the algorithm. In particular, at any point in time, the current threshold is equal to the similarity score of the k-th best current candidate. Hence, as better candidates are discovered, the threshold gradually increases, and consequently, the signature length gradually decreases. Thus, computing the full signature at the beginning of the process would have led to the examination of unnecessary tokens. Instead, we select tokens progressively and stop when the number of examined tokens becomes equal to the current length of the signature. Moreover, whenever the next signature token is selected, an $Unseen$ upper bound is computed for any remaining future candidates.

The second difference lies in the application of filters. Instead of sending the results of each filter to the next, we use a priority queue to maintain intermediate results sorted in decreasing order of a computed upper bound. Specifically, after each filter is applied, an upper bound is computed for the candidate that passed the filter. We show how these upper bounds are computed next. Assume a pair of sets (R, S) with $|R| < |S|$. The upper bound for the size filter is:

$$SF_{UB}(R,S) = \frac{|R|}{|S|}.$$

If the pair (R, S) passes the check filter, this upper bound is updated (tightened) as follows:

$$CF_{UB}(R,S) = \left(\sum_{\mathrm{r} \in \mathrm{R_{CF}}} score_{CF}(r) + \sum_{r \notin R_{CF}} threshold_{CF}(r) \right) / |S|$$

where $R_{CF} \subseteq R$ is the subset of elements of R that have passed the check filter, $score_{CF}(r)$ is the maximum similarity score that an element that has passed the check filter has achieved, and $threshold_{CF}(r)$ is a threshold that is associated with each element during the signature generation process.

Furthermore, if the pair (R, S) passes the nearest neighbour filter, its upper bound is again updated (tightened) as follows:

$$\mathrm{NNF_{UB}}(R,S) = \left(\sum_{\mathrm{r} \in \mathrm{R}} score_{NNF}(r) \right) / |S|$$

where $score_{NNF}(r)$ is the similarity score between r and its nearest neighbour.

At each step, the algorithm decides whether to generate new candidates by computing the next signature token and retrieving candidates from the inverted index or to further refine an already existing candidate that is kept in this priority queue. The decision is made by examining whether the $Unseen$ upper bound is greater than the score of the top item in the queue. Whenever a candidate is selected from the queue, the algorithm checks its status, i.e., which filters it already has passed, and applies the next filter. If all filters have been successfully applied, the candidate is verified, i.e., its exact score is computed.

The third difference lies in the handling of the verified candidates. Instead of sending the resulting pairs to the output, they are maintained in another priority queue that holds candidate results. If the score of the top item in this queue becomes greater or equal to both the $Unseen$ upper bound and the score of the top item in the priority queue that holds intermediate results, the pair is sent to the output. The algorithm terminates once the output contains k results.

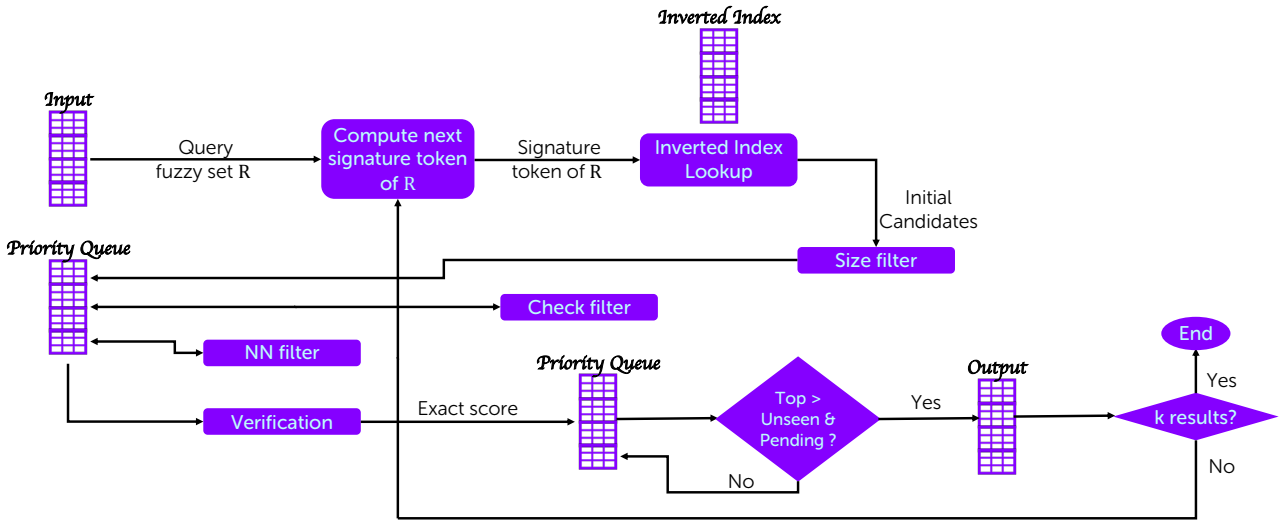


Figure 15: Flow of a k-NN fuzzy set similarity join operation.

Top-k join. The process is illustrated in Figure 16 and it is very similar to the one for k-NN join described above. The difference is that the query sets are not examined sequentially. Instead, in each iteration, the algorithm determines which query set to consider next. This depends on the current $Unseen$ upper bound of the existing query sets in the input collection. Therefore, all query sets are processed in an intertwined manner, producing candidates that contribute to the global top-k list of results.

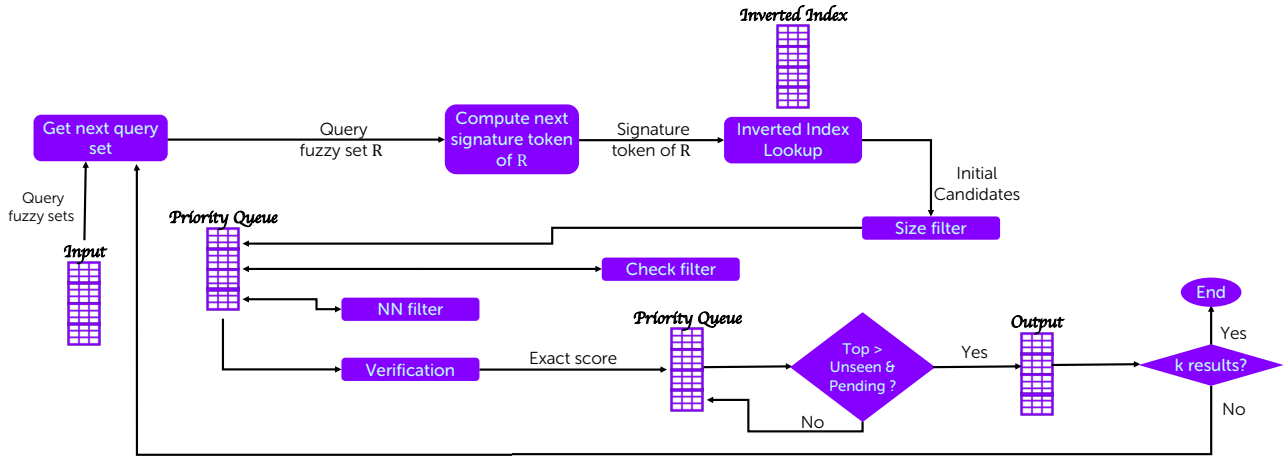


Figure 16: Flow of a top-k fuzzy set similarity join operation.

3.2.4. Extension to weighted sets

So far, we have focused on similarity search and join among entities based on their (fuzzy) set representations but without taking the importance of individual entities into account. However, entities may be associated with ranking scores, which can be computed based on their position in the HIN, as presented in Section 2, or other properties. This information is useful and needs to be considered during data exploration to guide the user's attention towards more important entities. For instance, when exploring similar companies, the user may be interested in prioritizing results that involve companies with a higher centrality score in the network, higher revenue, or a larger number of employees or projects. In the following, we show how to incorporate entity weights on (fuzzy) set similarity joins, thus providing a means to combine ranking and similarity joins during the analysis.

Definition 9 (Weighted Similarity Score). Assume two entities R and S with weights $w_R \in (0,1]$ and $w_S \in (0,1]$, respectively. We denote their original similarity score, based on their (fuzzy) set representations, as $sim(R,S)$. Then, we define the weighted similarity score of R and S as follows:

$$wsim(R,S) = \frac{w_R + w_S}{2} \times sim(R,S).$$

As shown from the above equation, if $w_R = w_S = 1$, then $wsim(R,S) = sim(R,S)$, which corresponds to the case without considering entity weights. Instead, for $w_R < 1$ or $w_S < 1$, we

have $wsim(R,S) < sim(R,S)$. This can be thought of as penalizing the matching score between R and S if R and S have lower importance.

We now explain how to adapt the original threshold to obtain a relative one that takes into consideration set weights. Assume a pair of entities R and S, and a threshold θ on their weighted similarity score. We can derive a corresponding weight-aware threshold θ_w on their (fuzzy) set representations, as follows:

$$wsim(R,S) \geq \theta \Rightarrow \frac{\mathrm{w_R} + \mathrm{w_S}}{2} \times sim(R,S) \geq \theta \Rightarrow sim(R,S) \geq \frac{2 \times \theta}{w_R + w_S} = \theta_w$$

Notice that the weight-aware threshold θ_w depends on both weights w_R and w_S, thus it varies for each pair of entities R and S. Moreover, since $w_R, w_S \in (0,1]$, it holds that $\theta_w \geq \theta$. In other words, entities with weights lower than 1 need to achieve a higher similarity score to compensate for their lower importance.

Based on the above, we can translate a similarity search or join operation with threshold θ on weighted entities to a corresponding operation without weights and equivalent threshold θ_w. However, since θ_w depends on both w_R and w_S, to be able to generate candidates in the filtering stage, we also need a weight-aware threshold that depends only on the query set R. Given that $w_S \in (0,1)$, we can observe that

$$\theta_w = \frac{2 \times \theta}{w_R + w_S} \geq \frac{2 \times \theta}{\mathrm{w_R} + 1} = \theta_{w,r}$$

This implies that we can use $\theta_{w,r}$, which depends only on r, to generate candidates for r. Since $\theta_{w,r} \leq \theta$, any candidate that cannot pass the threshold $\theta_{w,r}$ will also not pass the threshold θ_w. From this, for each identified candidate S, we can calculate and use the tighter threshold θ_w, i.e., consider w_S to further refine and verify each particular pair.

3.3. Similarity search over heterogeneous attributes

In the previous section, we have addressed similarity search and join operations for entities with attribute values represented as sets. Next, we consider entities that have multiple attributes of different types, including set-valued, numeric, and spatial.

3.3.1. Problem definition

So far, we have focused on (fuzzy) set similarity search and join, as this is particularly relevant in HINs. Given a HIN and a metapath, each node N_i is associated with a set of neighbouring nodes. Therefore, (fuzzy) set similarity joins can be used to compute the similarity among nodes based on how many common or similar neighbours they share. Nevertheless, entities usually have other

properties as well, such as numeric or geospatial attributes. For instance, a company may be described by attributes such as its revenue, the number of employees, or its location. Thus, multiple attributes of different types need to be taken into account when searching for similar entities.

In particular, we focus on *aggregate top-k similarity search queries* on entities with heterogeneous attributes. When multiple attributes of different types are involved, it may not be convenient or intuitive for the user to search by specifying appropriate thresholds for each attribute. Instead, it is preferable to compute a top-k list of most similar entities to the query, where the similarity takes into account all requested attributes. Based on this, we formally define the problem below.

Definition 10 (Multi-attribute Top-k Similarity Search). Assume a collection E of entities, where each entity $e \in E$ is described by a set of attribute-value pairs, i.e.,

$$e = \{(attr_1, val_1), \dots, (attr_n, val_n)\}$$

where attributes can be of different types. Assume also a query $Q = \langle C, k \rangle$, where $C = \{(attr_1, q_1), \dots, (attr_n, q_n)\}$ indicates the desired values for each attribute and k denotes the number of results to return. Let $w_i \in [0,1]$ be a weight for each attribute, and $f_i(q, val)$ be a scoring function that computes a similarity score for each attribute for the corresponding value in the query. The goal is to return the top-k results, where the score of each result e is computed by:

$$score(C, e) = \sum_i w_i \times f_i(q_i, val_i)$$

In a nutshell, our approach works as follows. First, for each attribute, a ranked list of results is generated, ordered descending according to the similarity score based on that attribute. Then, the individual lists are aggregated using a weight function to obtain the final top-k list. This allows the user to specify different weights at query time, thus indicating different preferences to different attributes, which is particularly useful for data exploration.

3.3.2. Distance calculations

For each type of attribute, we assume a distance function d that measures the distance between any value of this attribute and the value indicated in the query. Next, we discuss set-valued, numeric and spatial attributes; other types of attributes can be treated analogously.

Set-valued attributes. Given two sets R and S, we define their distance as $d(R, S) = 1 - sim(R, S)$, where $sim(R, S)$ is defined as Jaccard similarity if binary matching is assumed among their elements, or as the maximum matching score on the corresponding bipartite graph, if fuzzy matching is assumed (see Section 0). Hence, k-NN similarity search over such attributes is performed as described in Sections 3.2 and 0, respectively.

Numeric attributes. Given two numeric values v_i and v_j, we define their distance as $d(v_i, v_j) = |v_i - v_j|$. k-NN similarity search on numeric attributes is straightforward and can be done efficiently by first sorting the values or assuming a B+ tree index over the attribute.

Spatial attributes. Given two spatial points $p_i = (x_i, y_i)$ and $p_j = (x_j, y_j)$, we define their distance as the Euclidean distance of their coordinates, i.e.,

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

For k-NN search over spatial attributes, we assume a spatial index, such as R-tree.

Scaling. The distances across attributes, especially among those of different types, may largely vary in terms of scale. This needs to be addressed to be able to meaningfully combine these scores into a single aggregate score for ranking. Although we can assume that the minimum value for distance is zero, we cannot always establish a value for the maximum distance. Thus we may not be able to use a typical scaling method such as min-max normalization to rescale the range of distance values to a fixed interval such as $[0,1]$. Instead, given the original distance d, as defined above, we compute a scaled distance as follows:

$$d_\sigma = \frac{d}{\sigma}$$

where σ is a scaling factor. Since it may not be intuitive or convenient to require the user to specify this scaling factor for each attribute, it is preferable to determine it automatically from the data. To this end, given a query Q, we set the scaling factor σ_i for each attribute i to be equal to the distance of the k-th nearest neighbour for that attribute. By default, we set $k = 1$, i.e., we use the distance to the first nearest neighbour as the scaling factor. Notice that, to avoid division by zero, we select the k-th nearest neighbour after omitting those that have a distance to the query equal to zero, i.e., those having a value that is equal to the one specified in the query.

Scoring function. Finally, we need to define a scoring function f that is used to assign a ranking score to each attribute based on the corresponding (scaled) distance. We use exponential decay for this purpose:

$$f = e^{-\lambda \times d_\sigma}$$

where $\lambda > 0$ is the exponential decay constant which controls the rate of decay. Observe that, when $d = 0$, then $f = 1$, while when d grows towards infinity, the value of f drops to zero.

3.3.3. Result aggregation

To compute the aggregate list of top-k most similar entities to the specified query, we follow an approach based on rank aggregation. Assuming that the entities have been ranked individually for each one of the attributes in the query, it is possible to apply the Threshold Algorithm [51] to

aggregate these individual ranked lists into the final top-k list. Figure 17 illustrates the idea for the case of three different attributes, one that is set-valued, one numeric, and one spatial.

Summarizing, this algorithm works as follows. Each ranked list is processed in parallel, and at each iteration, the next item in each list is accessed. These items constitute candidate results. For each new candidate retrieved from one of the lists, random access to the other lists is performed to retrieve all scores for this candidate and compute its exact total score. Thus, throughout the execution of the algorithm, a list of candidates is maintained together with their exact scores. Moreover, at each iteration, the algorithm uses the scores of the top items in each list to establish an upper bound for the score of all items not encountered so far. Since the items are retrieved from each list in descending order of their score, this upper bound always decreases. Once it becomes lower than the exact scores of k candidates, the algorithm terminates, returning the top-k candidates.

In the Threshold Algorithm, it is assumed that the ranked lists for each attribute are already known (i.e., materialized), and that, given an item in one of the lists, it is possible to perform random access (i.e., a lookup) to retrieve its score in the other lists. In our case, these lists are not available in advance. They are produced on-the-fly, as the result of k-NN queries executed on each attribute. This introduces a trade-off regarding the number of k_i results to request for each attribute. If k_i is small, the items contained in the ranked lists may not suffice for obtaining the global top-k results, since the lists may be exhausted before the threshold has dropped below the score of the current k-th best candidate. In that case, the search can either terminate, issuing a message that the results are approximate (i.e., some better results may exist, up to a maximum score), or it may proceed by invoking an additional round of k-NN queries with larger values of k_i, to obtain longer lists that can provide a sufficient number of candidates. Conversely, if k-NN queries with large values of k_i are issued, this may incur unnecessary computations for computing individual ranked lists of much larger sizes than eventually needed. Therefore, the number of k_i results to retrieve per attribute in each occasion is a tuning parameter that affects the performance of the algorithm and depends on the selectivity of each attribute, the computational cost of k-NN queries over each attribute, and the degree of correlation among attributes. One way to tune this parameter is through the visual analytics components. We also investigate how to automatically tune this parameter based on the analysis of previous queries.

Performing random accesses assumes that the algorithm can access all attributes of each candidate entity to compute the distance of any required attribute value to the query on-demand. However, in some cases, the algorithm may only have access to the scores obtained from the retrieved ranked lists. Then, longer lists are again needed for each attribute, so that each candidate appears in all the lists or in a sufficiently large number of them to obtain a partial score that is large enough to exceed the current upper bound. This factor also contributes to the choice mentioned above for setting the individual k_i parameters per attribute.

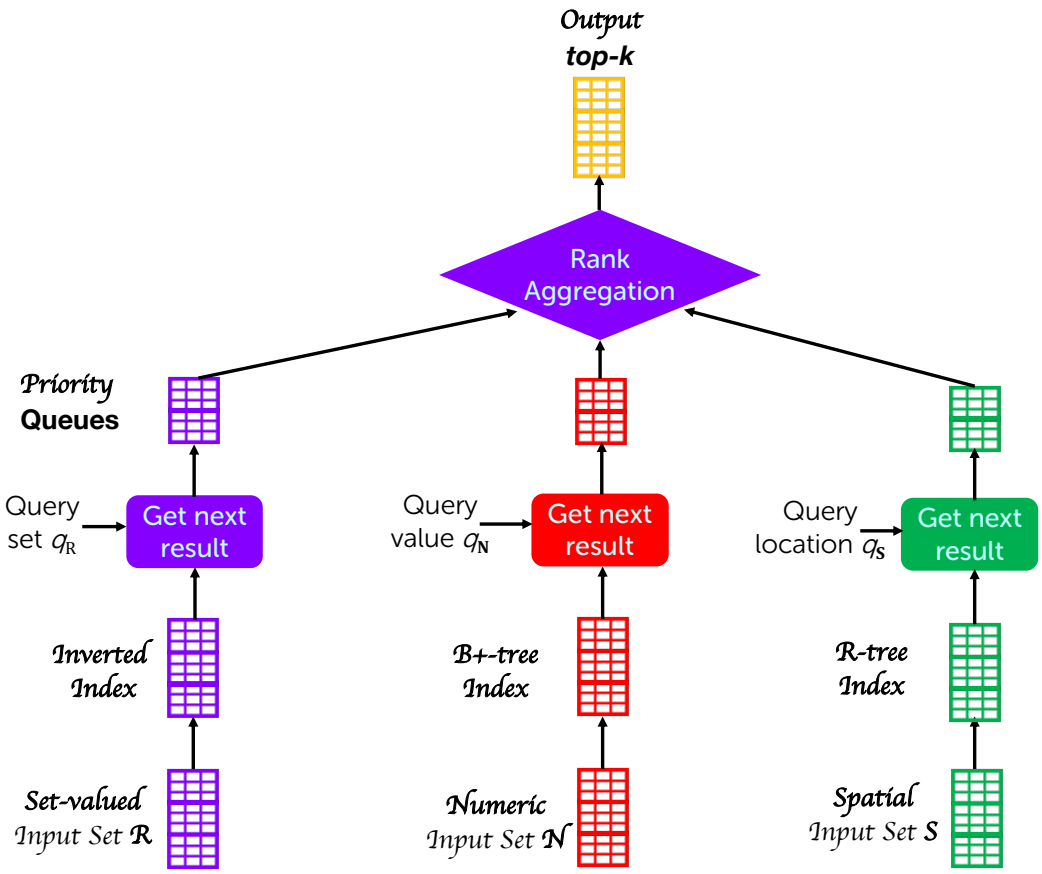


Figure 17: Processing flow for top-k similarity search across three attributes of different types.

3.4. Similarity search over time series

In this section, we focus on similarity search over time series data. Time series is a specific type of data that is crucial in many domains and applications. A time series indicates how the value of an entity's attribute changes over time. Typical examples are a company's revenue or the price of its stock. First, we outline the state of the art. Then, we define more formally the problem addressed in the project, and we present an overview of our approach.

3.4.1. State of the art

Similarity search over time series has attracted a lot of research interest [80]. One well-studied family of approaches includes wavelet-based methods [81]. These rely on Discrete Wavelet Transform to reduce the dimensionality of time series. Then, they generate an index using the coefficients of the transformed sequences. The Symbolic Aggregate Approximation (SAX) representation [82] has led to the design of a series of indices. The SAX representation of a time series is derived from its Piecewise Aggregate Approximation [100] by quantizing its segments along the y-axis. SAX-based indices include iSAX [83], iSAX 2.0 [84], iSAX2+ [85], ADS+ [86],

Coconut [87], DPiSAX [88], and ParIS [89]. In the same line of work, the ULISSE index [90] has been proposed for answering similarity search queries over time series of variable length. All these indices are designed for similarity search over entire time series, i.e. for whole sequence matching.

In addition, several approaches have been proposed for subsequence matching. In this case, a query subsequence is provided, and the goal is to identify matches of it across one or more time series, typically of large length. The UCR suite [91] offers a framework comprising various optimizations regarding subsequence similarity search. In computing full-similarity-joins over large collections of time series, i.e., to detect for each possible subsequence its nearest neighbor, the matrix profile [92] keeps track of Euclidean distances among each pair within a similarity join set (i.e., a set containing pairs of each subsequence with its nearest neighbor).

Finally, several works deal with detecting correlated time series. In particular, common approaches compute pairwise statistics (e.g., Pearson correlation, beta values), often focusing on streaming time series [93][94][95].

As explained next, the problem we address here differs from the above settings. Instead of identifying matches of a query subsequence against one or more time series, we are interested in discovering *locally similar* pairs of *time-aligned subsequences* of equal duration within a given collection of time series.

3.4.2. Problem definition

A time series is a time-ordered sequence of values $X = \{X_1, X_2, \dots, X_k\}$, where X_i is the value at the i-th timestamp and k is the length of the series (i.e., the number of timestamps). We consider a set of co-evolving time series. All time series are time-aligned and each series has a value at each of the k timestamps. Given a set of such co-evolving time series, our goal is to find pairs of time series that have similar values locally over some time intervals of significant duration. More specifically, we define two time series to be locally similar as follows.

Definition 11 (Locally Similar Time Series). Two co-evolving time series X_i and X_j are locally similar if there exists a time interval I spanning at least δ consecutive timestamps such that at every timestamp in I their corresponding values do not differ by more than a given threshold ε.

The threshold ε expresses the maximum tolerable deviation per timestamp between two time series. Our goal is to find all such pairs of time series. Thus, we treat the problem as a self-join operation over the dataset, specifying as join criteria the distance threshold ε and the minimum time duration δ of qualifying pairs.

An example is illustrated in Figure 18. The detected pairs for specified ε and δ are the locally similar time series shown within grey ribbons. Since two time series may be locally similar in more than one intervals, their matching subsequences are considered as two different pairs, one for each interval. For instance, in this example, the green and red time series yield two matching pairs in different time intervals.

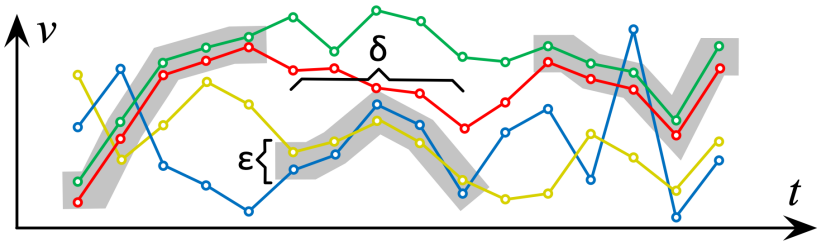


Figure 18: Locally similar pair discovery over a set of time series.

3.4.3. Our approach

Next, we outline our approach for addressing the problem defined above; more details can be found in [96]. We start with a baseline algorithm that uses a sweep line to scan the co-evolving time series throughout their duration. While doing so, it validates and keeps all the pairs that satisfy the given constraints, employing a value discretization scheme per timestamp. Then, we present an optimized method that reduces the number of pairs to consider by judiciously probing candidates at selected timestamp values, referred to as checkpoints. This significantly prunes the search space without missing any qualifying results.

To reduce the candidate pairs that need to be checked at each timestamp, we discretize the values of all time series in bins of size ε. Time series with values within the same bin at any timestamp form candidate pairs. Moreover, 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.

To generate all candidate pairs, we assume a sliding window of size ε along the y-axis, whose upper endpoint coincides with each value under consideration at a given timestamp. Then, all values of time series contained within the same window, form candidate pairs. At each timestamp, the process of finding all the pairs comprises two steps. The first step, *filetering*, searches among time series values in the same or adjacent bins to detect candidate pairs. The second step, *verification*, checks, for each candidate pair, whether the similarity of their respective values at successive timestamps still qualifies to the matching conditions. The latter step is essentially a "horizontal expansion" along the time axis in an attempt to eagerly verify and report pairs.

Based on the above, it is straightforward to derive a baseline algorithm for pair discovery over a set of time series. Specifically, this is done by checking all the candidate pairs formed at each timestamp, and verifying whether the minimum duration constraint δ is satisfied. However, searching over all timestamps, until a pair is verified or pruned, yields unnecessary computations, which is inefficient for large datasets of long time series. To tackle this, we describe next an optimization that identifies candidate pairs at selected timestamps only, so that only those pairs require verification.

To prune the search space, we introduce checkpoints along the time axis. Searching for candidate pairs is then performed at these specific timestamps only. The idea is that if the temporal span between two successive checkpoints does not exceed the minimal duration threshold δ, we can ensure no false negatives. Indeed, any qualifying pair starting at an intermediate timestamp between two consecutive checkpoints will certainly be detected at least on the second one. For example, in Figure 19, for $\delta = 4$, the pair starting at timestamp $t' - \delta + 1$ will be detected at the checkpoint placed at timestamp t'. Otherwise, its duration would be less than δ and it would be pruned. Based on this observation, it suffices to check for candidate pairs only at checkpoints, i.e., every δ timestamps. However, since we now skip over timestamps, we need to verify pairs with a horizontal expansion towards both directions, i.e. both before and after a given checkpoint. This is required to ensure that no false negatives occur.

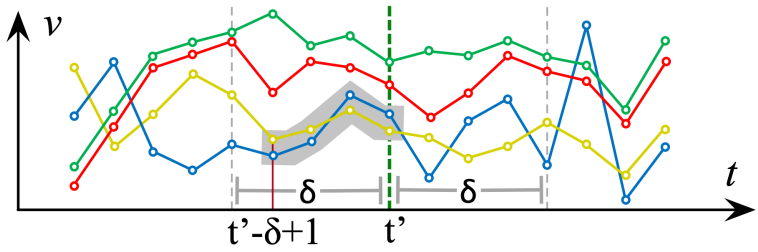


Figure 19: Checkpoints placed every δ timestamps.

Depending on the dataset, different checkpoint placement may yield different number of candidate pairs, thus affecting the number of verifications required. Intuitively, if the time series values at a specific timestamp are placed in a more "scattered" manner over the bins, then less candidates will be generated if this timestamp is considered. This is because the values of time series at that timestamp will differ from each other by more than ε, and will thus be pruned. Instead, placing checkpoints at "dense" areas will generate a larger number of candidates. To some extent, we can avoid such dense areas by shifting all checkpoints together, either to the left or to the right, while still maintaining their temporal span at δ.

3.4.4. Extension to geolocated time series

A time series may also be associated with a geographic location. We refer to such time series as geolocated time series. An example is a time series representing the revenue of a company, associated with the location of the company. In such cases, spatial distance may play an important role in the analysis, since discovery of trends and patterns may depend not only on time series similarity but also on geographic proximity.

In previous work, we have proposed a hybrid index, called BTSR-tree, that efficiently supports the retrieval of geolocated time series based on both spatial distance and time series similarity [97][98]. This is constructed by first building an R-tree over the locations of the time series data. Then, each node is enhanced with appropriate *Minimum Bounding Time Series* (MBTS) that enclose the subset

of time series represented by it. The structure of a BTSR-tree is illustrated in Figure 20. Combining MBTS and MBRs, the query evaluation algorithm can simultaneously prune the search space based on time series similarity and spatial distance while traversing the index. To further increase its pruning power, the BTSR-tree groups together similar time series within each node to derive tighter bounds. However, the search algorithm proposed in these works addresses the problem of retrieving time series that are similar in their entire duration.

Here, we have extended our previous approach on hybrid queries over geolocated time series to support local similarity. We present next an overview of our work; details can be found in [99]. The local similarity score σ between two time series is defined as the maximum number of consecutive timestamps during which their respective values do not differ by more than a user-specified threshold ε. On the one hand, compared to global similarity, this condition is more relaxed, in the sense that it is applied to subsequences rather than to the entire series. On the other hand, it is stricter in the sense that the threshold ε is applied at each individual timestamp of the matching subsequences rather than on the average distance over all timestamps. Combining this local similarity constraint with a condition on spatial distance leads to a set of hybrid queries.

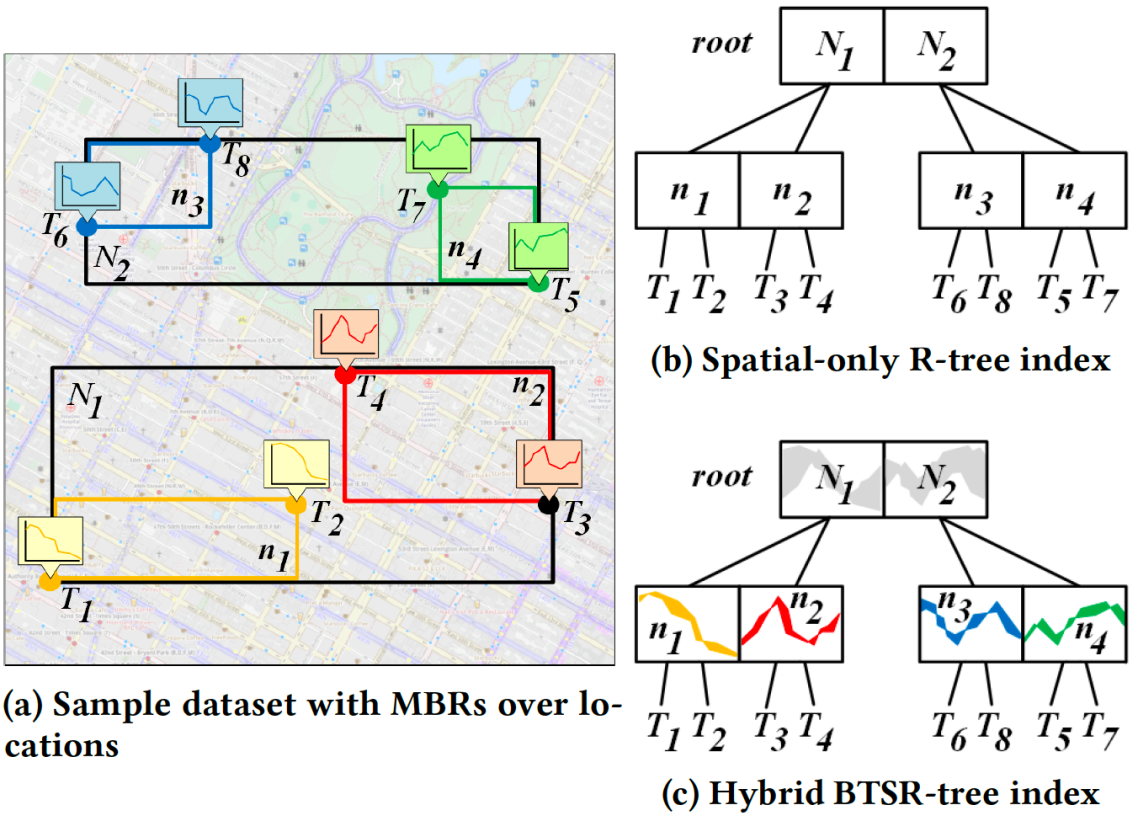


Figure 20: Example of the BTSR-tree index.

Figure 21 shows an example with a query time series T_q used to search over a set of time series $T_1, \ldots, T_9$. The goal is to detect those that are located within radius ρ from the location of T_q , while also being locally similar (i.e., having at least a given local similarity score) to T_q. In particular, given

a user-specified parameter ε, the subsequences are required to have a local similarity score at least $\sigma_i = 5$, i.e., to be within distance ε for at least $\delta = 5$ consecutive timestamps. We can see that, under these conditions, the qualifying results include T_2 , with local similarity score $\sigma_2 = 5$ (shown at the bottom of the figure) and T_7 , with $\sigma_7 = 7$ (shown at the top of the figure).

To evaluate such queries, we have developed two methods. The first method directly adapts the query processing algorithm over the BTSR-tree index to support the constraints for subsequence matching. In particular, this is achieved by modifying the algorithm to include the sweep-line method presented in the previous section, along with the use of checkpoints to generate and validate candidates. Although this method saves some computations, it does not fully exploit the pruning potential of the index. This is because it directly uses the BTSR-tree in its original form, which is designed for whole-sequence matching. Next, we explain how we adapt the index to prune more candidates, when performing similarity search over subsequences.

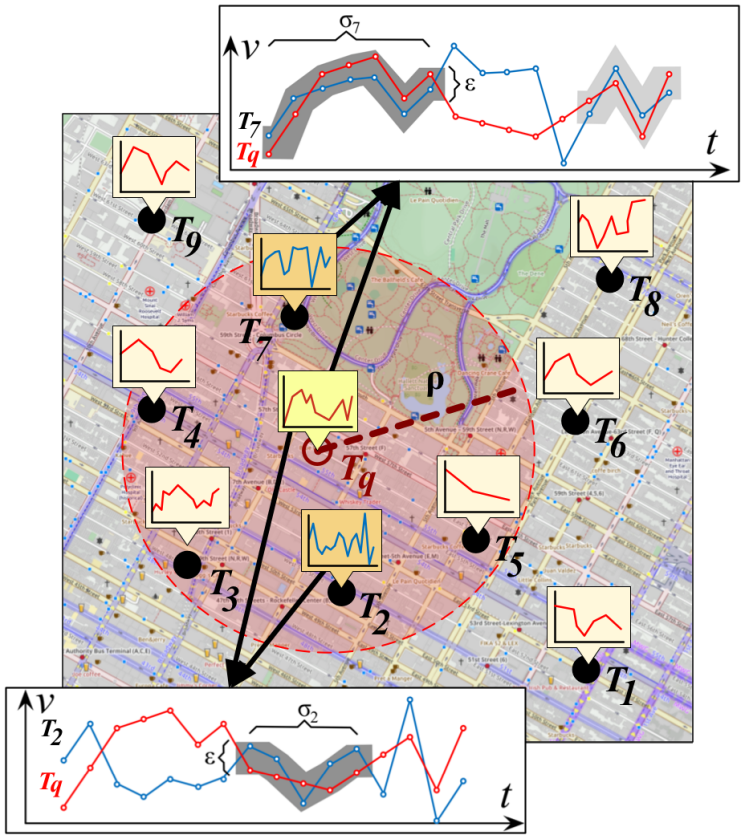
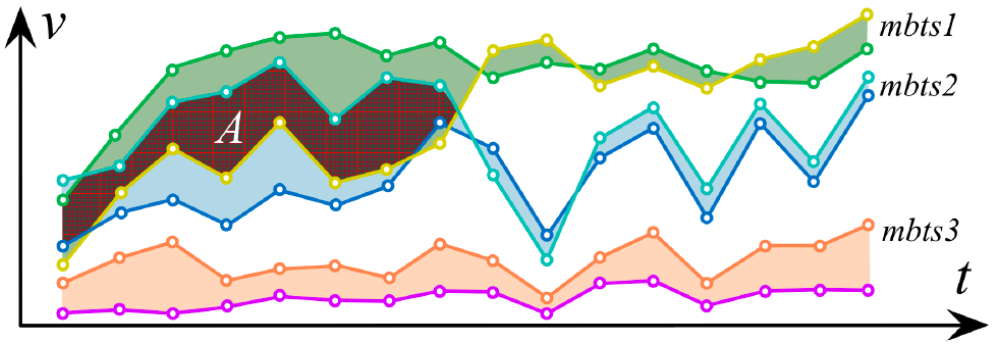


Figure 21: Similarity search over geolocated time series.

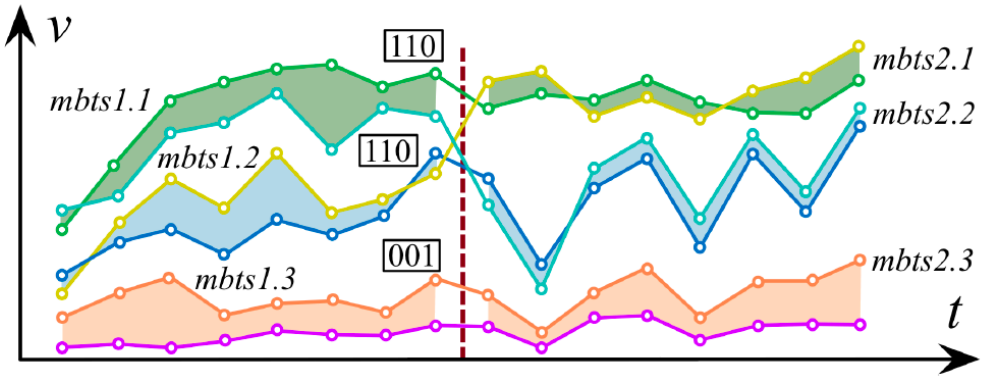
The BTSR-tree index uses k-means clustering to cluster the time series under each node and then stores the MBTS of those clusters. However, clustering entire time series typically generates many overlapping MBTS, incurring a lot of dead space. This has a negative impact on the pruning power of the index when considering local similarities (i.e., subsequence matching). Figure 22a depicts such a case of six time series indexed in a node. A k-means clustering with k = 3 will form the depicted MBTS denoted with shaded colors. As a result, the dark area A represents the overlap

between *mbts1* and *mbts2*, indicating that those bounds are less tight. Consequently, these MBTS inflate estimates for local similarity bounds, leading to a significantly higher number of false positives when generating candidates.

To reduce the amount of overlap within the MBTS of nodes, we adapt the BTSR-tree index, deriving a modified version called SBTSR-tree. SBTSR-tree eliminates a large part of the overlap illustrated above, by segmenting the indexed time series. Figure 22b depicts the intuition. By segmenting the time series before applying k-means clustering, the resulting MBTS for each segment become tighter, eliminating the excessive overlap A shown in Figure 22a. Besides that, the SBTSR-tree is built similarly to BTSR-tree. The only difference is that the MBTS of each node are calculated per segment instead of the entire sequence. As a result, SBTSR-tree enables more effective pruning when traversing the index to evaluate local similarity search queries.



(a) Example of a node's MBTS.



(b) Segmenting can eliminate whitespace.

Figure 22: Segmenting time series to create tighter bounds.

A downside of the segmentation approach is the loss of the MBTS continuity across time, which results in MBTS enclosing different time series in neighboring segments. For example, in Figure 22b, there are no MBTS in the right segment containing the same time series as *mbts1.1* and *mbts1.2*, which hinders the calculation of local similarity on the segment boundaries (the vertical line in the figure). To overcome this, we introduce a bit-vector V along each MBTS of a segment,

having one bit for each created MBTS. In the current segment, if a bit in vector V of a given MBTS is set, this indicates that this MBTS encloses at least one common time series with another MBTS in the next segment. In the example shown in Figure 22b, $V = 110$ for *mbts1.1*, indicating common time series with *mbts2.1* and *mbts2.2* in the next segment, while $V = 001$ for *mbts1.3*, signifying common time series only with *mbts2.3*. This way, during query evaluation, we can easily identify all MBTS that share common time series among two successive segments.

4. Entity Resolution

4.1. Introduction

Entity resolution refers to the task of identifying and linking entities across (possibly) different data sources that refer to the same real-world entity. Other names for entity resolution in the literature include record linkage, entity matching and entity deduplication [8]. Entity resolution is a crucial task in data integration and data cleaning, especially when the duplicated entities do not share a common identifier or have differing information, such as misspelling errors, or missing fields. The relevance of entity resolution has motivated the development of solutions on variations of the problem as it can be observed in several surveys [57][58][59].

A naive approach to perform entity resolution is to compute the similarity between all pairs of data available in the sources. If the similarity is greater than a predefined threshold, the pair of data is considered to be the same entity. Given this approach, some common techniques are:

- **Blocking techniques**. These reduce the pairwise similarity computation of data by grouping entities that are likely to match. Several recent surveys exist, presenting a comparative analysis of the state-of-the-art blocking techniques [60][8].
- **Similarity function**. There exist a variety of attribute similarity functions used for entity resolution, often based on string matching. Depending on the data domain, other similarity measures might be used [57]. For graphs, similarity measures that reflect some structural similarity between the entities are useful. In this context, a well-known similarity measure is SimRank, following the concept that "two objects are similar if they are related to similar objects" [58][59].
- **Matching**. Matching refers to techniques to decide if two entities are the same. These matching techniques include rule-based methods (i.e., according to a threshold), supervised, learning-based methods and unsupervised (clustering)-based methods [60]. An evaluation of both learning and non-learning-based methods for real-world matching problems is presented in [61].

In SmartDataLake, we address entity resolution in the context of HINs, using the *property graph* abstraction. Different approaches have been proposed to perform entity resolution on graphs in different levels of specifications. Some of the main approaches found in the literature are:

- **Network (user) alignment**. In network alignment, the goal is to match users from different social networks that refer to the same person (entity resolution in social networks). According to a recent survey [62], most of the existing methods for network user alignment can be generalized into a two-step framework: (1) feature extraction, and (2) model construction. In the first step, features from the users/profiles are extracted and serve as input to the model. Features used by these systems are profile features, content features (i.e., activities) and network features. The network alignment can have different approaches from general solutions for entity resolution due to the speciality of the online social network scenario.
- **Ontology alignment**. In ontology alignment, the goal is to find an equivalent representation of the same idea in different ontologies. Several ontology matching algorithms and frameworks have been proposed in the literature [63][64].
- **Graph dependencies**. This area focuses on defining classes of dependencies for graph data such as graph functional dependencies (GFDs), graph entity dependencies (GEDs), and graph differential dependencies (GDDs). One of the applications of the proposed graph dependencies is entity resolution [65][66][67][68].

To perform entity resolution in heterogeneous graph data sources, we focus on graph dependencies and propose a new class of graph dependencies called *Graph Generating Dependencies* (GGDs). Alike previously proposed graph dependencies, one of the applications of GGDs is entity resolution. In the following, we present the concept of GGDs, related work, and how to utilize GGDs for this task.

4.2. Graph Generating Dependencies

Data dependencies, such as functional dependencies (FDs) have been studied for relational databases and have been applied in integrity checking, query optimization and prevention of update anomalies. The definition of dependencies for graph data is also of great interest; however, it is challenging as graphs do not always have a defined schema, in contrast to relational data [66].

Recently, different classes of dependencies for graphs have been proposed, such as GFDs [66], GEDs [67], [69] and GDDs [65]. These types of dependencies extend the idea of functional dependencies, conditional functional dependencies and differential dependencies from relational data to graph data. However, the proposed graph dependencies in the literature cannot fully express tuple-generating dependencies (TGDs) for graph data and are only defined for a single graph pattern.

Moreover, these dependencies focus on generalizing FDs (i.e., variations of equality-generating dependencies) and cannot capture TGDs for graph data [67][69]. As an example, we might want to enforce the following constraint on a human resources graph: *"if two people-vertices have the same name and address property-values and they both have a works-at-edge to the same company-vertex, then there should be a same-as edge between these two people."* This is an example of a TGD on graph data, as the satisfaction of the constraint requires the existence of an

edge (i.e., the same-as edge), and, when not satisfied, we repair the graph by generating same-as edges where necessary. TGDs are important for many applications, e.g., for entity resolution during data cleaning and integration.

Indeed, TGDs arise naturally in graph data management applications. Given the lack of TGDs for graphs in the current study of graph dependencies, we propose a new class of graph dependencies called Graph Generating Dependencies (GGDs). These fully support TGDs for property graphs (i.e., graphs where vertices and edges have property values, such as names and addresses). They also generalize earlier graph dependencies. Informally, a GGD expresses a constraint between two (possibly) different graph patterns enforcing relationships between property values and topological structure. In the following section, we present related works on GGDs as well as its definition and validation process.

4.2.1. Related work

We can categorize the related work in two main classes: (a) dependencies proposed for relational data and (b) graph dependencies:

- **Relational data dependencies**. FDs have been widely studied in the area of dependencies for relational data. Their applications include query optimization and integrity checking. Other classes of dependencies that extend the concept of FDs were proposed in the literature. Some of these dependencies are Conditional Functional Dependencies (CFDs) and Differential Dependencies (DDs).
    - CFDs [70] are proposed for data cleaning tasks, and their main idea is to enforce an FD only for a set of tuples specified by a condition, unlike the original FDs in which the dependency holds for the whole relation.
    - DDs [71] extend FDs by specifying constraints according to the difference in the distance between two tuple attribute values.
- **Graph dependencies**. Some of the works in the literature focus on defining FDs for RDF data. More related to the proposed GGDs are the GFDs, GEDs and GDDs. Specifically:
    - GFDs [66] are formally defined as a pair $(Q[\overline{x}], X \rightarrow Y)$, where $Q[\overline{x}]$ is a graph pattern that defines topological constraints, while $X \rightarrow Y$ are two sets of literals that define the attribute-value dependencies of the GFD. The attribute-value dependency is defined for the vertices attributes present in the graph pattern.
    - GEDs [67] subsume the GFDs and can express FDs, GFDs, and EGDs. Besides the attribute-value dependencies present in GFDs, GEDs also carry special id literals to identify vertices in a graph pattern.
    - GDDs [65] extend GEDs by introducing distance functions instead of equality functions similar to DDs for relational data but defined over a topological constraint.
    - Graph Repairing Rules (GRRs) [72] have a similar definition to our proposed GGDs but different semantics. GRRs were proposed as an automatic semantic for graphs.

The semantic of a GRR is given a source graph pattern; it should be repaired to a target graph pattern.

- Graph-pattern association rules (GPAR) [73] propose association rules for graph patterns and have been applied to social media marketing. According to [74], GPARs are a special case of TGDs for graphs. A GPAR is defined as $Q(x,y) \Rightarrow q(x,y)$ in which, if there exists a subgraph isomorphic match of the graph pattern $Q(x,y)$, an edge labelled q between the vertices x, y is likely to hold.

The main differences of our proposed GGDs compared to the previous works include the use of differential constraints, edges are treated as first-class citizens in the graph patterns (in alignment with the property graph model), and the ability to entail new nodes and edges. With these new features of GGDs, we can encode a relation between two graph patterns as well as the (dis)similarity between its vertices and edges properties values.

4.2.2. Preliminaries

We first summarize standard notations and concepts. Let O be a set of objects, L be a finite set of labels, K be a set of property keys, and N be a set of values. We assume these sets to be pairwise disjoint [71][74][75]. A **property graph** is a structure $(V, E, \eta, \lambda, \nu)$ where:

- $V \subseteq O$ is a finite set of objects, called vertices,
- $E \subseteq O$ is a finite set of objects, called edges,
- $\nu: E \to V \times V$ is a function assigning to each edge an ordered pair of vertices,
- $\lambda: V \cup E \to P(L)$ is a function assigning to each object a finite set of labels (i.e., $P(S)$ denotes the set of finite subsets of set S). Abusing notation, we use λ_v for the function assigning labels to vertices and λ_e for the function assigning labels to edges, and
- $\nu: (V \cup E) \times K \to N$ is a partial function assigning values for properties/attributes to objects, such that the object sets V and E are disjoint (i.e., $V \cap E = \emptyset$) and the set of domain values where ν is defined is finite.

A **graph pattern** is a directed graph $Q[\overline{x}] = (V_Q, E_Q, \lambda_Q)$ where V_Q and E_Q are finite sets of pattern vertices and edges, respectively, and λ_Q is a function that assigns a label $\lambda_Q(u)$ to each vertex $u \in V_Q$ or edge $e \in E_Q$. Abusing notation, we use λ_{VQ} as a function to assign labels to vertices and λ_{EQ} to assign labels to edges. Additionally, $\overline{x}$ is a list of variables that includes all the vertices in V_Q and edges in E_Q.

We say a label l **matches** a label $l' \in L$, denoted as $l \asymp l'$, if $l \in L$ and $l = l'$ or $l =$ `$-$' (wildcard). A match denoted as $h[\overline{x}]$ of a graph pattern $Q[\overline{x}]$ in a graph G is a homomorphism of $Q[\overline{x}]$ to G such that for each vertex $u \in V_Q, \lambda_{VQ}(u) \asymp \lambda_{VQ}(h(u))$; and for each edge $e = (u, u') \in E_Q$, there exists an edge $e' = (h(u), h(u'))$ and $\lambda_{EQ}(e) \asymp \lambda_{EQ}(e')$.

A **differential function** $\phi[A]$ on attribute A is a constraint of difference over A according to a distance metric [71]. Given two tuples t_1, t_2 in an instance I of relation R, $\phi[A]$ is true if the difference between $t_1.A$ and $t_2.A$ agrees with the constraint specified by $\phi[A]$, where $t_1.A$ and

$t_2.A$ refers to the value of attribute A in tuples t_1 and t_2, respectively. We use the differential function idea to define constraints in GGDs.

4.2.3. Syntax and semantics

A GDD is a dependency of the form:

$$Q_s[\overline{x}], \phi_s \rightarrow Q_t[\overline{x}, \overline{y}], \phi_t$$

where:

- $Q_s[\overline{x}]$ and $Q_t[\overline{x}, \overline{y}]$ are graph patterns, called **source** graph pattern and **target** graph pattern, respectively,
- ϕ_s is a set of differential constraints defined over the variables $\overline{x}$ (variables of the graph pattern Q_s), and
- 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.

A differential constraint in ϕ_s on $\overline{x}$ (resp., in ϕ_t on $[\overline{x}, \overline{y}]$) is a constraint of one of the following forms [65][71]:

1. $\delta_A(x.A, c) \leq t_A$
2. $\delta_{A_1A_2}(x.A_1, x'.A_2) \leq t_{A_1A_2}$
3. $x = x'$ or $x \neq x'$

where $x, x' \in \overline{x}$ (resp. $\in \overline{x} \cup \overline{y}$) for $Q_s[\overline{x}]$ (resp. for $Q_t[\overline{x}, \overline{y}]$), δ_A is a user-defined similarity function for the property A, and $x.A$ is the property value of variable x on A, c is a constant of the domain of property A and t_A is a pre-defined threshold. The differential constraints defined by (1) and (2) can use the operators $(=, <, >, \leq, \geq, \neq)$. The user-defined distance function δ_A can be, for example, an edit distance when A is a string or the difference between two numerical values.

The constraint (3) $x = x'$ states that x and x' are the same entity (vertex/edge) and can also use the inequality operator stating that $x \neq x'$. Since the pattern variables $\overline{x}$ in Q_s (resp. $\overline{x}, \overline{y}$ in Q_t) include both, vertices and edges, this allows to match vertex-vertex-, edge-edge- and vertex-edge-variables.

Figure 23 shows examples of GGDs. Here, σ_1 implies that for the matches of the source graph pattern Q_s, if the student type is "high school"', there exists a target graph pattern Q_t, in which the same matched vertex for teacher has an edge labelled "works" to a "high school" vertex in which the difference/(dis)similarity between the high school name and the student school name should be less than or equal to 1. According to σ_2, for the matches of Q_s if the project department and the department name are (dis)similar according to the threshold "2", there exists an edge labelled "manages" linking the department and the project (graph pattern Q_t).

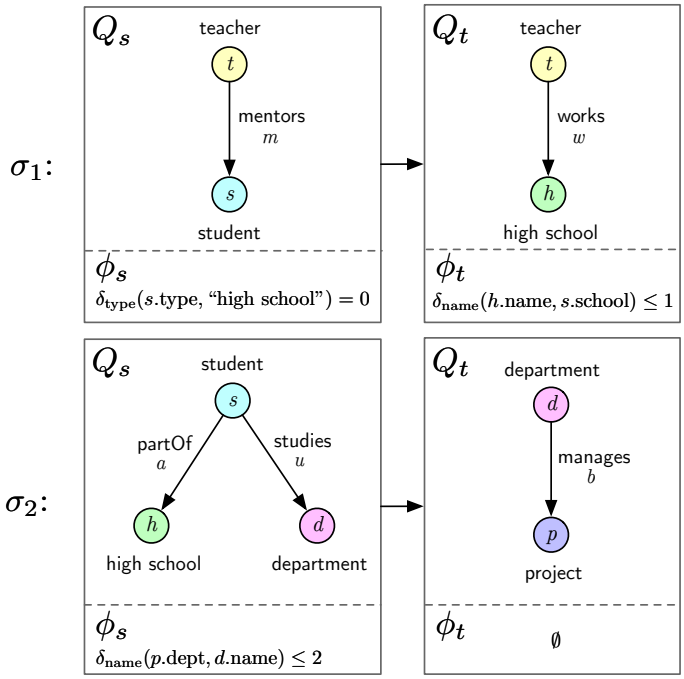


Figure 23: Examples of GGDs (I).

4.2.4. Semantics of GGDs

In order to interpret a GGD $Q_s[\overline{x}], \phi_s \rightarrow Q_t[\overline{x}, \overline{y}], \phi_t$, we first specify what it means for a graph pattern match to satisfy a set of differential constraints. Consider a graph pattern $Q[\overline{z}]$, a set of differential constraints ϕ_z and a match of this pattern represented by $h[\overline{z}]$ in a graph G. The match $h[\overline{z}]$ satisfies ($\vDash$) a differential constraint $k \in \phi_z$, if:

1. When k is $\delta_A(z.A, c) \leq t_A$ then an attribute $z.A$ exists at vertex/edge $z = h(z)$ and $\delta_A(z.A, c) \leq t_A$ meaning that the user-defined distance (for property A) δ_A between a constant c and the attribute A value of vertex/edge z is less or equal than the defined threshold t_A.
2. When k is $\delta_{A_1A_2}(z.A_1, z'.A_2) \leq t_{A_1A_2}$ then the attributes A_1, A_2 exist at vertex/edge $z = h(z)$ and $z' = h(z')$ and $\delta_{A_1\,A_2}(z.A_1, z'.A_2) \leq t_{A_1A_2}$.
3. When k is $z = z'$, then $h(z)$ and $h(z')$ refer to the same vertex/edge.

The match $h[\overline{z}]$ satisfies ϕ_z, denoted as $h[\overline{z}] \vDash \phi_z$ if the match $h[\overline{z}]$ satisfies every differential constraint in ϕ_z. If $\phi_z = \{\emptyset\}$ then $h[\overline{z}] \vDash \phi_z$ for any match of the graph pattern $Q[\overline{z}]$ in G.

Given a GGD $Q_s[\overline{x}], \phi_s \rightarrow Q_t[\overline{x}, \overline{y}], \phi_t$ we denote the matches of the source graph pattern $Q_s[\overline{x}]$ as $h_s[\overline{x}]$ while the matches of the target graph pattern $Q_t[\overline{x}, \overline{y}]$ are denoted by $h_t[\overline{x}, \overline{y}]$ which can include the variables from the source graph pattern $\overline{x}$ and additional variables $\overline{y}$ particular to the target graph pattern $Q_t[\overline{x}, \overline{y}]$.

Figure 24 shows further examples of GGDs.

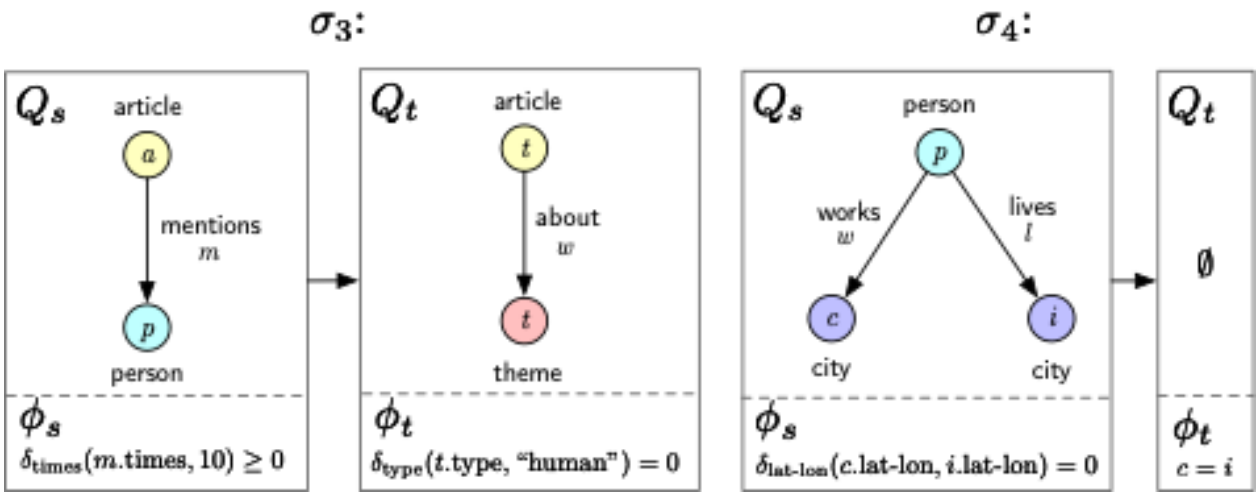


Figure 24: Examples of GGDs (II).

In the GGD σ_3, following the semantics of GGDs, for every match of Q_s where an article mentions a person more than 10 times, there exists a match of Q_t such that the theme type is “human”. Observe that, in this example, we use the property value of the edge variable m in the differential constraint, which is possible in GGDs as edges are also considered variables in the graph patterns.

The GGD σ_4 enforces that, if the latitude and longitude coordinates of the city c, in which a person works, and of the city i, in which a person lives are the same, c and i should refer to the same city. Observe that in this case, the target graph pattern is empty.

4.2.5. Validation

We next discuss the **validation problem** for GGDs, defined as follows: given a finite set Σ of GGDs and a non-empty graph G, does the set of GGDs Σ holds in G, denoted as $G \models \Sigma$?

To validate a set Σ of GGDs is the same as validating each $\sigma \in \Sigma$ in the graph G (i.e., $G \models \sigma$ for each $\sigma \in \Sigma$). We propose an algorithm to validate a GGD $\sigma = Q_s[\overline{x}], \phi_s \rightarrow Q_t[\overline{x}, \overline{y}], \phi_t$, which is illustrated in Figure 25. The algorithm returns true if σ is validated and returns false if σ is violated.

We proceed as follows. For each match $h_s(\overline{x})$ of the graph pattern $Q_s[\overline{x}]$ in G:

1. Check if $h_s(\overline{x})$ satisfies the source constraints (i.e., $h_s(\overline{x}) \models \phi_s$). If yes, then continue.
2. Retrieve all matches $h_t(\overline{x}, \overline{y})$ of the target graph pattern $Q_t[\overline{x}, \overline{y}]$ where $h_s(x) = h_t(x)$ for all $x \in \overline{x}$. If there are no such matches of the target graph pattern, return false.
3. Verify if $h_t(\overline{x}, \overline{y}) \models \phi_t$. If there exists at least one match of the target graph pattern such that $h_t(\overline{x}, \overline{y}) \models \phi_t$, then return true, else return false.

This process is repeated for each $\sigma \in \Sigma$. For each match on which σ is violated, new vertices/edges can be generated to repair it (i.e., to make the GGD σ valid on G).

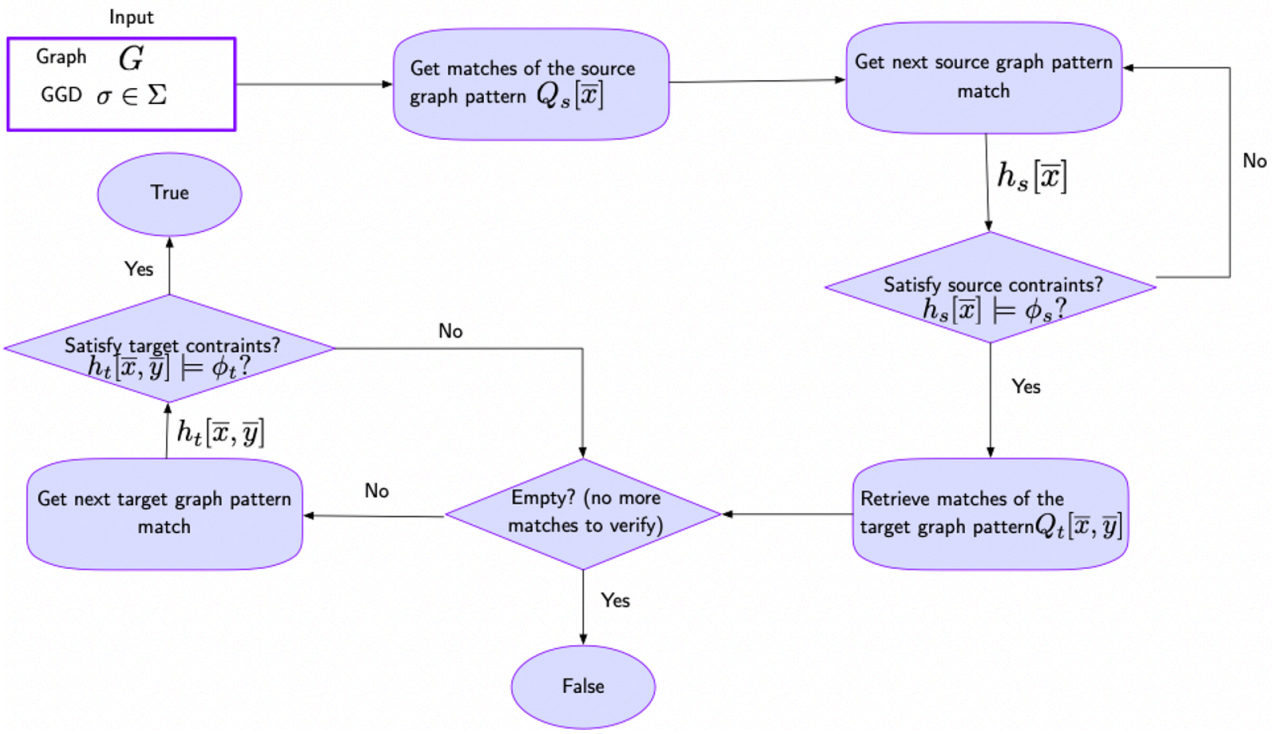


Figure 25: Validation algorithm overview.

4.3. GGDs for entity resolution

The main novelty of GGDs is in the generation of new vertices or edges in case a GGD is violated. Given this feature, GGDs can be applied in different scenarios. In this section, we show how GGDs can be used in solutions for the entity resolution task [76][77].

As mentioned beforehand, entity resolution refers to identifying and linking entities across (possibly different) data sources that refer to the same real-world entity [76]. The generation of new vertices and edges in case a GGD is violated gives the possibility to rewrite matching rules or conditions as a GGD. To solve the task, we can define the source graph patterns as several disjoint patterns from (possibly) different graph sources and use the target graph pattern specifications as the representation of the deduplicated graphs. Thus, using this approach, we can also encode more information than just vertex-to-vertex, or row-to-row in relational databases, as we consider all the information in a defined graph pattern.

Observe the following figure. As discussed before, the source graph pattern encodes the rules to perform entity resolution over (possibly) different graph sources. To perform entity resolution, we

can add links of type "sameAs" between the matched entities in the target graph pattern. These links are generated to validate the defined GGD.

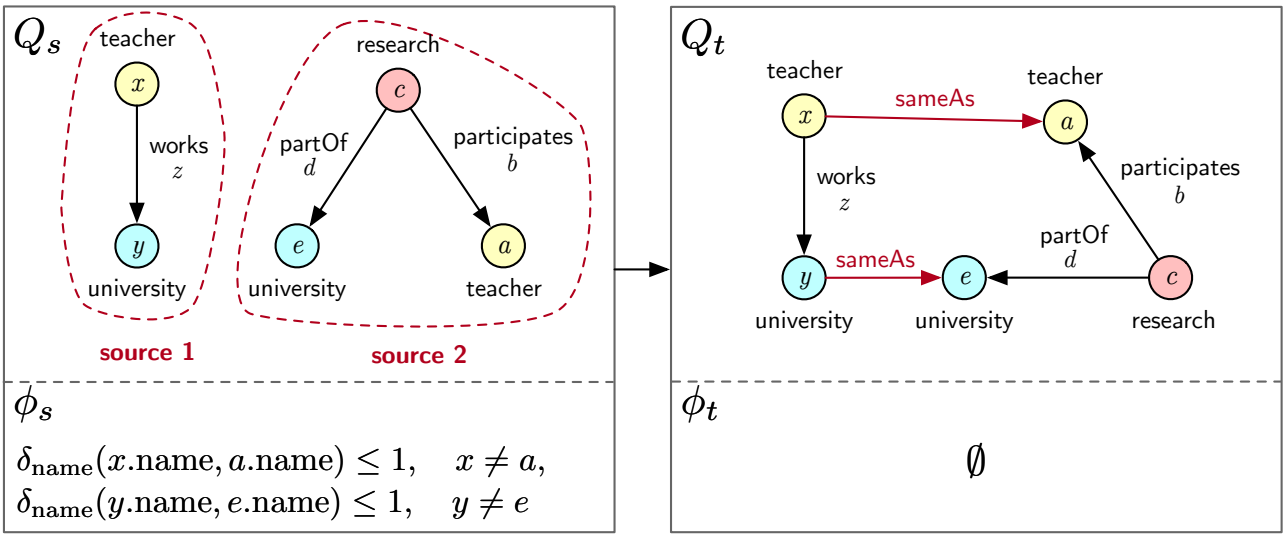


Figure 26: Example of using GGDs for Entity Resolution adding "sameAs" links.

A second case in which the GGDs can be used for entity resolution is when two graph patterns that refer to the same real-world entity have different structures in (possibly) different sources. In this case, with GGDs, we can generate a vertex or a graph pattern that can summarize all the information from these two graph patterns (see Figure 27). An advantage of GGDs is the use of edges as variables, allowing to use the information of edge properties also in the matching rules, as it can be observed in the next example.

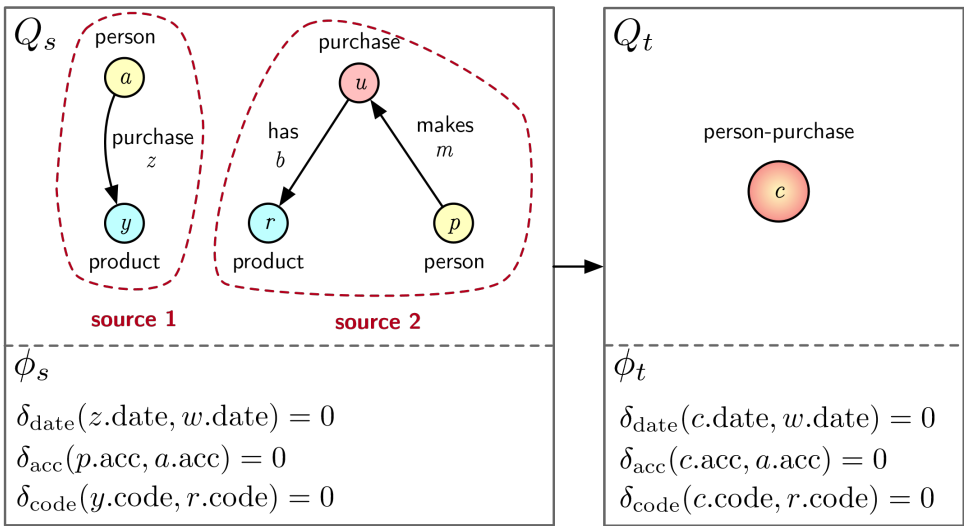


Figure 27: Example of GGDs for Entity Resolution using aggregation.

In this case, we have two graph sources that model the same act of purchasing a product differently. To deduplicate this data, it is useful to create a vertex in the integrated graph that is able to aggregate the information that matches in both sources.

Given these examples on how GGDs can be used, the general process is described next. Here, we modify the presented Validation algorithm to return the matches of the source graph pattern that violates the GGDs when a GGD is not validated (when Validation algorithm returns false).

Given a set Σ of GGDs and two different graph data sources, the output is an integrated graph according to the defined GGDs. The algorithm works as follows:

1. For each GGD in the defined set, verify if the GGD is valid by using the Validation algorithm proposed.
2. If the GGD is not valid, return the matches of the source graph pattern query that violates the GGD.
3. For each match that violates the GGD, repair it by generating new nodes and edges to make the GGD valid.

The output integrated graph contains all the data from source graphs along with the generated nodes and edges in the repair process. We summarize this process of performing entity resolution on different graph data sources in Figure 28.

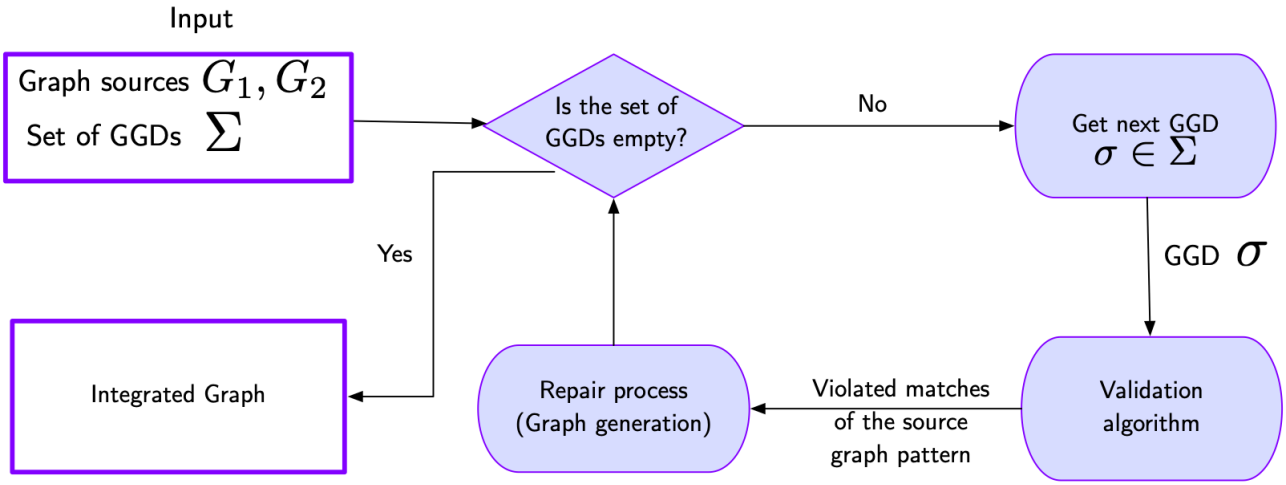


Figure 28: Entity Resolution process using GGDs.

According to [76], *repairs errors* refers to the task of finding another data instance that follows the set of data quality rules. Several error repair methods were proposed for relational data dependencies [76][78]. In the case of GGDs, the set of data quality rules are the defined GGDs Σ and the instances that should be repaired are related to the source graph pattern matches returned by the validation algorithm. The repairing process of the GGDs, as mentioned

beforehand, can generate new nodes and edges to validate a GGD. To repair and create the integrated graph, we retrieve candidate nodes/edges from the data sources which satisfy the target constraints and evaluate which nodes/edges should be generated to validate the declared GGD.

5. Conclusions

In this report, we have presented our ongoing work on entity ranking, similarity search and entity resolution, carried out in Tasks 3.1 and 3.2 of the SmartDataLake project. For entity ranking in HINs, our approach allows computing different ranking scores that are relative to the metapaths under consideration. Moreover, for the case of geospatial entities, we have developed a parallel and distributed method for identifying top-k rectangular spatial regions according to user-defined objective functions. We have then focused on similarity search and join operations. We have reviewed the state of the art, in particular concerning efficient algorithms following the filter-verification framework. Our work focuses mainly on fuzzy set similarity joins, especially for identifying top-k results progressively. We also allow sets to have different weights, so that ranking scores can be incorporated in the search process. Moreover, we have described our approach for similarity search over entities with heterogeneous attributes, including numerical, textual and spatial properties. In addition, we have developed algorithms for similarity search on time series, to address the case of entities with attribute values that change over time. Finally, we have introduced a novel approach for entity resolution relying on the concept of graph generating dependencies. We have presented related work on relational data dependencies and graph dependencies, defined the main concepts, syntax and semantics, and explained how these graph generating dependencies can be used for entity resolution.

Based on these approaches and methods, we are currently working on the implementation and evaluation of the respective components, which will be presented in Deliverable D3.2.

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] Y. Sun, J. Han, X. Yan, P. Yu, T. Wu: PathSim: Meta Path-Based Top-K Similarity Search in Heterogeneous Information Networks. PVLDB 4(11): 992-1003 (2011)

[3] W. Tobler: A computer movie simulating urban growth in the Detroit region. Economic Geography, 46(Supplement): 234–240 (1970)

[4] K. Feng, G. Cong, S. Bhowmick, W. Peng, C. Miao. Towards Best Region Search for Data Exploration. SIGMOD Conference 2016: 1055-1070

[5] D. Skoutas, D. Sacharidis, K. Patroumpas. Efficient progressive and diversified top-k best region search. SIGSPATIAL 2018: 299-308

[6] 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

[7] N. Augsten, M. Böhlen. Similarity Joins in Relational Database Systems. Morgan & Claypool Publishers, 2013

[8] G. Papadakis, D. Skoutas, E. Thanos, T. Palpanas. Blocking and Filtering Techniques for Entity Resolution: A Survey. ACM Comput. Surv. 53(2) (2020)

[9] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In VLDB, pages 491–500, 2001

[10] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In SIGMOD, pages 743–754, 2004

[11] T. Bocek, E. Hunt, and B. Stiller. Fast Similarity Search in Large Dictionaries. Technical Report ifi-2007.02, Department of Informatics, University of Zurich, April 2007. http://fastss.csg.uzh.ch/

[12] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In ICDE, pages 5–5, 2006

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

[14] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In ICDE, pages 257–266, 2008

[15] C. Xiao, W. Wang, and X. Lin. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. PVLDB, 1(1):933–944, 2008

[16] J. Qin, W. Wang, Y. Lu, C. Xiao, and X. Lin. Efficient exact edit similarity query processing with the asymmetric signature scheme. In SIGMOD, pages 1033–1044, 2011

[17] W. Wang, J. Qin, C. Xiao, X. Lin, and H. T. Shen. Vchunkjoin: An efficient algorithm for edit similarity joins. IEEE TKDE, 25(8):1916–1929, 2013

[18] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In WWW, pages 131–140, 2008

[19] L. A. Ribeiro and T. Härder. Generalizing prefix filtering to improve set similarity joins. Inf. Syst., 36(1):62–78, 2011

[20] P. Bouros, S. Ge, and N. Mamoulis. Spatio-textual similarity joins. PVLDB, 6(1):1–12, 2012

[21] J. Wang, G. Li, and J. Feng. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In SIGMOD, pages 85–96, 2012

[22] X. Wang, L. Qin, X. Lin, Y. Zhang, and L. Chang. Leveraging set relations in exact set similarity join. PVLDB, 10(9):925–936, 2017

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

[24] E. Zhu, D. Deng, F. Nargesian, and R. J. Miller. JOSIE: overlap set similarity search for finding joinable tables in data lakes. In SIGMOD, pages 847–864, 2019

[25] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In VLDB, pages 918–929, 2006

[26] G. Li, D. Deng, J. Wang, and J. Feng. PASS-JOIN: A partition-based method for similarity joins. PVLDB, 5(3):253–264, 2011

[27] D. Deng, G. Li, H. Wen, and J. Feng. An efficient partition based method for exact set similarity joins. PVLDB, 9(4):360–371, 2015

[28] Z. Zhang, M. Hadjieleftheriou, B. C. Ooi, and D. Srivastava. Bed-tree: an all-purpose index structure for string similarity search based on edit distance. In SIGMOD, pages 915–926, 2010

[29] W. Lu, X. Du, M. Hadjieleftheriou, and B. C. Ooi. Efficiently supporting edit distance based string similarity search using B+-trees. IEEE TKDE, 26(12):2983–2996, 2014

[30] Y. Zhang, X. Li, J. Wang, Y. Zhang, C. Xing, and X. Yuan. An efficient framework for exact set similarity search using tree structure indexes. In ICDE, pages 759–770, 2017

[31] J.Wang, G. Li, and J. Feng. Trie-join: Efficient trie-based string similarity joins with edit-distance constraints. PVLDB, 3(1):1219–1230, 2010

[32] M. Yu, J. Wang, G. Li, Y. Zhang, D. Deng, and J. Feng. A unified framework for string similarity search with edit-distance constraint. VLDB J., 26(2):249–274, 2017

[33] Y. Zhang, J. Wu, J. Wang, and C. Xing. A transformation-based framework for knn set similarity search. IEEE TKDE, 2018

[34] F. Afrati, A. D. Sarma, D. Menestrina, A. Parameswaran, and J. Ullman. Fuzzy joins using mapreduce. In ICDE, pages 498–509, 2012

[35] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In SIGMOD, pages 495–506, 2010

[36] C. Rong, W. Lu, X. Wang, X. Du, Y. Chen, and A. K. H. Tung. Efficient and scalable processing of string similarity join. IEEE TKDE, 25(10):2217–2230, 2013

[37] C. Rong, C. Lin, Y. N. Silva, J.Wang,W. Lu, and X. Du. Fast and scalable distributed set similarity joins for big data analytics. In ICDE, pages 1059–1070, 2017

[38] J. Zhai, Y. Lou, and J. Gehrke. ATLAS: a probabilistic algorithm for high dimensional similarity search. In SIGMOD, pages 997–1008, 2011

[39] V. Satuluri and S. Parthasarathy. Bayesian locality sensitive hashing for fast similarity search. PVLDB, 5(5):430–441, 2012

[40] T. Christiani, R. Pagh, and J. Sivertsen. Scalable and robust set similarity join. In ICDE, pages 1240–1243, 2018

[41] J. Wang, X. Yang, B. Wang, and C. Liu. Ls-join: Local similarity join on string collections. IEEE TKDE, 29(9):1928–1942, 2017

[42] P. Wang, C. Xiao, J. Qin, W. Wang, X. Zhang, and Y. Ishikawa. Local similarity search for unstructured text. In SIGMOD, pages 1991–2005, 2016

[43] W. Tao, D. Deng, and M. Stonebraker. Approximate string joins with abbreviations. PVLDB, 11(1):53–65, 2017

[44] J. Wang, G. Li, and J. Feng. Extending string similarity join to tolerant fuzzy token matching. ACM TODS, 39(1):7:1–7:45, 2014

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

[46] J. Wang, C. Lin, and C. Zaniolo. Mf-join: Efficient fuzzy string similarity join with multi-level filtering. In ICDE, pages 386–397, 2019

[47] G. Li, J. He, D. Deng, and J. Li. Efficient similarity join and search on multi-attribute data. In SIGMOD, pages 1137–1151, 2015

[48] P. Suganthan, A. Ardalan, A. Doan, and A. Akella. Smurf: Self-service string matching using random forests. PVLDB, 12(3):278–291, 2018

[49] P. Xu and J. Lu. Towards a unified framework for string similarity joins. PVLDB, 12(11):1289–1302, 2019

[50] W. Mann, N. Augsten, P. Bouros. An Empirical Evaluation of Set Similarity Join Techniques. PVLDB 9(9): 636-647 (2016)

[51] 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)

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

[53] A. N. Langville and C. D. Meyer, Google's PageRank and beyond: The science of search engine rankings. Princeton University Press, 2011.

[54] G. Jeh, J. Widom: Scaling personalized web search. WWW 2003: 271-279.

[55] 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).

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

[57] I. Bhattacharya and L. Getoor, "Collective entity resolution in relational data," ACM Trans. Knowl. Discov. Data, vol. 1, no. 1, Mar. 2007.

[58] P. Lee, L. Lakshmanan, J. Yu: On Top-k Structural Similarity Search. ICDE 2012: 774-785

[59] G. Jeh, J. Widom: SimRank: a measure of structural-context similarity. KDD 2002: 538-543

[60] M. Nentwig, M. Hartung, A. Ngonga Ngomo, E. Rahm: A survey of current Link Discovery frameworks. Semantic Web 8(3): 419-436 (2017)

[61] H. Köpcke, A. Thor, E. Rahm: Evaluation of entity resolution approaches on real-world match problems. PVLDB 3(1): 484-493 (2010)

[62] K. Shu, S. Wang, J. Tang, R. Zafarani, H. Liu: User Identity Linkage across Online Social Networks: A Review. SIGKDD Explorations 18(2): 5-17 (2016)

[63] L. Otero-Cerdeira, F. Rodríguez-Martínez, A. Gómez-Rodríguez: Ontology matching: A literature review. Expert Syst. Appl. 42(2): 949-971 (2015)

[64] P. Ochieng, S. Kyanda: Large-Scale Ontology Matching: State-of-the-Art Analysis. ACM Comput. Surv. 51(4): 75:1-75:35 (2018)

[65] S. Kwashie, J. Liu, J. Li, L. Liu, M. Stumptner, L. Yang: Certus: An Effective Entity Resolution Approach with Graph Differential Dependencies (GDDs). PVLDB 12(6): 653-666 (2019)

[66] W. Fan, Y. Wu, J. Xu: Functional Dependencies for Graphs. SIGMOD Conference 2016: 1843-1857

[67] W. Fan, P. Lu: Dependencies for Graphs. ACM Trans. Database Syst. 44(2): 5:1-5:40 (2019)

[68] W. Fan, Z. Fan, C. Tian, X. Dong: Keys for Graphs. PVLDB 8(12): 1590-1601 (2015)

[69] W. Fan, P. Lu: Dependencies for Graphs. PODS 2017: 403-416

[70] P. Bohannon, W. Fan, F. Geerts, X. Jia, Anastasios Kementsietsidis: Conditional Functional Dependencies for Data Cleaning. ICDE 2007: 746-755

[71] S. Song, L. Chen: Differential dependencies: Reasoning and discovery. ACM Trans. Database Syst. 36(3): 16:1-16:41 (2011)

[72] Y. Cheng, L. Chen, Y. Yuan, G. Wang: Rule-Based Graph Repairing: Semantic and Efficient Repairing Methods. ICDE 2018: 773-784

[73] W. Fan, X. Wang, Y. Wu, J. Xu: Association Rules with Graph Patterns. PVLDB 8(12): 1502-1513 (2015)

[74] W. Fan: Dependencies for Graphs: Challenges and Opportunities. J. Data and Information Quality 11(2): 5:1-5:12 (2019)

[75] A. Bonifati, G. Fletcher, H. Voigt, N. Yakovets: Querying Graphs. Synthesis Lectures on Data Management, Morgan & Claypool Publishers 2018

[76] I. Ilyas, X. Chu: Data Cleaning. ACM 2019, ISBN 978-1-4503-7152-0, pp. 1-285

[77] W. Fan, F. Geerts: Foundations of Data Quality Management. Synthesis Lectures on Data Management, Morgan & Claypool Publishers 2012

[78] L. Bertossi: Database Repairing and Consistent Query Answering. Synthesis Lectures on Data Management, Morgan & Claypool Publishers 2011

[79] J. Fekete, R. Primet: Progressive Analytics: A Computation Paradigm for Exploratory Data Analysis. CoRR abs/1607.05162 (2016)

[80] K. Echihabi, K. Zoumpatianos, T. Palpanas, H. Benbrahim: The Lernaean Hydra of Data Series Similarity Search: An Experimental Evaluation of the State of the Art. PVLDB 12(2): 112-127 (2018)

[81] K. Chan, A. Fu: Efficient Time Series Matching by Wavelets. ICDE 1999: 126-133

[82] J. Lin, E. Keogh, L. Wei, S. Lonardi: Experiencing SAX: a novel symbolic representation of time series. Data Min. Knowl. Discov. 15(2): 107-144 (2007)

[83] J. Shieh, E. Keogh: iSAX: indexing and mining terabyte sized time series. KDD 2008: 623-631

[84] A. Camerra, T. Palpanas, J. Shieh, E. Keogh: iSAX 2.0: Indexing and Mining One Billion Time Series. ICDM 2010: 58-67

[85] A. Camerra, J. Shieh, T. Palpanas, T. Rakthanmanon, E. Keogh: Beyond one billion time series: indexing and mining very large time series collections with i SAX2+. Knowl. Inf. Syst. 39(1): 123-151 (2014)

[86] K. Zoumpatianos, S. Idreos, T. Palpanas: Indexing for interactive exploration of big data series. SIGMOD Conference 2014: 1555-1566

[87] H. Kondylakis, N. Dayan, K. Zoumpatianos, T. Palpanas: Coconut: A Scalable Bottom-Up Approach for Building Data Series Indexes. PVLDB 11(6): 677-690 (2018)

[88] D. Yagoubi, R. Akbarinia, F. Masseglia, T. Palpanas: Massively Distributed Time Series Indexing and Querying. IEEE Trans. Knowl. Data Eng. 32(1): 108-120 (2020)

[89] B. Peng, P. Fatourou, T. Palpanas: ParIS: The Next Destination for Fast Data Series Indexing and Query Answering. BigData 2018: 791-800

[90] M. Linardi, T. Palpanas: Scalable, Variable-Length Similarity Search in Data Series: The ULISSE Approach. PVLDB 11(13): 2236-2248 (2018)

[91] T. Rakthanmanon, B. Campana, A. Mueen, G. Batista, M. Westover, Q. Zhu, J. Zakaria, E. Keogh: Searching and mining trillions of time series subsequences under dynamic time warping. KDD 2012: 262-270

[92] C. Yeh, Y. Zhu, L. Ulanova, N. Begum, Y. Ding, H. Dau, D. Silva, A. Mueen, E. Keogh: Matrix Profile I: All Pairs Similarity Joins for Time Series: A Unifying View That Includes Motifs, Discords and Shapelets. ICDM 2016: 1317-1322

[93] R. Cole, D. Shasha, X. Zhao: Fast window correlations over uncooperative time series. KDD 2005: 743-749

[94] S. Papadimitriou, J. Sun, P. Yu: Local Correlation Tracking in Time Series. ICDM 2006: 456-465

[95] Y. Zhu, D. Shasha: StatStream: Statistical Monitoring of Thousands of Data Streams in Real Time. VLDB 2002: 358-369

[96] G. Chatzigeorgakidis, D. Skoutas, K. Patroumpas, T. Palpanas, S. Athanasiou, S. Skiadopoulos: Local Pair and Bundle Discovery over Co-Evolving Time Series. SSTD 2019: 160-169

[97] G. Chatzigeorgakidis, D. Skoutas, K. Patroumpas, S. Athanasiou, S. Skiadopoulos: Indexing Geolocated Time Series Data. SIGSPATIAL 2017: 19:1-19:10

[98] G. Chatzigeorgakidis, K. Patroumpas, D. Skoutas, S. Athanasiou, S. Skiadopoulos: Scalable hybrid similarity join over geolocated time series. SIGSPATIAL 2018: 119-128

[99] G. Chatzigeorgakidis, D. Skoutas, K. Patroumpas, T. Palpanas, S. Athanasiou, S. Skiadopoulos: Local Similarity Search on Geolocated Time Series Using Hybrid Indexing. SIGSPATIAL 2019: 179-188

[100] 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)